Media Engineering and Technology Faculty
German University in Cairo

Android Application for Translation Office

Bachelor Thesis

Author:	Mahmoud Abdelaziz Wagdy
Supervisor:	Assoc. Prof. Dr. Rimon Elias
Submission Date:	1 June, 2023

Media Engineering and Technology Faculty
German University in Cairo

Android Application for Translation Office

Bachelor Thesis

Author:	Mahmoud Abdelaziz Wagdy
Supervisor:	Assoc. Prof. Dr. Rimon Elias
Submission Date:	1 June, 2023

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowledgement has been made in the text to all other material used

Mahmoud Abdelaziz Wagdy
1 June, 2023

Acknowledgments

I would like to thank my Professor Dr. Rimon Elias for giving me the opportunity to work on this project and explore the world of Android development which I have always wanted to explore and work on but never had the chance to, as well as thank my colleagues who inspired me with new ideas to renovate my work and excel at making premium quality applications, as well as giving me ideas to support my work. I would also like to thank my friends and family for the constant support as well as inspiring me to come with better designs for the application, which is crucial as an outside perspective for how a user, not a developer, would want their application and experience using the application to be.

Abstract

This Thesis Report dives into how to develop an Android application, with the problem being that of developing an Android application for a translation company. This reports further includes the technologies used which were NodeJS, React Native, and MongoDB for the Backend, Frontend, and Database of the project respectively. The Application features 3 roles for the 3 types of users to use the application which are Client, Translator, and Admin, each with their own features available to use. The report then follows the path and the steps taken to create the frontend screens of the application as well as the Backend files required to communicate with the frontend screens to communicate with and update the database.

Contents

Chapter 1

Introduction

1.1 Problem

A translation company by the name of TranslateCo, Inc. wishes to expand its interface and its way to communicate with customers needs by developing an application such that it can handle its business digitally.

1.2 Aim

The aim of this project is to create a mobile application such that TranslateCo, Inc. can handle customer needs and communication without having in person reservations where through this application, communication between translators and clients can be made easier and almost effortless. This application was made using React Native and NodeJS technologies, with its database being uploaded on MongoDB atlas servers.
The following chapters are going to explain the project in more depths, where there are three more chapters which are Background, Implementation, and Conclusion and Future Work.

1.3 Organization

The Background chapter will go in to details regarding the technologies used in the application, how and what they are commonly used for, and how these technologies are related to each other. Some of the information in the Background chapter involves the definition of the technologies and the features that are included with these technologies to help with context.
The following chapter is the Implementation, and the Implementation chapter is to do

with how the technologies explained in the Background chapter are used in the application. The Implementation chapter will also go in depths about the format of the project code such as the file division and organization, as well as an explanation of what specific files do and their relation to other files.
Finally the Conclusion and Future Work chapter includes the final words related to the project as well as any details related to future changes or adaptations to the project as well as a recap of the key information presented in the previous two chapter.

Chapter 2

Background

2.1 Overview:

This project takes the form of a MERN stack application, where a MERN stack application consists of the technologies MongoDB, Express JS, Node JS, and React Native.

2.2 Frontend:

2.2.1 Overview:

This project uses React Native for its Frontend work of the Android application, where React Native is a Cross Platform/baseless Technology that can be used to develop applications for both Android and iOS. The term baseless refers to code base, where for example to make View container style for Andriod Native Android View would be used, but to do so in iOS native Window View would be used. React Native would just use the term View and it is automatically translated to the type of native needed once compiled. React Native is used in this application for the complete Frontend Design of the project. The type of React Native used in this application is Expo CLI, where Expo is a more open version of the React Native CLI which has more available features and bandwidth than the normal React Native CLI. To install Expo visit the Expo CLI official documentation to get the full process. [3].

2.2.2 React Native:

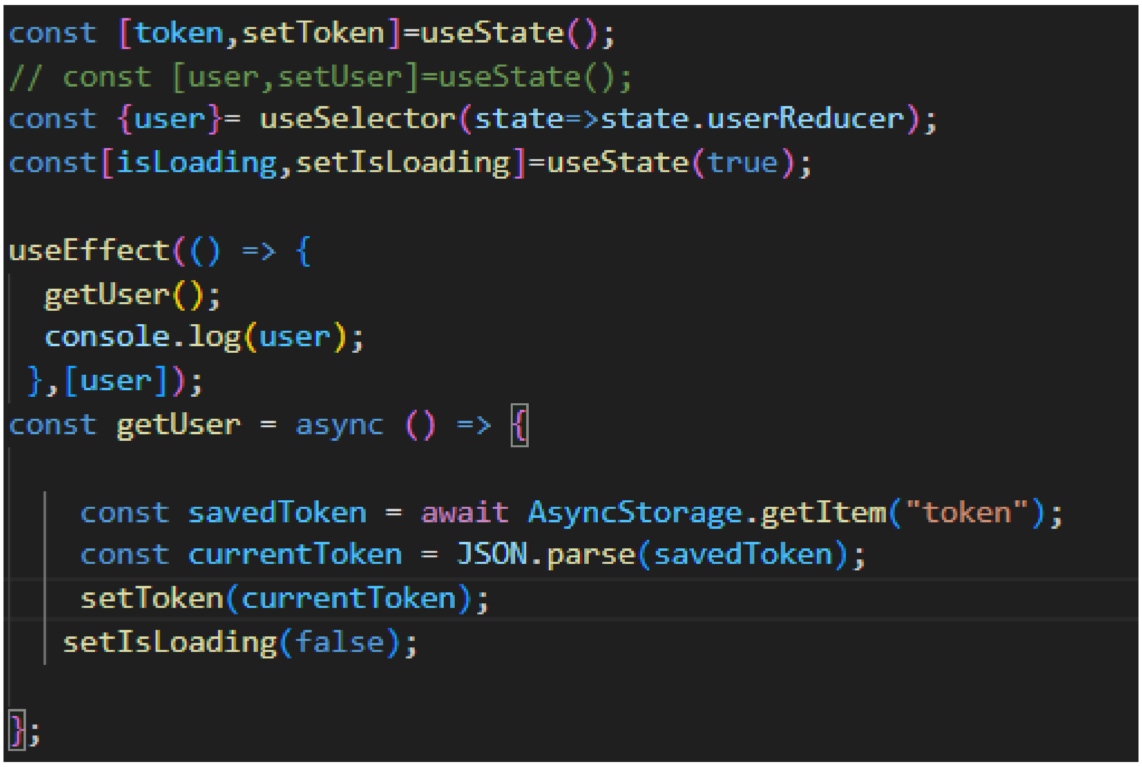
React Native follows the style of having the main App.js file be used for Navigation identification, and other files used for Screen Creation. A Screen is a JavaScript file that contains a function, also known as a functional component, which has various methods and states followed by a return statement which contains the content that will be displayed on the Screen such as a View Container, Text, etc. At the bottom of the Screen file a stylesheet component could optionally be placed to add some styling to the returned features in the retun statement. This way, React Native does the functions of HTML, JavaScript, and CSS all in the same file. All of that is optional as well, so like a web application, these various actions previously stated can be done in separate files and then called into one file by importing them.

2.2.3 React Native Life Cycle:

React Native follows the same Life Cycle as React DOM web applications, where React Native allows for the implementation of Hooks outside of Classes, Functional Components, as well as within classes ie. Class Components. A component Life Cycle in React terms is the different stages that a component goes through during it render. Hooks are functions or methods that can be called at different stages of the application. An example of which is in a React Class one could use the Hook `componentDidUpdate`, which updates the render of the UI when some props or states change to update the UI to display them. A prop or a state are variables which can be used to display or use conditionally to display other things onto the UI. There are also a multitude of other Hooks which can be used in a React Class Component [7]; However, React Native provides also provides Hooks which can be used in React Functional Components in order to actually react to those changes during the stages of the application in a Function and not in a Class, which are the `Effect`,`State`, `Context`, `Ref`, and `Performance Hooks` [1].

2.2.4 Effect and State Hooks:

The are also a few more hooks but in this Application only State, Ref, and Effect hooks are used. The Effect Hook performs the actions that the `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` Hooks performed in React Classes, which are reactions to when the component first renders, when it updates, and when it de-renders or dismounts. The Effect Hook is used to load in or call on certain functions at the very first render of the Screen by simply just calling the methods within it or performing an action within it such as setting a state, and unmounting is done placing the return statement within it followed by the action to be performed upon dismounting. The Effect Hook can also be called upon any state change by placing the state at the end of the `useEffect` method between square brackets. The State Hook is used to save certain states or values within the loaded screen for use throughout the Screen or even to pass as parameters or props to a child screen, the State Hook is referred to as `UseState` in code [8].

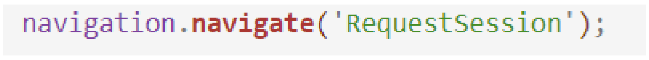


```
const [token,setToken]=useState();
// const [user,setUser]=useState();
const {user}= useSelector(state=>state.userReducer);
const[isLoading,setIsLoading]=useState(true);

useEffect(() => {
  getUser();
  console.log(user);
 },[user]);
const getUser = async () => {

    const savedToken = await AsyncStorage.getItem("token");
    const currentToken = JSON.parse(savedToken);
    setToken(currentToken);
   setIsLoading(false);

};
```

Figure 2.1: useEffect example on how states are loaded and set

2.2.5 React Native Navigation:

React Native provides different techniques for navigating through the Screens of an application, and these are known as Stack, Tab, and Draw bar Navigation. Stack Navigation is the type of navigation used to identify a tag for a specific screen which can be used to navigate to that screen using the navigation command. Tab Navigation is the type of navigation used to create a Tab Bar at the bottom of a screen and through that navigation could be done by pressing on a Tab Bar icon to navigate to a specific screen. Lastly Draw Bar Navigation creates a draw bar at the side of the screen which can be dragged into and out of view by swiping from the side of the screen, and this could also be used to navigate to screens by pressing on icons to navigate to those screens. Both Tab and Draw Bar navigation can also be navigated to through the navigation command.



```
navigation.navigate('RequestSession');
```

Figure 2.2: Example of the Navigation command to navigate to the Requested Sessions Screen

2.2.6 React Redux and AsyncStorage:

React Redux is a library defined by react which allows for data to be stored in a global state throughout the application. The way Redux works is by defining states and reducers, where a state is the item needed to store, and a reducer is the way to interact with the state. With this any changes to the states are visibly seen throughout the whole application. This is useful as core information which is required by multiple screens can be adapted to immediately with any change. AsyncStorage is a React community library, and it has a similar role as Redux except it is more useful for huge amounts of data that is does not change often. This can be used to store a currently logged in user's activity token for example. The main difference between them is that unlike Redux, changes to AsyncStorage are not made visibley seen immediately, as to detect change it would have to be retrieved first, while Redux changes are seen immediately. This makes Redux a more useful tool when it comes to updating screens through `useEffect` upon any changes to a state.

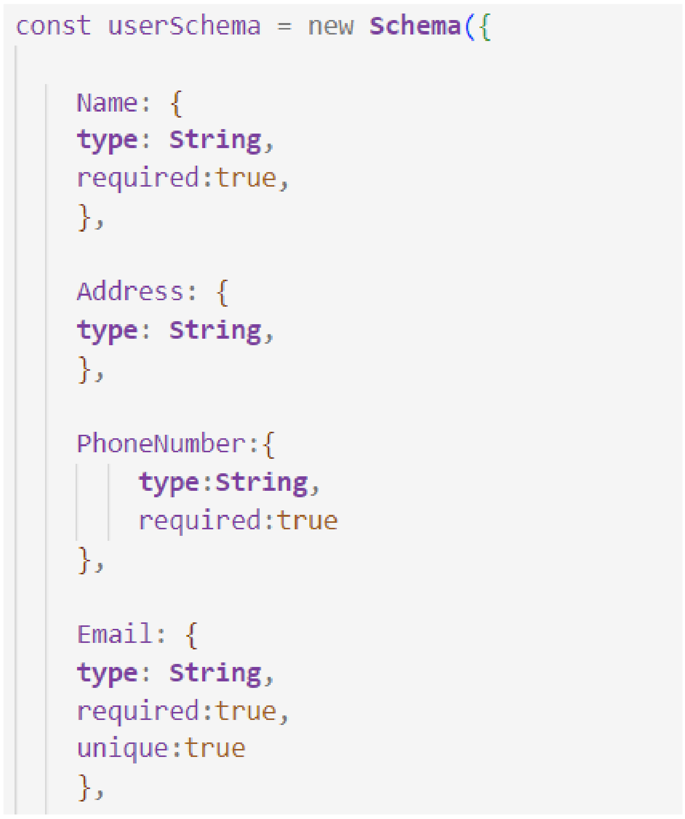
2.3 Backend:

2.3.1 NodeJs and Express JS:

Node JS and Express JS are the technologies used for the Backend of this project, where the are both technologies that use JavaScript to create locally hosted applications which can be used to run queries to databases or local queries that dont affect databases. Express JS is a Framework applied on Node JS, and it provides more features that NodeJS does not provide, such as controllers and routing for example. Throughout this project, Express JS is what is used to create all functionality of the controller classes, while NodeJS acts as the template to use Express JS.

2.3.2 Model Format:

In NodeJS files can be used to create JavaScript classes which can represent Models, with attributes representing them such as for example Name and Age with their own object types. These Model classes can be used to create a schema on a database using a code first approach, instead of defining the schema in SQL.

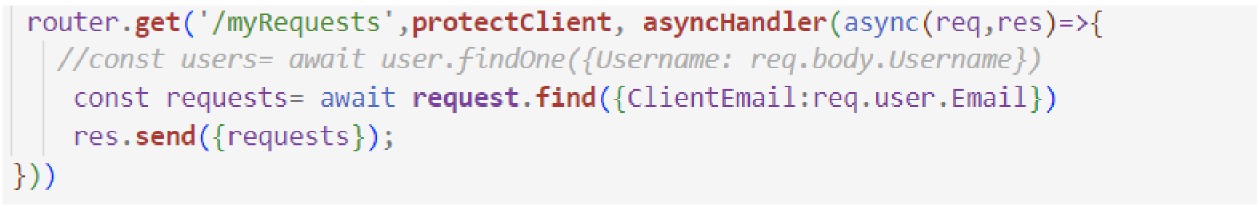


```
const userSchema = new Schema({

    Name: {
    type: String,
    required:true,
    },

    Address: {
    type: String,
    },

    PhoneNumber:{
        type:String,
        required:true
    },

    Email: {
    type: String,
    required:true,
    unique:true
    },
```

Figure 2.3: Backend Model for User Class

2.3.3 Controller Format:

Express JS allows for the creation of Controller classes. Controller classes are JavaScript classes that can be used to communicate with a database, and these are the classes that act as the API for the application. Express JS provides a restful way of communicating with the backend, as well as providing a mechanism to route, making the creation of an API simpler than classic NodeJS. NodeJS can be used for both React DOM and React Native, hence the same Backend could run on both a web application and a mobile application.



```
 router.get('/myRequests',protectClient, asyncHandler(async(req,res)=>{
   //const users= await user.findOne({Username: req.body.Username})
     const requests= await request.find({ClientEmail:req.user.Email})
     res.send({requests});
})) 
```

Figure 2.4: Express Request to Get Client's Requests

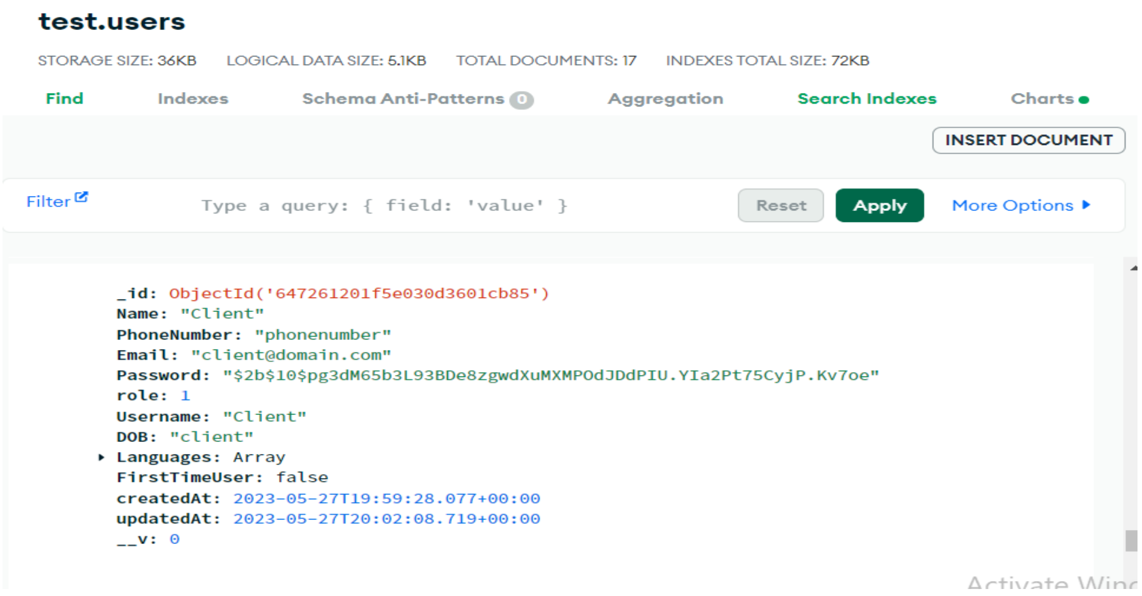
2.4 Database:

2.4.1 MongoDB

MongoDB is used for the database of this project, and it is the cloud storage that stores tables for any schema in a project. MongoDB is a NOSQL technology which allows for the creation of databases using minimalistic SQL commands. In this project, MongoDB stores the Models as the schema for the project and creates tables for them to store their information. The database is organized into clusters and those clusters are organized into collections. Collections are basically datasets which relate to a particular class in the project, so for example there could be one Users collection and one Daily Tasks collection. [4]

2.4.2 Mongoose:

Mongoose is a technology which connects the database on MongoDB to the Backened on the Node JS Application. It is used along with Nodemon to connect to the Node JS project. The way it works is upon creating models in our system such as for example a User class that has attributes such as Name and Email, once the program is compiled and run it would automatically be created in our MongoDB database with the database structure. This provides the method described above where now Model schema can be implemented code-first rather than by SQL or NOSQL.

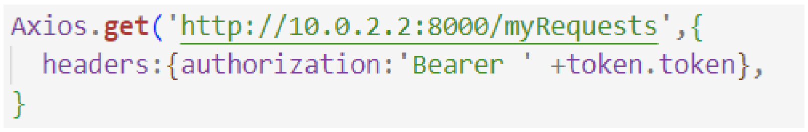

test.users
STORAGE SIZE: 36KB LOGICAL DATA SIZE: 5.1KB TOTAL DOCUMENTS: 17 INDEXES TOTAL SIZE: 72KB
Find
Indexes
Schema Anti-Patterns 0
Aggregation
Search Indexes
Charts
INSERT DOCUMENT
Filter
Type a query: { field: 'value' }
Reset
Apply
More Options

```
_id: ObjectId('647261201f5e030d3601cb85')
Name: "Client"
PhoneNumber: "phonenumber"
Email: "client@domain.com"
Password: "$2b$10$pg3dM65b3L93BDe8zgwdXuMXMPOdJDdPIU.YIa2Pt75CyjP.Kv7oe"
role: 1
Username: "Client"
DOB: "client"
▸ Languages: Array
FirstTimeUser: false
createdAt: 2023-05-27T19:59:28.077+00:00
updatedAt: 2023-05-27T20:02:08.719+00:00
__v: 0
```

Figure 2.5: MongoDB test collection, User schema

2.4.3 Axios:

The connection between the front end and the Back end is made using Axios, which is a feature that is used to send HTTP requests from the front end to the Back end [2]. Queries which reach MongoDB through Node JS are for example HTTP GET or HTTP POST requests. A GET request for example is a request made to retrieve information from the database, and POST requests are requests made to upload data to a specific collection on the database. Using Axios, a Frontend application can make HTTP request calls to the Backend, for which then the Backend would relay that HTTP request to the database and back.



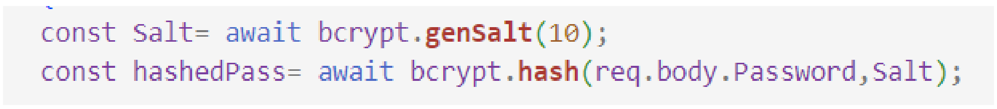
```
Axios.get('http://10.0.2.2:8000/myRequests',{
  headers:{authorization:'Bearer ' +token.token},
}
```

Figure 2.6: Axios GET Method

2.5 Encryption:

2.5.1 Bcrypt:

To protect passwords from being openly visible to the public eye, Bcrypt is used. Bcyrpt is a technology that encrypts information using something known as `Salt`, where a `Salt` is an autogenerated string for a given entry based on the length given to it. With this, a user's password can be encrypted to an unrecognizable string, and the only way to decrypt it is by running it through the same Bcrypt `Salt`. This is very crucial when it comes to storing passwords to databases, as most databases, if not IP restricted, can be accessed by anyone with the credentials. This way, even if the database gets unidentified access, the passwords are still not leaked.

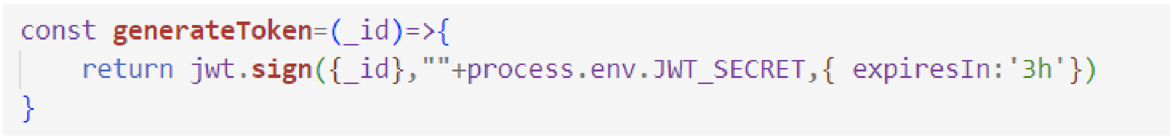


```
const Salt= await bcrypt.genSalt(10);
const hashedPass= await bcrypt.hash(req.body.Password,Salt);
```

Figure 2.7: Bcrypt Functionality to generate a Salt for Encryption

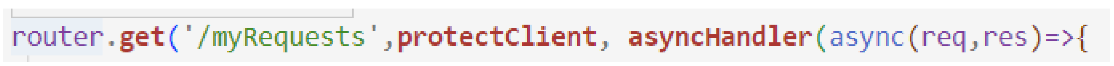
2.5.2 JWT:

To ensure that no access is allowed to users who do not belong to a certain type of user, for example a Client trying to access a feature that is restricted only to Admins, JWT Tokens are used. A JWT, JSON Web Token, Token is a string which defines specific attributes related to the logged in user. The way it works is similar to Bcrypt, where based on a string known as the JWTSECRET, a token encapsulating specified information is generated, where through decrypting the token, this information is made accessible. This is a very useful features for when users need to call on Backend methods which requires no Frontend input but only personal information, like getting their purchase history for example. With JWT, Protect statements can be made, where these Protect statements can put boundaries on what a user can and cannot access.



```
const generateToken=(_id)=>{
    return jwt.sign({_id},""+process.env.JWT_SECRET,{ expiresIn:'3h'})
}
```

Figure 2.8: Generation of a JWT Token



```
router.get('/myRequests',protectClient, asyncHandler(async(req,res)=>{
```

Figure 2.9: Example of Protect statement shown to be ProtectClient

2.6 Emulator:

To view the application while developing, Android Studio is used to launch an Android phone Emulator on it, which is basically a virtual Android phone which can be used to view our frontend design as well as test the functionality of the whole project. Through Android studio one can also test the application on a physical Android device by simply connected the device to the laptop with the application and scanning a QR code which opens the Application on the Expo Go app.

The Expo Go app is an application design by the developers of the Expo CLI which can be used to open application under development on physical Android or IOS devices and be able to use them physically. The Expo Go App is also installed on the Virtual device hence the application is modeled on it on the virtual device.

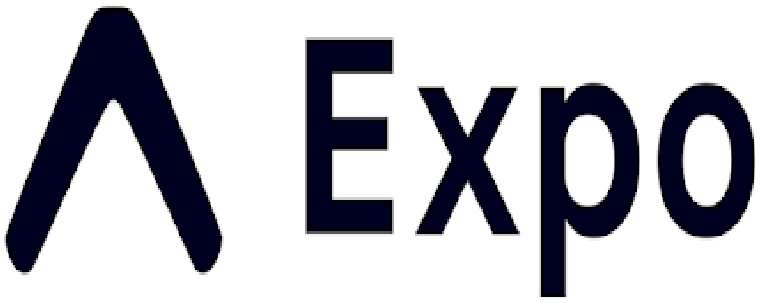


Figure 2.10: Expo Go app logo

2.7 Postman:

The Postman application is an application that allows for API Testing by a developer on any Backend system. The use of this is very crucial during testing phases for Backend systems to see if the requests made by them to their data collections are successful and correct, as well as see their effect on the data collections. The way Postman works is by selecting the type of Request to send, for example a GET request, then typing in the link to the API, the code that is written for the Backend which can be NodeJS for example and is defined by the index file with the port and link (localhost for example), which updates the data collections. Postman also allows parameters, body fields, or headers that the requests needs, and after clicking on send the request will either be successful and display the response in the console or it will be unsuccessful and display the error code in the console. [11]

2.8 Stripe:

The Stripe API is a transaction based API, which can be used in real life scenarios to actually handle business transactions. Stripe also allows a testing mode for developers to test the API in their applications with fake credit cards provided by Stripe. With Stripe, credit card transactions can be added to any application by using Stripe's API methods in the Backend and Stripe's payments sheets in the Frontend. This makes adding card payments to any application easy and secure, as the transactions are all handled by Stripe. Stripe not only provides payment handling, but also handles refunds and payment cancellations through payment intent keys. [13]

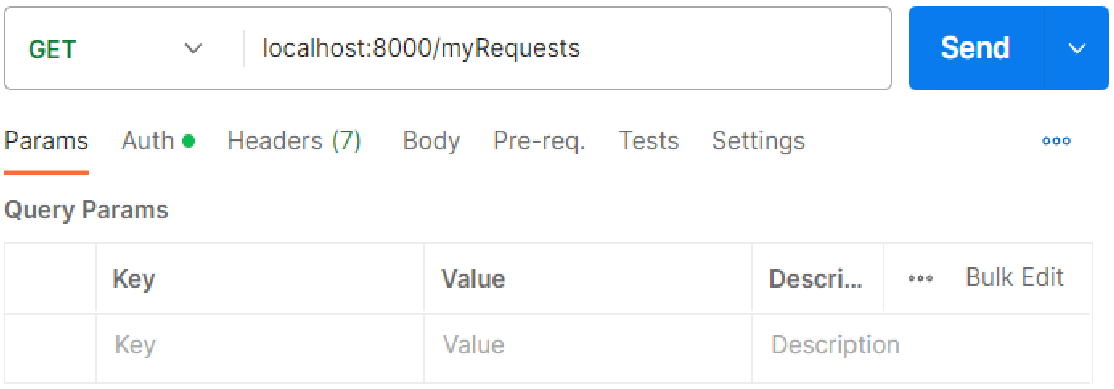


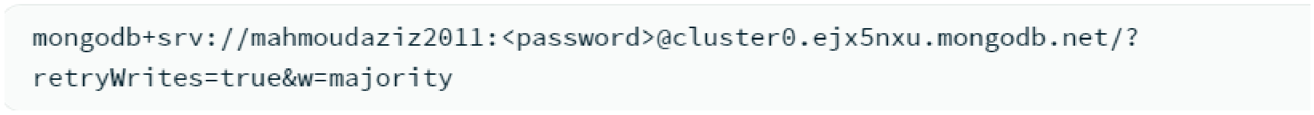
Figure 2.11: Postman Request Window

Chapter 3

Implementation

3.1 Database

The Database creation in this project is very simple, as the only requirement needed to create a database is to create a new project on `MongoDB`. After creating a new project, a new collection/database should then be created. This created collection acts as the main storage unit for the application, and the way that schema is mapped to it is through `Mongoose` as mentioned before. With this, collection creation with `MongoDB` is very simply, as only the `MongoDB` connection string is required in the Backend to map all the mongoose models to schema on `MongoDB`.



```
mongodb+srv://mahmoudaziz2011:<password>@cluster0.ejx5nxu.mongodb.net/?
retryWrites=true&w=majority
```

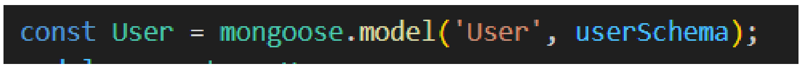
Figure 3.1: MongoDB connection string for connecting MongoDB with the Backend application to map Model Classes

3.2 Backend

The format of the Backend is folders of MiddleWare, Models, and Routes.

3.2.1 Models

The Models folder consists of the classes which are defined in the database. Where a class is basically the data structure which is sent to MongoDB to create a schema and collection for. This application's classes are `User, ApprovedSessions, RequestedSessions,` `PreviousSessions, Languages, and Reports` where for example the `User` class consists of attributes such as`Name, Username`, etc. which then upon running the application gets added to the database through `Mongoose` [9]. All `Models` discussed below could be referred to as `Mongoose` Models, as `Mongoose` is what allows for their table creation on `MongoDB`, hence they become local `MongoDB` accessible classes. Since all the `Models` are initialized as `Mongoose Models`, all routes made in the application are able to communicate directly with the database through them.

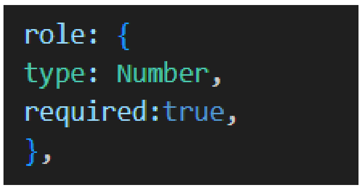


```
const User = mongoose.model('User', userSchema);
```

Figure 3.2: Mongoose Model Identification of the User Class

User Class

The `User` class relates to three types of users which are `Client, Admin, and Translator`, and allows all of them a set of attributes which are `Name, PhoneNumber, Email, Password, role, Username, DOB, Languages, FirstTimeUser, and WarnStatus`. The way the application has distinction between the types of users is through the `role` attribute in the `User` class. The `role` attribute is a number that is required upon user creation and is the core of the application as it is the key to loading certain properties to accomodate to the needs of the specific user. Role number one represents a client, role number two represents a translator, and role number three represents an admin. Not all of the attributes of the `User` class belong to all of the users in the system, as some are user specific. An Example of that could be the `WarnStatus` and `Languages` attributes which are only required by translators as they represent the warning level of a translator as well as the languages they administer.



```
role: {
type: Number,
required:true,
},
```

Figure 3.3: Role attribute in the User Class

Sessions Classes

The `RequestedSessions, ApprovedSessions, and PreviousSessions` classes are the classes related to the session information requested by clients and approved by translators. The sessions classes all have the same attributes which are `LanguageFrom, LanguageTo, Date, TimeFrom, Duration, ClientEmail, TranslatorEmail, ClientName, TranslatorName,` `CancelledBy, Type, UserPhoneNumber, TranslatorPhoneNumber, Payment, Amount, and pid`. However, the `PreviousSessions` class has one more attribute which is the `Reported` attribute which shows if the session had been reported yet or not. The purpose of the sessions classes is to show the different stages of a client's request. The request first starts as a `RequestedSessions` request where the client has yet to receive an approval by any of the translators available. Once approved/accepted by a translator, the request is then removed from `RequestedSessions` and added to `ApprovedSessions`. Once the request has been started it is then removed from `ApprovedSessions` and added to `PreviousSessions`. The `CancelledBy` array is the array that displays the emails of all translators that either declined or cancelled the session. It is later on used in loading the translators that could be assigned to the session, as then we will only load the translators that haven't cancelled or declined the session previously.

Languages Class

The `Languages` class is the class which represents all the languages in the system and only has one attribute which is `Language`. The `Languages` class serves as the data table which stores all the languages that are currently supported by our company, TranslateCo, Inc. Any languages that should be added to the system are always added to the `Languages` class, and from that class translators can add languages to their skillset as well. The `Languages` class's main use; however, is scalability, as storing these languages in Frontend states would require all of the states to be updated upon adding or removing languages. With the `Languages` class, an update to languages is made directly to the database and can be seen by anyone who should see it, without requiring changing anything in the Frontend.

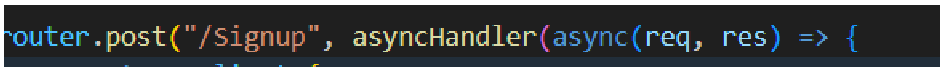
Reports Class

The `Reports` class adds a very useful feature to the application, which is the ability for a client to report a problem. User input is always crucial to any application or company in order to improve their service, and with the `Reports` class that is possible. The `Reports` class also allows for client reports to get feedback from admins, which is crucial for user experience as clients. The `Reports` class includes the attributes `SessionId, TranslatorEmail, TranslatorName, ClientName, TranslatorWarnStatus, ClientEmail, ReportDescription, ReportType, Resolved, ResolveAnswer`.

3.2.2 Routes

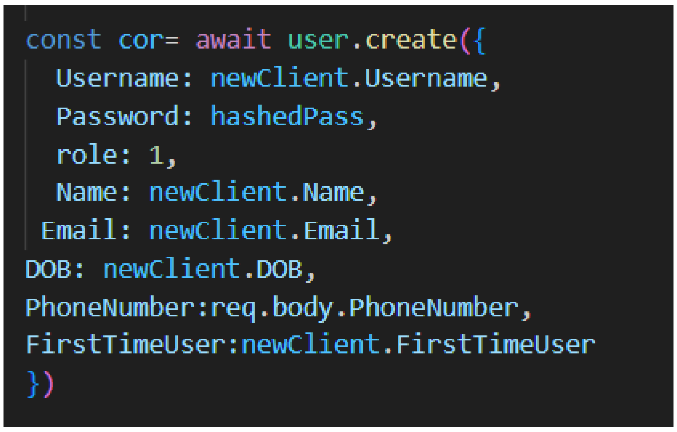
Routes Overview

The `Routes` folder contains the controllers for each type of user, where a controller is a class which allows for communication with the hosted Database. A controller class has multiple methods which make HTTP requests to the database to affect the data in it, and this is implemented with `Express JS`. For example the `Signup` feature is in the `authController` uses an HTTP Post request to the database to add a new user. These methods are written in `JavaScript` and are composed of `JavaScript` functionalities as well as mongoose methods which are used to communicate with and alter data in the database. To ensure authorized actions, where only specific users have ac-



```
outer.post("/Signup", asyncHandler(async(req, res) => {
```

Figure 3.4: The Signup method in the authController class



```
const cor= await user.create({
  Username: newClient.Username,
  Password: hashedPass,
  role: 1,
  Name: newClient.Name,
 Email: newClient.Email,
DOB: newClient.DOB,
PhoneNumber:req.body.PhoneNumber,
FirstTimeUser:newClient.FirstTimeUser
})
```

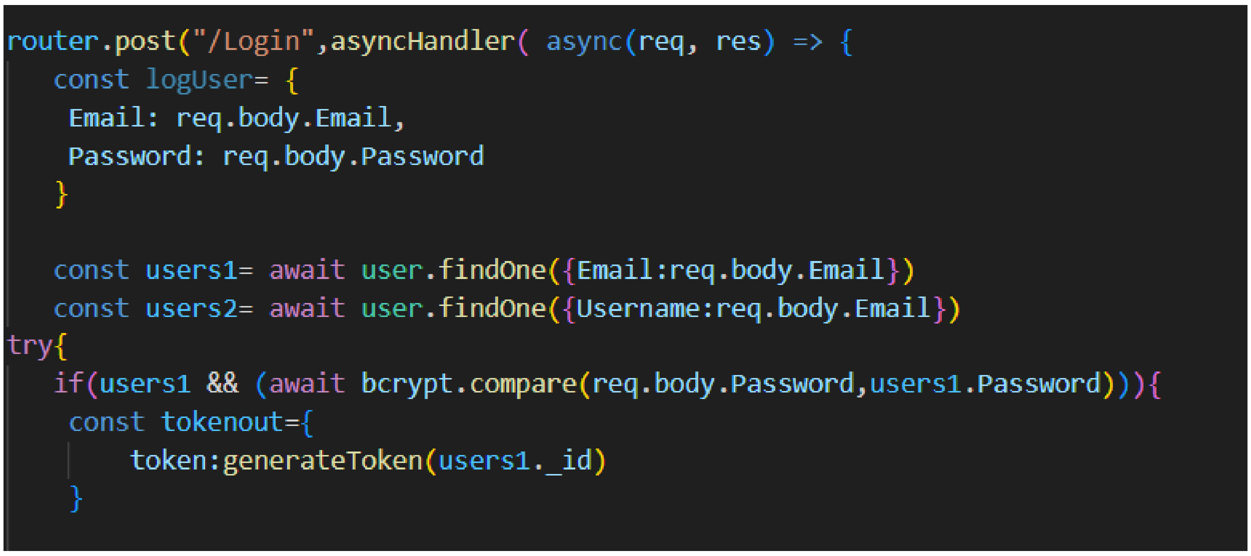
Figure 3.5: The create mongoose method applied on the user model, which is used to create a new user

cess to certain functions, we have five different controllers which are the `authController, adminController, clientController, translatorController, and paymentController`. Having seperate controllers for the different types of users allows for only users of specific roles to be able to perform the actions that only belong to their role. This is displayed through the calling on Backend features from the Frontend, as a Frontend that belongs to the a type of user has access to the methods only in its controller. This is also insured through the protection statements which are explained in the MiddleWare section below.

authController Class

The `authController` class contains all actions which are commonly shared by all types of users as well as those that are necessary for authentication, which includes login, verification, signup, password Changes, profile Updates, and first time user handling.
The `Login` function handles the issuance of a token, that is kept by the user, to ensure that this user has the specific access to a specific action. A token contains information such as the role and email of the user, which are necessary for other actions to be performed by the

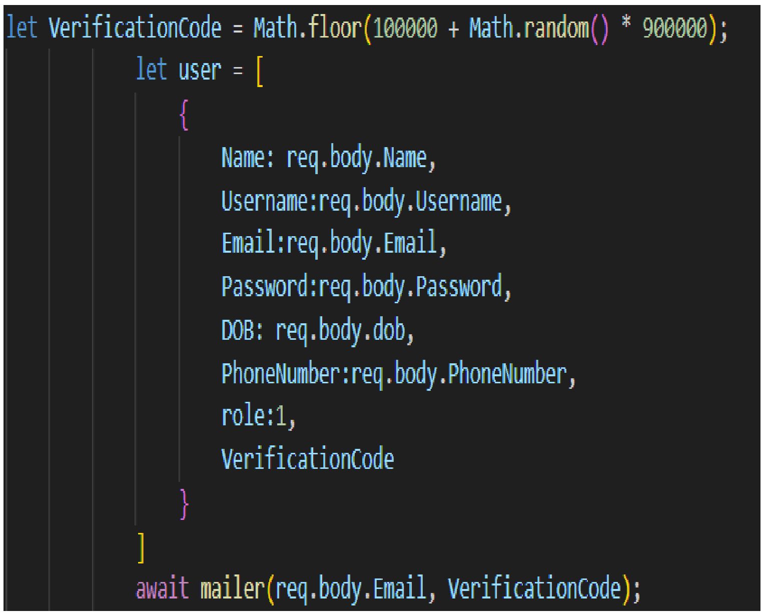
user and also to protect those actions from being accessed by unauthorized users.Token creation takes as input the user's `Email` and password then searches the database for the user with said `Email`, and if found it will decrypt the password and compare it with the entered password. If the passwords are identical then the token is generated for three hours and the user is logged in.



```
router.post("/Login",asyncHandler( async(req, res) => {
   const logUser= {
    Email: req.body.Email,
    Password: req.body.Password
   }

   const users1= await user.findOne({Email:req.body.Email})
   const users2= await user.findOne({Username:req.body.Email})
try{
   if(users1 && (await bcrypt.compare(req.body.Password,users1.Password))){
    const tokenout={
        token:generateToken(users1._id)
    }
```

Figure 3.6: The Login method in the authController class, with the token generation upon found email and matching passwords

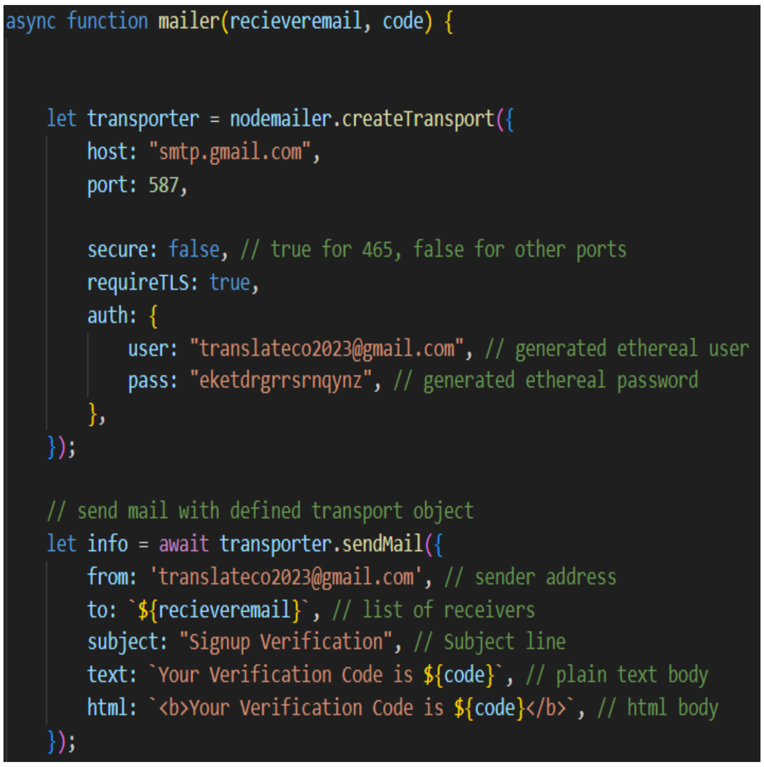
`Verification` is the step that is needed before signing up and upon requesting a password change due to forgetting the password. In the case of signing up, it is first performed after the client fills the signup form. How it work is that an email is sent to the email entered by the client with a verification code and the client is then prompted to enter that same code. The entered code is then compared with the required verification code and if they match then the user is allowed access/ signs up and is logged in.In the forgotten password case, it follows the same idea which is the user is sent an email with a verification code and is then prompted to enter that code to be able to change their password. The verification code is randomly generated in both Verification methods.



```
let VerificationCode = Math.floor(100000 + Math.random() * 900000);
            let user = [
                {
                    Name: req.body.Name,
                    Username:req.body.Username,
                    Email:req.body.Email,
                    Password:req.body.Password,
                    DOB: req.body.dob,
                    PhoneNumber:req.body.PhoneNumber,
                    role:1,
                    VerificationCode
                }
            ]
            await mailer(req.body.Email, VerificationCode);
```

Figure 3.7: The verification code generation and sending upon user signup

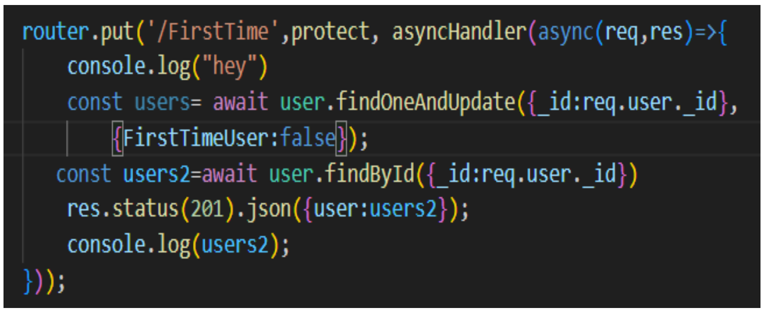
This email sending process is done through the `NodeMailer` library [12]. The `NodeMailer` library creates a transporter which then takes the sender email, the receiver email, an email subject and text, and an HTML component which is for designing the email to have a specific HTML template, otherwise it will be blank. In this application the mailer's HTML template is just a statement that says "Your Verification Code is" and is followed by the code that was generated. This mailer method as previously mentioned is called by the `verifyEmail and verify` methods, which are used for the case of forgotten password and signup respectively. The verification methods are very crucial for the security of the application, as they are used to ensure that the user trying to access the application is an actual Client and this is checked with the verification code sent in to their email. Having emails in the application as well provides a sense of responsiveness from the application to the user, which is crucial in any application for users not to feel the drag of application functionalities.



```
async function mailer(recieveremail, code) {


    let transporter = nodemailer.createTransport({
        host: "smtp.gmail.com",
        port: 587,

        secure: false, // true for 465, false for other ports
        requireTLS: true,
        auth: {
            user: "translateco2023@gmail.com", // generated ethereal user
            pass: "eketdrgrrsrnqynz", // generated ethereal password
        },
    });

    // send mail with defined transport object
    let info = await transporter.sendMail({
        from: 'translateco2023@gmail.com', // sender address
        to: `${recieveremail}`, // list of receivers
        subject: "Signup Verification", // Subject line
        text: `Your Verification Code is ${code}`, // plain text body
        html: `<b>Your Verification Code is ${code}</b>`, // html body
    });
```

Figure 3.8: The mailer function in the authController that is used to send emails upon requesting password change or signup

The `Signup` method is the feature used to create and add a new user into the database. The `Signup` method takes all the required attributes of the `User` class definition as parameters. The `Signup` method also encrypts the entered password using Bcrypt, and only after that can the user be added to the database. The `Signup` method is the method that is called after the verification process takes place, if the entered verification code matches.
The `updateProfile` method is used for updating the users key profile details, which are email, date of birth, name, and phone number. The `FirstTime` method is a method that turns the `FirstTimeUser` attribute for the user to false, and the affect of this is seen in the Frontend. The `FirstTimeUser` attribute is what is used by the Frontend to provide introduction slides to the application, where if it is true then the slides are displayed, but if it is false then they are not. Hence the `FirstTime` method is what allows the introduction slides to only be displayed upon first time login, as after first time login it is the `FirstTime` method is called and the `FirstTimeUser` attribute is set to false.



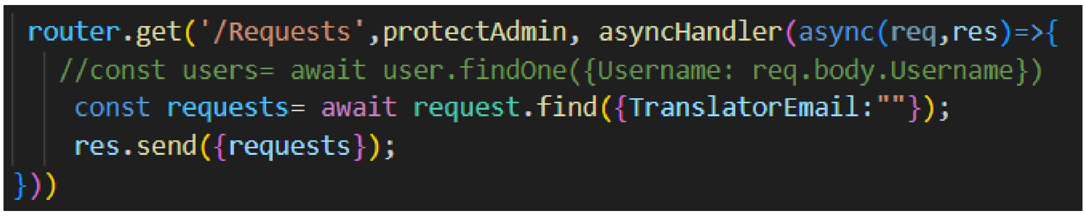
```
router.put('/FirstTime',protect, asyncHandler(async(req,res)=>{
    console.log("hey")
    const users= await user.findOneAndUpdate({_id:req.user._id},
        {FirstTimeUser:false});
   const users2=await user.findById({_id:req.user._id})
    res.status(201).json({user:users2});
    console.log(users2);
}));
```

Figure 3.9: The FirstTime method in the authController

adminController Class

The `adminController` class follows the same structure as the `authController`; however, the main thing that protects and distinguishes the `adminController` from other controllers is the imported `protectAdmin` method which is imported from the `Authentication` file in the MiddleWare folder. This is important as the `protectAdmin` method ensures the only a user with a token that is a valid and belonging to the role of an admin/ role number three can access the routes in the `adminController`. The `adminController` has many routes/methods which will now be discussed one by one.

First there are the request retrieval methods/routes, and these are the methods used by an admin to get all requests at their different stages. The `Requests` route is the route that retrieves all of the requests made by all clients in the system which have yet to be assigned to a translator.

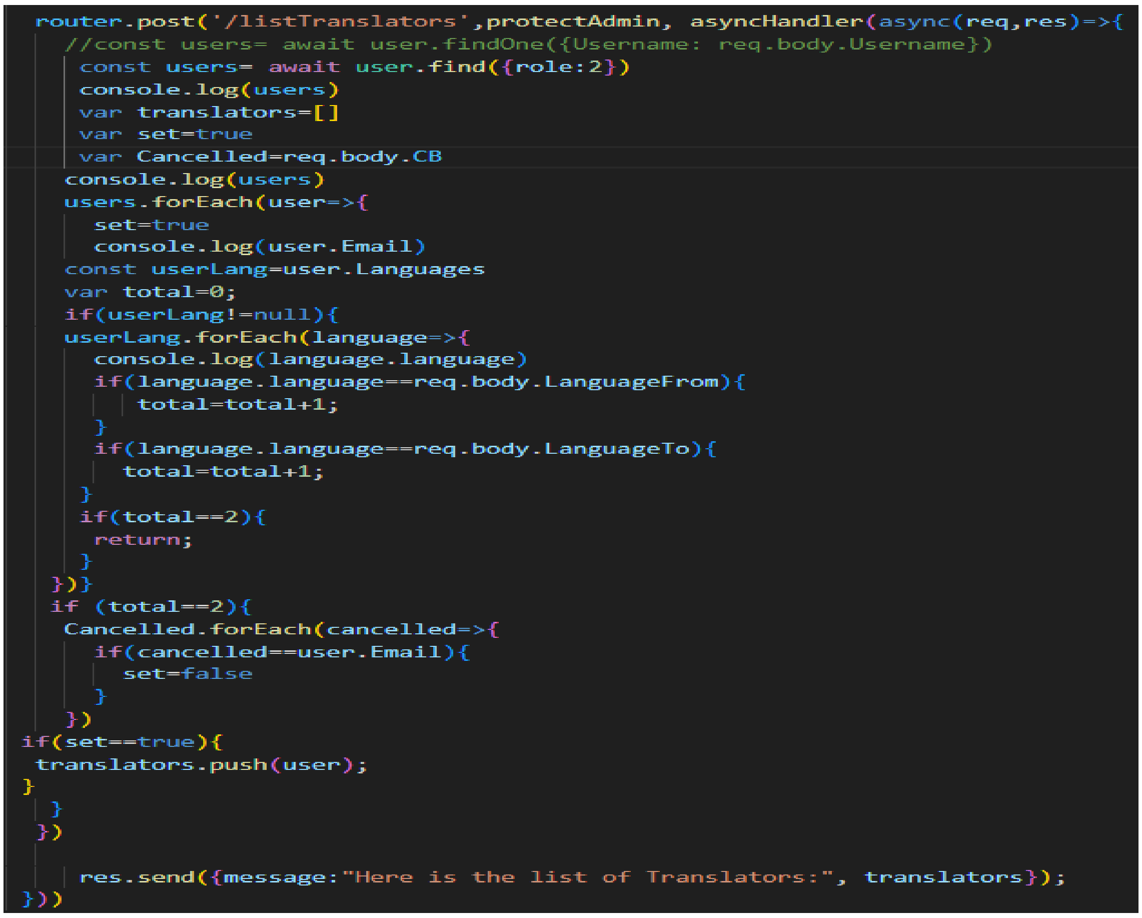


```
router.get('/Requests',protectAdmin, asyncHandler(async(req,res)=>{
  //const users= await user.findOne({Username: req.body.Username})
   const requests= await request.find({TranslatorEmail:""});
   res.send({requests});
}))
```

Figure 3.10: Requests Route in adminController class

The `AssignedRequests` route is the route that retrieves all of the requests made by all clients which have been assigned to a translator. The `ApprovedRequests` route is the route that retrieves all the approved requests that have been accepted by a translator and moved from being a `RequestedSession` to being an `ApprovedSession`. The `PreviousRequests` route is the route that retrieves all of the sessions that have been moved from `ApprovedSessions` to`PreviousSessions`, meaning that they have been completed already.

There are more data retrieval/fetching routes as well, and they are the `searchUser, getUsers, and listTranslators` routes. The `searchUser` route is the route that return a specific user upon searching for their username or email. The `getUsers` route is the route that gets all users belonging to a specific type of user, which is either client, translator, or admin. The `listTranslators` route is the route that returns all the translators that have not declined/cancelled a specific session as well as those who speak the languages of that specific session.

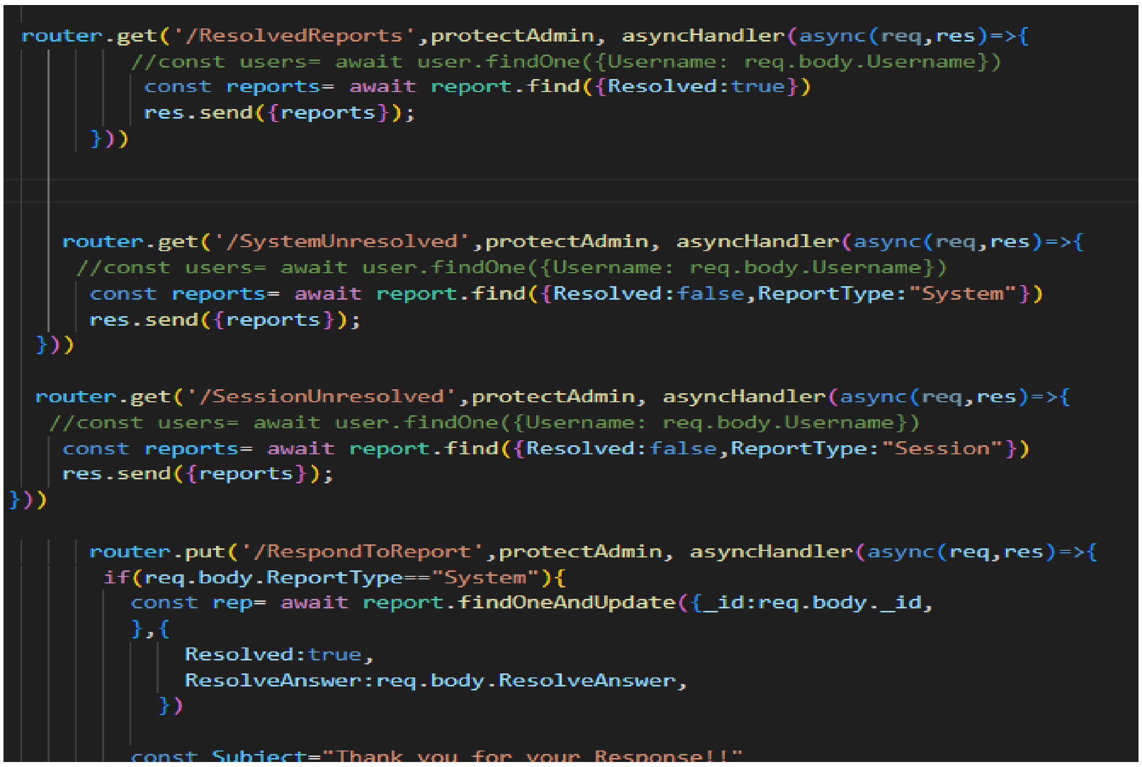


```
router.post('/listTranslators',protectAdmin, asyncHandler(async(req,res)=>{
  //const users= await user.findOne({Username: req.body.Username})
   const users= await user.find({role:2})
   console.log(users)
   var translators=[]
   var set=true
   var Cancelled=req.body.CB
  console.log(users)
  users.forEach(user=>{
    set=true
    console.log(user.Email)
  const userLang=user.Languages
  var total=0;
  if(userLang!=null){
  userLang.forEach(language=>{
    console.log(language.language)
    if(language.language==req.body.LanguageFrom){
       total=total+1;
    }
    if(language.language==req.body.LanguageTo){
      total=total+1;
   }
   if(total==2){
    return;
   }
 })}
 if (total==2){
  Cancelled.forEach(cancelled=>{
    if(cancelled==user.Email){
      set=false
    }
  })
if(set==true){
 translators.push(user);
}
 }
 })

   res.send({message:"Here is the list of Translators:", translators});
}))
```

Figure 3.11: listTranslators route in the adminController

The `adminController` has POST methods as well to both create new users and delete users, which are the `addUser` and `deleteUser` methods. The `addUser` method work as so to create a new user, and the way it work is by getting as input all the attributes requried for user creation and creates them. The `deleteUser` method does as the name would entail, where given a User id as input, the method would delete them from the system.

The `adminController` also includes methods to assign Translators to sessions, reassign a session to another translator, cancel a request, as well as view reports and take actions on resolving said reports. Upon cancelling a request, an email is sent to the client using `NodeMailer` as well as their payment is refunded if it was with card. The `Reassign` method removes the previous translator's email from the request and also adds it in the `CancelledBy` array belonging to the request.



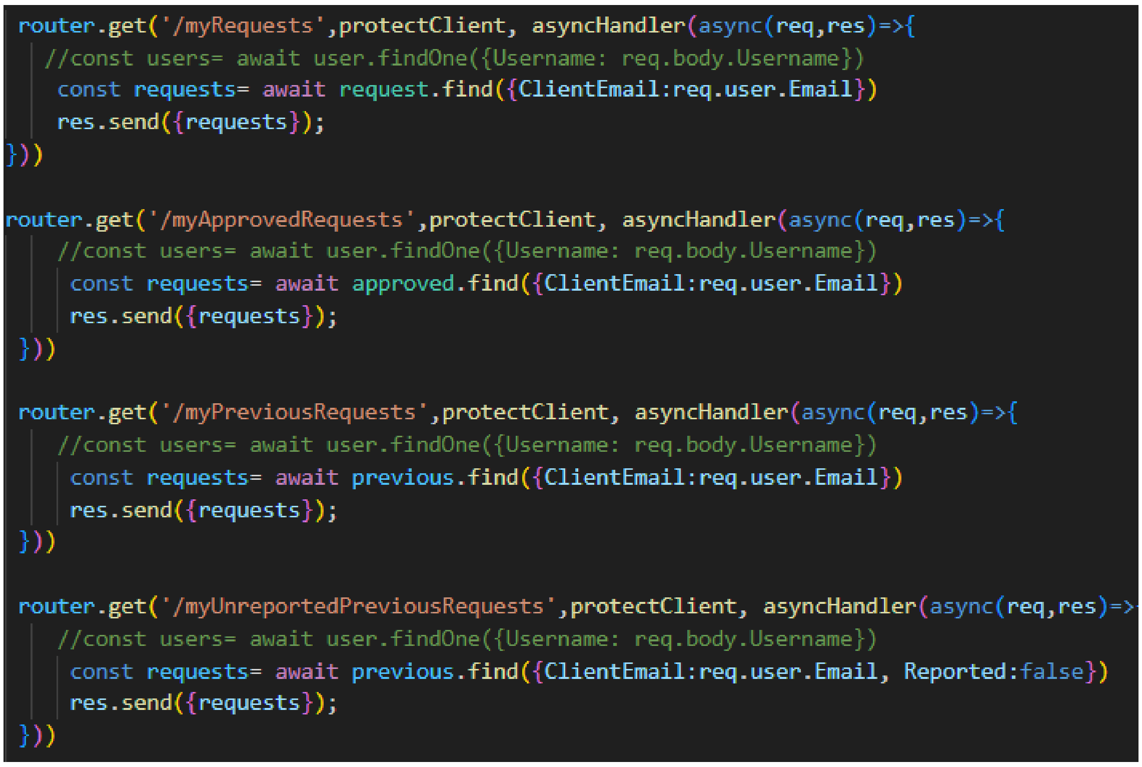
```
router.get('/ResolvedReports',protectAdmin, asyncHandler(async(req,res)=>{
        //const users= await user.findOne({Username: req.body.Username})
         const reports= await report.find({Resolved:true})
         res.send({reports});
    }))


   router.get('/SystemUnresolved',protectAdmin, asyncHandler(async(req,res)=>{
    //const users= await user.findOne({Username: req.body.Username})
     const reports= await report.find({Resolved:false,ReportType:"System"})
     res.send({reports});
  }))

  router.get('/SessionUnresolved',protectAdmin, asyncHandler(async(req,res)=>{
   //const users= await user.findOne({Username: req.body.Username})
    const reports= await report.find({Resolved:false,ReportType:"Session"})
    res.send({reports});
}))

    router.put('/RespondToReport',protectAdmin, asyncHandler(async(req,res)=>{
     if(req.body.ReportType=="System"){
       const rep= await report.findOneAndUpdate({_id:req.body._id,
       },{
           Resolved:true,
           ResolveAnswer:req.body.ResolveAnswer,
         })

       const Subject="Thank you for your Response!!"
```

Figure 3.12: The report retrieval and resolving routes in the adminController

clientController Class

The `clientController` file follows the same structure as the previous controller files, and what distinguishes its actions from the `adminController` is the `protectClient` function. This is important as the `protectClient` method ensures the only a user with a token that is a valid and belonging to a Client or role of one can access the routes in the `clientController`, otherwise an error statement would be returned saying "not Authorized". The clientController has many routes which will be discussed in detail in the following paragraphs.

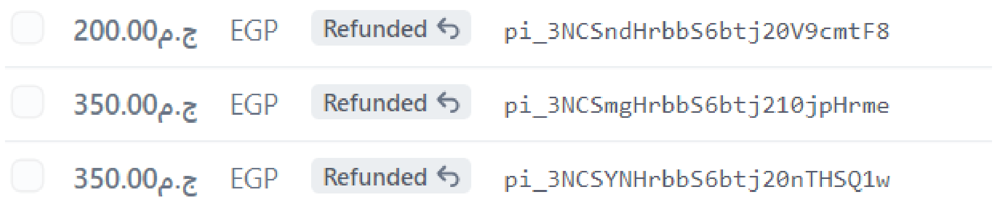
The `myRequests`, `myApprovedRequests`, `myPreviousRequests`, and `myUnreported-PreviousRequests` are request retrieval routes and are very similar to the request retrieval routes in the `adminController` class. The main difference between the GET request routes here and in the `adminController` is that here the `find() Mongoose` method takes in the `ClientEmail` as a parameter to retrieve only the requests that belong to this email. The only method that did not exist in the `adminController` is the `myUnreportedPreviousRequests` method, and what it does is retrieve the previous sessions that have yet to be reported.



```
router.get('/myRequests',protectClient, asyncHandler(async(req,res)=>{
   //const users= await user.findOne({Username: req.body.Username})
    const requests= await request.find({ClientEmail:req.user.Email})
    res.send({requests});
}))

router.get('/myApprovedRequests',protectClient, asyncHandler(async(req,res)=>{
    //const users= await user.findOne({Username: req.body.Username})
    const requests= await approved.find({ClientEmail:req.user.Email})
    res.send({requests});
 }))

router.get('/myPreviousRequests',protectClient, asyncHandler(async(req,res)=>{
    //const users= await user.findOne({Username: req.body.Username})
    const requests= await previous.find({ClientEmail:req.user.Email})
    res.send({requests});
 }))

router.get('/myUnreportedPreviousRequests',protectClient, asyncHandler(async(req,res)=>
    //const users= await user.findOne({Username: req.body.Username})
    const requests= await previous.find({ClientEmail:req.user.Email, Reported:false})
    res.send({requests});
 }))
```

Figure 3.13: The request retrieval routes in the clientController

The `cancelRequests`, `cancelApprovedRequests`, `requestSession`, and `AcceptStart-Session` routes are all POST request routes. The cancel routes are routes that are used to delete a type of session, where `cancelRequests, and cancelApprovedRequests` delete either a RequestedSession or an ApprovedSession respectively. Both of these cancel sessions also provide a full refund using the `Stripe` API refund feature if the payment was made using card. The way it works is that the refund method is given the payment intent from the requests `pid` attribute, and using that `pid` Stripe can find that payment and refund it.

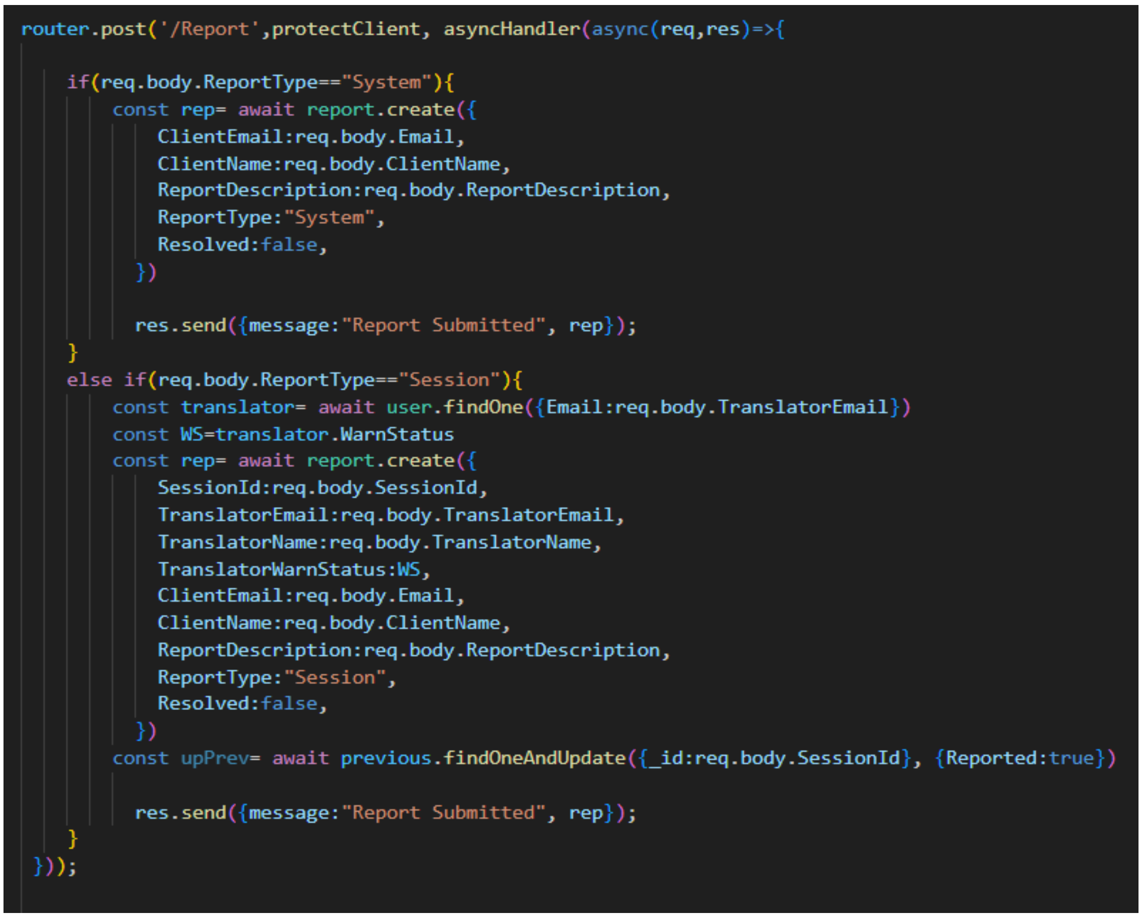


	200.00ج.م	EGP	Refunded ↩	pi_3NCSndHrbbS6btj20V9cmtF8
	350.00ج.م	EGP	Refunded ↩	pi_3NCSmgHrbbS6btj210jpHrme
	350.00ج.م	EGP	Refunded ↩	pi_3NCSYNHrbbS6btj20nTHSQ1w

Figure 3.14: Stripe Website refunded payments tab sample

The `requestSession` route is the route for which a client can request a new session. The `requestSession` uses the Mongoose method create which takes in the parameters needed to create a new `RequestedSession` and adds a new `RequestedSession` to the `RequestedSession` collection with those attributes. The `AcceptStartSession` route is the route that is used by a client to accept a request to start a session. This is only the case when the session is of type `InPerson`, and in the Frontend the option to access this method is only displayed on sessions that are of that type. In the case that the client does use the method, the method will remove the session from the `ApprovedSessions` table and will add it to the `PreviousSessions` table.

The `clientController` also has a set of routes to deal with reports. The `Report` route is the route that is used to create a new report for a session, and it creates the report based on the type of report the client makes. The report can either be of type session or system, and both of the have their own entry values used to create the report. Both types of reports belong to the same `Reports` class; however, as per the fact that some of the attributes in the `Reports` class are not required, not all attributes are set upon report creation depending on the report type. The `myResolvedReports and myNonResolvedReports` are the routes for retrieving all reports made by the client, belonging to that client's email, that have either been resolved or havent yet been resolved by an admin. These routes are useful for clients to keep track of their report's status, as later will be shown in the Frontend section.



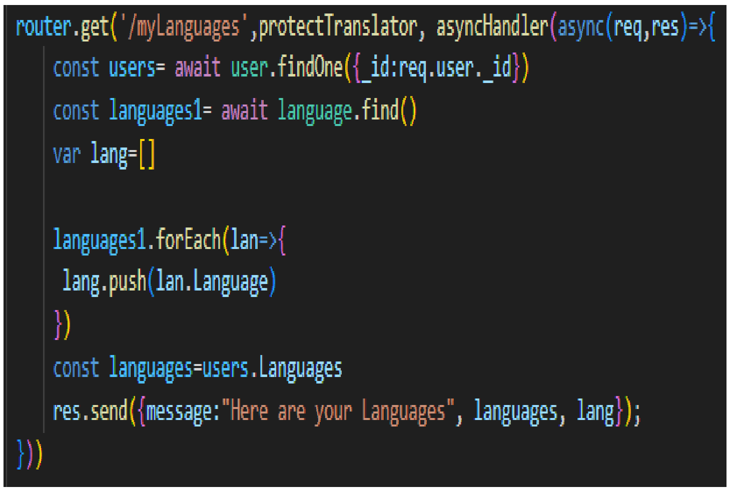
```
router.post('/Report',protectClient, asyncHandler(async(req,res)=>{

    if(req.body.ReportType=="System"){
        const rep= await report.create({
            ClientEmail:req.body.Email,
            ClientName:req.body.ClientName,
            ReportDescription:req.body.ReportDescription,
            ReportType:"System",
            Resolved:false,
          })

          res.send({message:"Report Submitted", rep});
    }
    else if(req.body.ReportType=="Session"){
        const translator= await user.findOne({Email:req.body.TranslatorEmail})
        const WS=translator.WarnStatus
        const rep= await report.create({
            SessionId:req.body.SessionId,
            TranslatorEmail:req.body.TranslatorEmail,
            TranslatorName:req.body.TranslatorName,
            TranslatorWarnStatus:WS,
            ClientEmail:req.body.Email,
            ClientName:req.body.ClientName,
            ReportDescription:req.body.ReportDescription,
            ReportType:"Session",
            Resolved:false,
          })
        const upPrev= await previous.findOneAndUpdate({_id:req.body.SessionId}, {Reported:true})

          res.send({message:"Report Submitted", rep});
    }
  }));
```

Figure 3.15: Report route in the clientController

translatorController Class

The `translatorController` class follows the same structure as the previous controller files, where the imports are all at the very top of the code and the route methods are below the imports. Like the previous two controllers, the `translatorController` is also distinguished by a protect method importend from the `MiddleWare`. This is important as the `protectTranslator` method ensures the only a user with a token that is a valid and belonging to the role of an Translator/role number two can access the routes in the `translatorController`, otherwise an error statement would be returned saying ”not Authorized”. The `translatorController` has many routes which will be explained in the paragraphs below.

The `translatorController`, like the previous two controllers, also has GET routes to retrieve the different sessions, and these routes are `myRequestsT, myApprovedRequestsT, and myPreviousRequestsT`. These routes are very similar to the equivalent ones in the `clientController`, as they also retrieve only the sessions belonging to a specific email. In this case; However, the email in question is not the client’s email but rather the translator's email. The `translatorController` also has other GET request routes which are the `myLanguages, and getLanguagesT` routes. The `myLanguages` route is the route that returns an array of all the languages that the translator that accessed the route speaks. Not only does it return the translator's languages, but it also returns an array of all the languages in the system. This was created specifically to cater to the Frontend screen that allows Translators to add languages to their profile and will be later explained in detail. The `getLanguagesT` route is the route that returns an array filled with all the languages in the system, and is used in the Frontend to fill language dropdown lists with the language names.

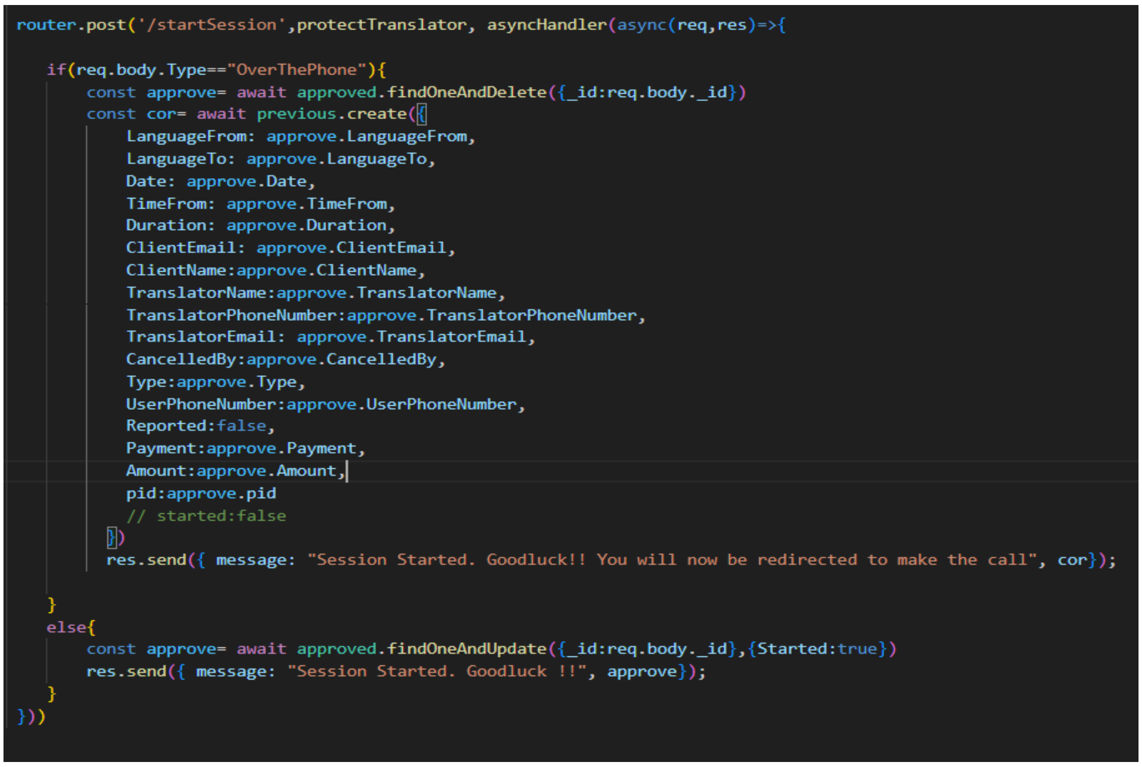


```
router.get('/myLanguages',protectTranslator, asyncHandler(async(req,res)=>{
    const users= await user.findOne({_id:req.user._id})
    const languages1= await language.find()
    var lang=[]

    languages1.forEach(lan=>{
     lang.push(lan.Language)
    })
    const languages=users.Languages
    res.send({message:"Here are your Languages", languages, lang});
}))
```

Figure 3.16: myLanguages route in the translatorController

Similar to the previous two controller classes, the `translatorController` also has cancellation routes. The cancellation routes in the `translatorController` are `decline-Request and cancelApprovedRequestsT`. The way `declineRequest` works is that it simply removes the translator email from the request in question. The way it does so is by having the request Mongoose id passed in as a body field to the route and using it to find and update the request to have an empty string as the translator email. The `cancelApprovedRequestsT` is not the same concept as the one from the `clientController` to cancel an approved session either. In the `translatorController` it does not erase the request completely from the system, but rather erases it from ApprovedSessions and adds it back to RequestedSessions. Not only does it do that but it also adds the translator's email to the request's `CancelledBy` attribute array field. In this case, related to the `adminController's listTranslators` method, if an admin were to try to assign a new translator from the translator list, the one who just cancelled wouldnt be in the list.

Finally, the `translatorController` has some translator specific routes as well, which are `acceptSession, addLanguages, deleteLanguage, and startSession`. The `accept-Session` route is one of the more important routes throughout the entire system, and it is the basis of the whole concept of reserving sessions. The `acceptSession` route is the route that allows a translator to approve of a session and add it to the ApprovedSessions with the session details as well as their own personal details. With that said, the session is also removed from RequestedSessions and an email confirmation is sent to the client's email informing them that their session has been approved.



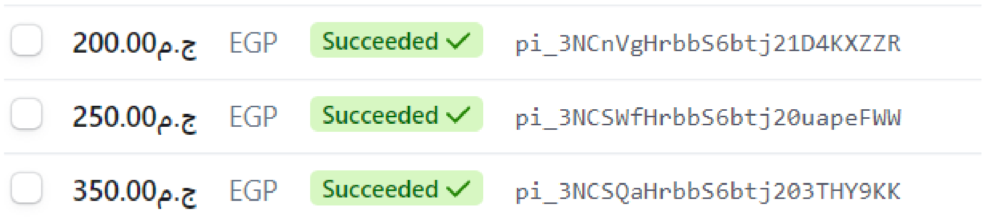
```
router.post('/startSession',protectTranslator, asyncHandler(async(req,res)=>{

    if(req.body.Type=="OverThePhone"){
        const approve= await approved.findOneAndDelete({_id:req.body._id})
        const cor= await previous.create({
            LanguageFrom: approve.LanguageFrom,
            LanguageTo: approve.LanguageTo,
            Date: approve.Date,
            TimeFrom: approve.TimeFrom,
            Duration: approve.Duration,
            ClientEmail: approve.ClientEmail,
            ClientName:approve.ClientName,
            TranslatorName:approve.TranslatorName,
            TranslatorPhoneNumber:approve.TranslatorPhoneNumber,
            TranslatorEmail: approve.TranslatorEmail,
            CancelledBy:approve.CancelledBy,
            Type:approve.Type,
            UserPhoneNumber:approve.UserPhoneNumber,
            Reported:false,
            Payment:approve.Payment,
            Amount:approve.Amount,
            pid:approve.pid
            // started:false
          })
          res.send({ message: "Session Started. Goodluck!! You will now be redirected to make the call", cor});

    }
    else{
        const approve= await approved.findOneAndUpdate({_id:req.body._id},{Started:true})
        res.send({ message: "Session Started. Goodluck !!", approve});
    }
}))
```

Figure 3.17: startSession route in the translatorController

The startSession route is the route that allows a translator to start a session. By starting a session, we mean deleting it from ApprovedSessions and creating it in PreviousSessions. There is; however, two different outcomes of this route, as if the session was of type OverThePhone then the process previously mentioned would occur. However, if the session was of type InPerson, then the route would on turn the attribute Started to true. This in turn in the Frontend allows the client to see this change and by able to use the AcceptStartSession route in the clientController to remove it from ApprovedSessions and add it to PreviousSessions. Lastly, the addLanguages and deleteLanguage routes are used to add a language to the translator's languages array in their User object, as well as delete a language from that same array.

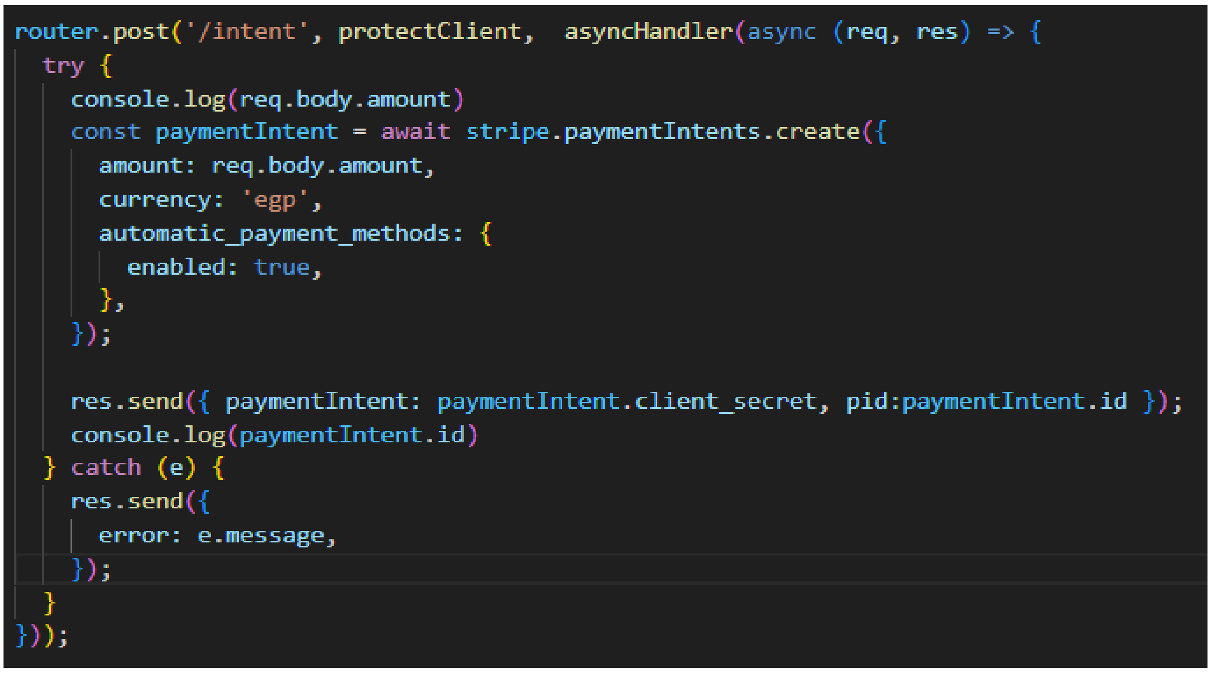
paymentController Class

The paymentController class is the class in charge of creating Stripe payment intents for all sessions that are paid for using card. The paymentController class is quite simple in terms of amount of routes in it compared to the previous controllers, as it only has one route. The route is called intent, and as the name suggests, it role is to create a payment intent using an amount which is passed to it as an input body field from the Frontend. A payment intent can also be thought of as a payment request, where its a request to withdraw a certain amount of money from a card. The request remain inactive or pending until the `Stripe` payment sheet is filled in the Frontend. If the card has enough funds the transaction will be successful, and if not it will be failed or uncaptured. Lastly, the intent route is client protected, meaning that only clients can have access to this route, which is the logical choice here since only clients make payments in this application.



☐	200.00ج.م	EGP	Succeeded ✓	pi_3NCnVgHrbbS6btj21D4KXZZR
☐	250.00ج.م	EGP	Succeeded ✓	pi_3NCSWfHrbbS6btj20uapeFWW
☐	350.00ج.م	EGP	Succeeded ✓	pi_3NCSQaHrbbS6btj203THY9KK

Figure 3.18: Stripe Website successful payments tab sample

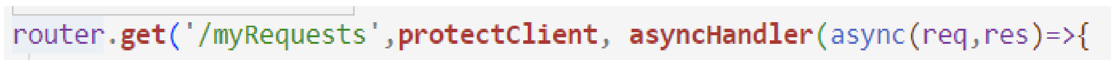


```
router.post('/intent', protectClient,  asyncHandler(async (req, res) => {
  try {
    console.log(req.body.amount)
    const paymentIntent = await stripe.paymentIntents.create({
      amount: req.body.amount,
      currency: 'egp',
      automatic_payment_methods: {
        enabled: true,
      },
    });

    res.send({ paymentIntent: paymentIntent.client_secret, pid:paymentIntent.id });
    console.log(paymentIntent.id)
  } catch (e) {
    res.send({
      error: e.message,
    });
  }
}));
```

Figure 3.19: Stripe payment intent route in the paymentController

3.2.3 MiddleWare

The `MiddleWare` Folder contains the Authentication file which ensures the authenticity of a signed in user by checking tokens to restrict certain actions. The checking on tokens is done through methods called `Protect`, where a `Protect` method checks on the Authorization Bearer token which is sent along as a headers parameter to the method/route that is requested. If the Token contains the role that matches with the role belonging to the method in question then it is allowed, otherwise the action is denied to the user. This is done by importing the `Protect` methods from the `Authentication` file into a controller as previously shown. An example of this could be the `myRequests` route in the `clientController`, where only the client should have access to the method that returns their own requests. Therefore, the `Protect` statements are the safety nets that ensure no user can access any feature that is not authorized for them. The way tokens are used is also very crucial to how the entire application is run, as all screens in the Frontend that make Backend requests must also send the token as a header. Not only that, but tokens ensure that only logged in users can use any of the applications features, as without a token, the user would not be able to get past the login screen as a safety measure, as only logged in users get a token generated for them.



```
router.get('/myRequests',protectClient, asyncHandler(async(req,res)=>{
```

Figure 3.20: Example of Protect statement shown to be ProtectClient

3.2.4 Miscellaneous

This section contains all the files which do not belong to any specific folder, such as the`env, db, index`, and `package` files. The `env` file contains the database connections URL, the port to locally deploy the application, as well as the `JSON Web Token Secret` which is used by the `generateTokens` method in the `authController`, and it basically is the password which allows the method to actually generate a token. The `db` file contains the JavaScript code which allows for the connection to the MongoDB database through the `Mongoose` connect method. The `index.js` file is the main run file of the Backend of the project and it contains the code required to be able to use the `Models`, controllers, database (by importing db file), specific technologies, and Port listening. Finally the `package.json` file contains information relating to the versions of the technologies used as well as licensing and application version information, and the `package-lock.json` file contains the same information with the addition of the information in the `node modules` folder, which it can be helpful for testing.

3.3 Frontend

The Frontend format is split into four folders which are assets, common, redux, and components.

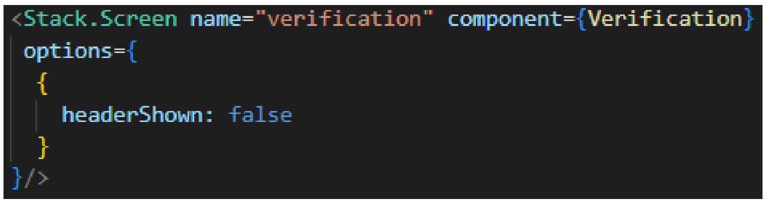
3.3.1 Redux

The Redux folder is the folder that represents the state handling features of the application. The Redux folder contains three files which are the `Store, Reducers, and Actions` files. The `Actions` file is the file that establishes the actions that could be performed on states. In our application, the only action used is the `setUser` action which enables a dispatch action to be performed on an attribute called user. A dispatch action can be thought of as a setting action, which allows us to set an item with a specific value. The `Reducer` file is what establishes the states that are to be made globally available, and within it as well creates a function called `userReducer`. The `userReducer` function is the function that actually sets the state to the value that is applied to the `setUser` method. Finally, the `Store` file is the file that is used to create a root reducer for all of the reducers in a system. Redux states are retrieved in any screen using the `useSelector` hook on the `userReducer`, and are set in any screen by calling the `useDispatch` hook and passing to it the `setUser` method with the value to be set. An example of this can be seen in the Login screen as it sets the user state to contain the attributes of the logged in user.

3.3.2 Components

The Components folder is located within the src folder, and it includes two more folders which are screens and navigators. The screens folder contains JavaScript files which display the React Native Frontend design of multiple different views which represent different features (for example the home page has `Home.js`). Each action has its own JavaScript file, so to assign a translator for example, the view that represents that option is `AssignTranslator`. The format of the React Native JavaScript files is functional components, which have the imports at the top, main function in the middle, and stylesheet at the bottom. By main function, what is meant is the function that is exported as default from that file. The main function can contain within it other functions, constants, and the main return statement which is the display of that screen. Below the main function, there is the stylesheet component, which serves a way to create style for returned element from within the functional component itself and not have it be imported.

Within the Components folder there is also another folder titled navigators, and this folder is what contains all the navigator components in our application. A navigator component is a component that establishes a route for a screen component, and route in this case is like linking or screen connection route unlike the Backend route. What these route can then be used for is to connect different screen together through certain actions, for example going from the home screen to the screen for requesting a new session. What a navigator does to establish this route is to simply establish a name for a screen within navigator borders, then that name can be used to navigate to that screen from another screen. The core navigator class for this application is the startup file which is the `App.js` file; however, this class is not included in the navigators folder.



```
<Stack.Screen name="verification" component={Verification}
 options={
  {
    headerShown: false
  }
}/>
```

Figure 3.21: The Stack screen created to identify a route for the Verification screen in the App.js file

In the following sections, the flow of the entire application will be discussed as startup screens, navigators, navigator screens, shared screens, client screens, translator screens, and admin screens.

3.3.3 Startup Screens

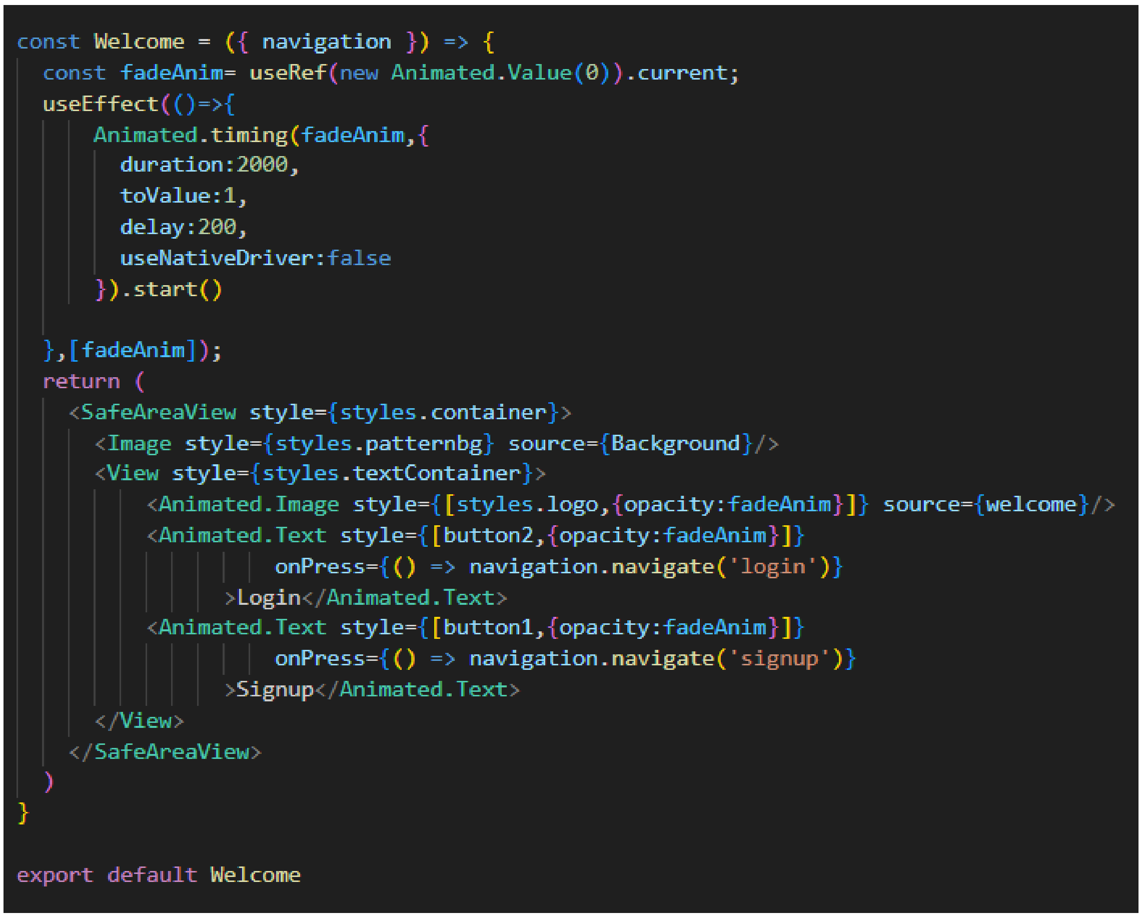
Startup Overview

The Startup screens are the screens display upon opening the application, and they are almost exactly the same for all users, with only one screen being different. The user first starts of on the Welcome, which can then take them to either `Login` or `Signup` screens. Following the Login screen there are the `VerifyEmail` screen, `ForgotPassword` screen, the `ChangeForgottenPassword` screen, the `Home` screen, and the `IntroScreens` screen. These are the screens used for verifying the email for password change, requesting the change, changing the password, user's home, and introduction to application. Following the `Signup` screen there is another screen which is the `Verification` screen, the screen used to verify a user upon signing up. After the user logs in, They are then taken to the Home screen, which is different depending on the user's role.

It is important to note that all Backend method calls are done through `Axios` calls from locally defined function with the screens. For list style screens, the `useEffect` method does not directly call the Backend method, but rather calls a function which makes an `Axios` call to the Backend methods

Welcome Screen

The `Welcome` screen is the very first screen that is displayed upon starting up the application. The `Welcome` screen contains two buttons for `Login` and `Signup` as well as the welcome logo for the application. The `Login` button navigates the user to the `Login` Screen and the `Signup` button navigates the user to the `Signup` screen. Both `Login` and `Signup` screens' navigation identification are included in a stack navigator in the `App.js` file, and can be navigated to using the navigation command discussed earlier. The `Welcome` screen also has an `Animated` feature, where the logo and buttons fade into vision. This `Animated` feature is done by using the `Animated` element and applying it on the Image and Button elements. The constant that get animated is the `fadeAnim` constant which is a `useRef` hook function. The way `fadeAnim` works is the it is initialized with an animated value of zero, and is then passed to the `useEffect` function and put into motion through the timing function displayed below, which moves it to be value one over a two hundred milliseconds delay.



```
const Welcome = ({ navigation }) => {
  const fadeAnim= useRef(new Animated.Value(0)).current;
  useEffect(()=>{
      Animated.timing(fadeAnim,{
        duration:2000,
        toValue:1,
        delay:200,
        useNativeDriver:false
      }).start()

  },[fadeAnim]);
  return (
    <SafeAreaView style={styles.container}>
      <Image style={styles.patternbg} source={Background}/>
      <View style={styles.textContainer}>
          <Animated.Image style={[styles.logo,{opacity:fadeAnim}]} source={welcome}/>
          <Animated.Text style={[button2,{opacity:fadeAnim}]}
                    onPress={() => navigation.navigate('login')}
                >Login</Animated.Text>
          <Animated.Text style={[button1,{opacity:fadeAnim}]}
                    onPress={() => navigation.navigate('signup')}
                >Signup</Animated.Text>
      </View>
    </SafeAreaView>
  )
}

export default Welcome
```

Figure 3.22: The code displaying the animated feature and the return function of the Welcome screen

Login Screen

The `Login` screen is a very important screen when it comes to the functionality of the whole system. What the `Login` screen does is that it contains two text fields for Email or Username and for Password, and upon pressing the `Login` button it calls on the Backend route/method of the same name. As previously discussed, the Backend `Login` method is the screen that checks for the validity of the user and if they are valid. If the user is valid, then a token is generated for that user and is returned to the Frontend along with the user's information. Then as a user logs in to the application, their data and token are also retrieved and stored in their screens. The way that the token and user details are stored is through two ways, the token is stored in `AsyncStorage`, and the user details are stored in Redux. Both the token and the user details are retrieved from the response to the `Axios` request that calls on the `Login` Backend method. With this, both token and user details are made globally available within the applications local context, meaning that it is available to every screen in the local run of the application.

The `Login` screen also contains other features within the card input form, which gets almost all of its styles from the button and form class which contain styles for forms in general. The card features the display of an error message, defined by a state called `errorMsg`, at the top of the screen depending on the response to the `Axios` call to the `Login` Backend method. If the user is not found, the error message will be update to say invalid credentials, and before the `Axios` call, if the user was to forget one of the two fields it would be set to All fields are required. Upon pressing the text fields after an error, the error message would be set to null again to clear it. The password text box has secure entry enabled, hence once the user enters a key it is turned into a bullet point. Above the input form View there is also the company welcome logo as well as the trademark for the company, and clicking on them would redirect/navigate the user back to the Welcome screen.

The `Login` screen finally also has two more ways to navigate out of itself, which are to the `ForgotPassword` screen, or to the `Signup` screen. For `ForgotPassword` and `Signup`, it is done through two text links which specify either "Forgot your Password?" or "Don't have an account? Create an account". After the user Logs in, they are either taken to their home screens or to a role based application introduction slides screen. The navigation in this case is based on whether the user is a first time user or not. This is defined by the `FirstTimeUser` attribute, which was explained in the Backend `Models` section. The `FirstTimeUser` attribute is retrieved from the user details return from the Login Bacekend to the `sendtoBackend` method that contains the `Axios` request to the Backend. If the `FirstTimeUser` attribute is false, then the user is navigated to the `Home` screen, and if it is true then they are navigated to the `IntroScreens` screen.

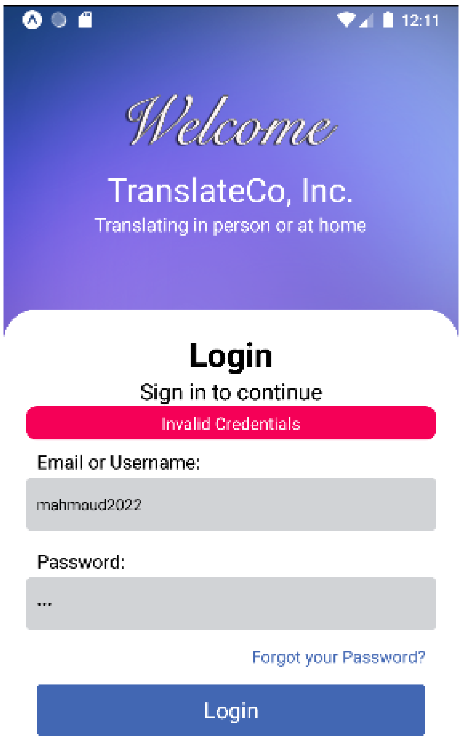
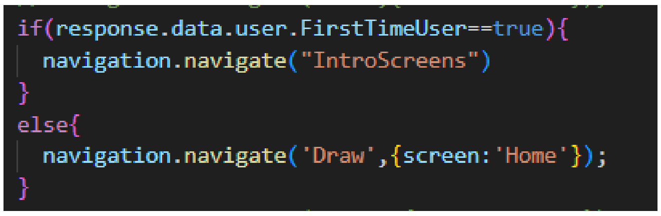
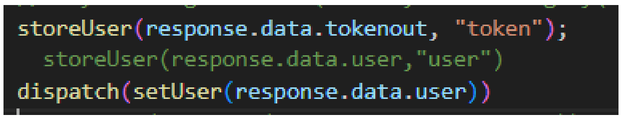


Figure 3.23: The Emulator display of the Login Screen upon incorrect input credentials



```
if(response.data.user.FirstTimeUser==true){
  navigation.navigate("IntroScreens")
}
else{
  navigation.navigate('Draw',{screen:'Home'});
}
```

Figure 3.24: The code displaying the navigation statements from Login to either IntroScreens or Home



```
storeUser(response.data.tokenout, "token");
  storeUser(response.data.user,"user")
dispatch(setUser(response.data.user))
```

Figure 3.25: The code displaying storing the token in AsyncStorage using the method SetUser, and storing the user details in Redux using dispatch

Signup Screen

The `Signup` Screen is the second option which can be accessed from the Welcome Screen. The `Signup` Screen features a main View which contains six text boxes as well as a button. The text boxes are the ones for entering the new Client'`sEmail, Username, Name, Password, and Password Confirmation`, and the button is for Signing up. Among the fields there is a also a button to enter the user's date of birth, and upon clicking it a calendar picker is opened for the use to pick their birthday. The calender picker is created using the `datetimepicker` community library [10]. Above the main view at the top of the screen is also the company logo and trademark, and clicking on them would also redirect the user back to the `Welcome` Screen. Within the view there is also a text link to redirect the user to the `Login` Screen in case the user already has an account with the title "Already have an account?". At the very top of the `Signup` Screen code there is a state called clientData which stores all of the input values put in to the text fields. There is also the same state in the `Login` screen for the error message which can be set using `setErrormsg`, and it is visible in the return at the very top of the main View.

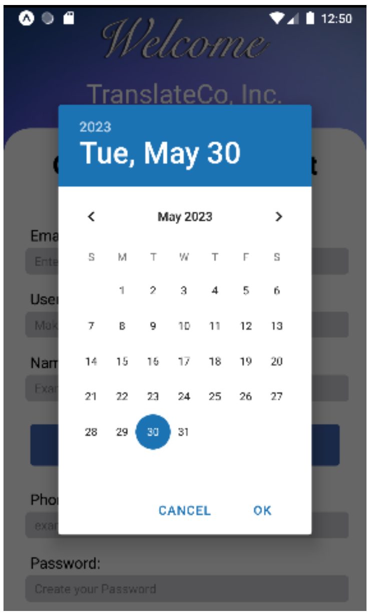
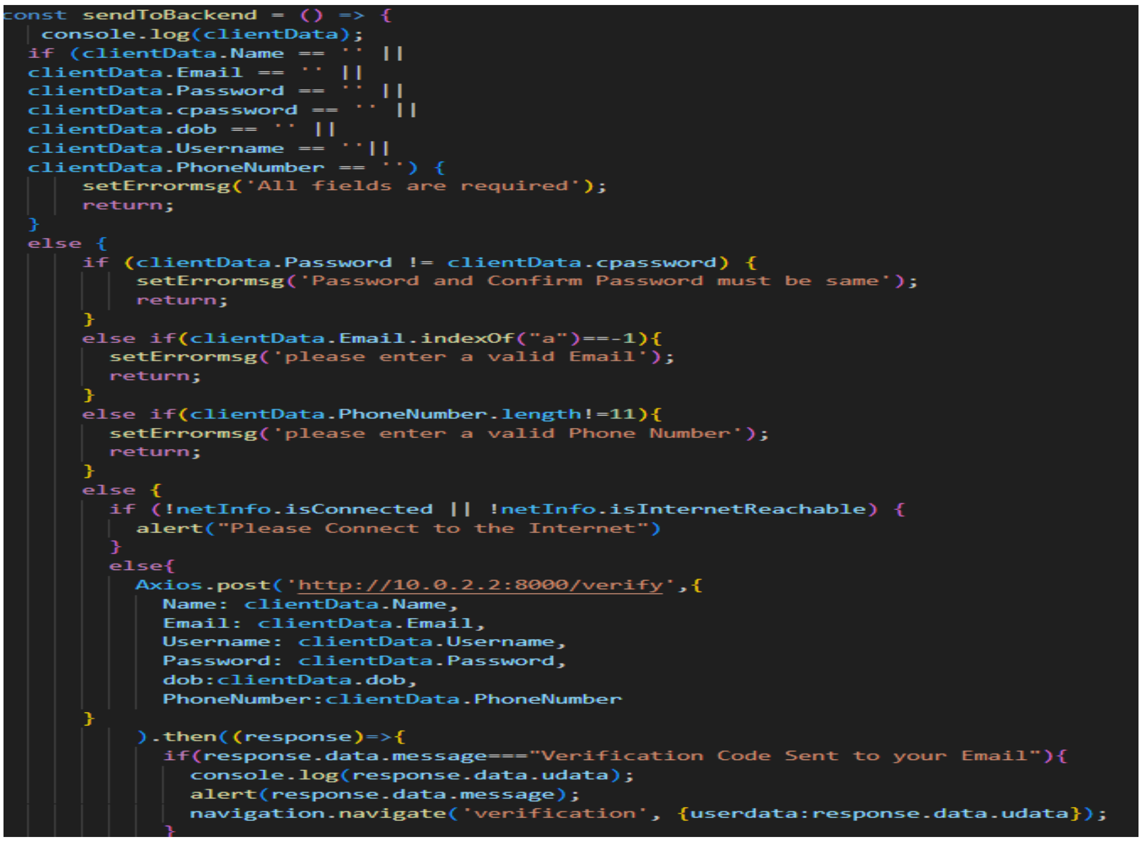


Figure 3.26: Signup calendar picker displayed on Expo Go App

Once the user clicks on the `Signup` button, the request is then sent to the `sendToBackend` method. The `sendToBackend` method first checks if all the fields have been filled by checking if any of them are empty strings. If they are then it sets an error message at the top of the main View saying 'All fields are required' using `setErrormsg('All fields are required')`. If they are not all empty strings then the `sendToBackend` method checks if both `Password` and `Confirm password` are the same, and if they are not then the error message will be set as "Password and Confirm Password must be same". If the passwords are the same then the `sendToBackend` method checks if the email entered is in email format or not using the method `indexOf` as `clientData.Email.indexOf("a")`. If it is equal to negative one, then that means that the @ separator is not included in the Email state, hence the error message is set to 'please enter a valid Email'. If it is in email format then the `sendToBackend` method finally sends the request to the `verify` Backend method, with the `clientData` state information excluding confirm password as the body field.

If the response is error that means that this email has already been registered for an account. If response is not an error that means that a verification Email has been sent to the entered email with the verification code retrieved from `userdata` return from the Backend method. The Backend method also returns a response message saying "Verification Code Sent to your Email". Once that is done, the client is navigated to the `Verification` screen where they are prompted to enter the verification code.



```
const sendToBackend = () => {
   console.log(clientData);
  if (clientData.Name == '' ||
  clientData.Email == '' ||
  clientData.Password == '' ||
  clientData.cpassword == '' ||
  clientData.dob == '' ||
  clientData.Username == ''||
  clientData.PhoneNumber == '') {
      setErrormsg('All fields are required');
      return;
  }
  else {
      if (clientData.Password != clientData.cpassword) {
          setErrormsg('Password and Confirm Password must be same');
          return;
      }
      else if(clientData.Email.indexOf("a")==-1){
        setErrormsg('please enter a valid Email');
        return;
      }
      else if(clientData.PhoneNumber.length!=11){
        setErrormsg('please enter a valid Phone Number');
        return;
      }
      else {
        if (!netInfo.isConnected || !netInfo.isInternetReachable) {
          alert("Please Connect to the Internet")
        }
        else{
          Axios.post('http://10.0.2.2:8000/verify',{
            Name: clientData.Name,
            Email: clientData.Email,
            Username: clientData.Username,
            Password: clientData.Password,
            dob:clientData.dob,
            PhoneNumber:clientData.PhoneNumber
      }
          ).then((response)=>{
            if(response.data.message==="Verification Code Sent to your Email"){
              console.log(response.data.udata);
              alert(response.data.message);
              navigation.navigate('verification', {userdata:response.data.udata});
            }
```

Figure 3.27: The sendToBackend method which calls upon the verify Backend method

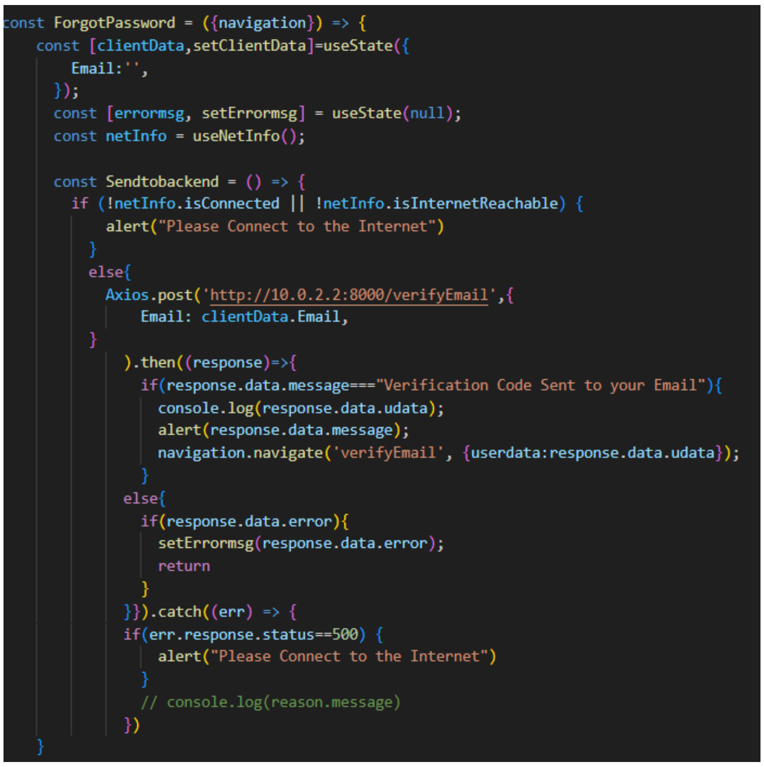
Verification Screen

The `Verification` Screen is the screen that the user is redirected to after the `Signup` Screen. The `Verification` Screen contains the same top trademark and company logo as the previous screens as well as a main view which contains a text box and a button, and well as the title "A Code has been sent to you on your email". The text box has the placeholder "Enter 6 digit Verification Code" as well as also having `secureTextEntry` so that the entered numbers are turned into bullet points for security reasons. The Screen contains three states which are the error message, the `userCode`, and the `actualCode`. The `userCode` is the code that the user enters into the text box, and the `actualCode` is the verification code that was sent as a parameter from the `Signup` Screen as part of the `userdata` parameter. The `actualCode` is set as that value by the `useEffect` method which runs when the page first renders.

The user obtains the code to enter through their mailbox for the email they entered when signing up. The user at that point should have received an email from TranslateCo, with the 6 digit verification code. Once the user clicks verify, the `sendToBackend` method is called and it first checks if the `userCode` is empty or still set as its default value. If it is then the error message is set to 'Please enter the code'. If the `userCode` is not either of those options, then the `sendToBackend` method compares the `userCode` with the `actualCode`, and if they are not the same then the error message is set to 'Incorrect code'. Otherwise, a new constant is prepared with the user data that was entered in the `Signup` screen and is sent to the `Signup` Backend. After that the user receives an alert saying "account created succesfully" and is navigated back to the Login Screen.

ForgotPassword Screen

The `ForgotPassword` Screen is the screen which the user is redirected to when they click on "Forgot your Password?" on the Login Screen. The screen contains the same header with the company logo and trademark as the previous screens, as well as the same `errormessage` state.The screens main View contains a text input for the user's email, which is stored in the `clientData` state, as well as a button to send the verification code to that email. Once the user clicks on the button the `Sendtobackend` method is called. The `Sendtobackend` method then uses Axios to sends email attribute from the `clientData` state to the `verifyEmail` Backend method. If the Email is not registered for an account in the system, then the Backend will return the error message 'This email not registered for an account'. If the Email was found in the system, then the Backend will return the message "Verification Code Sent to your Email" as well as send a generated verication code and user email back as a response. It will as well email the verification code to user's email. The screen then displays the response message as an alert and redirects the user to the `VerifyEmail` Screen with the parameter `userdata` containing the returned response data.

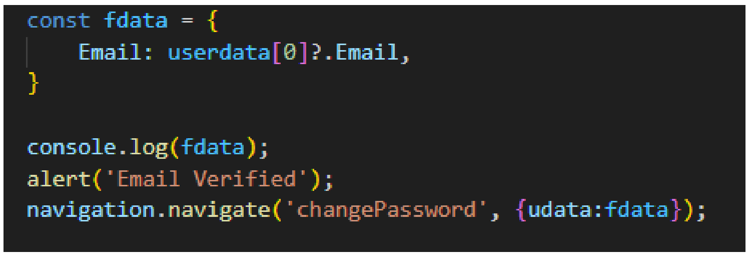


```
const ForgotPassword = ({navigation}) => {
    const [clientData,setClientData]=useState({
        Email:'',
      });
      const [errormsg, setErrormsg] = useState(null);
      const netInfo = useNetInfo();

      const Sendtobackend = () => {
        if (!netInfo.isConnected || !netInfo.isInternetReachable) {
            alert("Please Connect to the Internet")
          }
          else{
            Axios.post('http://10.0.2.2:8000/verifyEmail',{
                Email: clientData.Email,
          }
              ).then((response)=>{
                if(response.data.message==="Verification Code Sent to your Email"){
                  console.log(response.data.udata);
                  alert(response.data.message);
                  navigation.navigate('verifyEmail', {userdata:response.data.udata});
                }
              else{
                if(response.data.error){
                  setErrormsg(response.data.error);
                  return
                }
              }}).catch((err) => {
              if(err.response.status==500) {
                  alert("Please Connect to the Internet")
                }
                // console.log(reason.message)
              })
    }
```

Figure 3.28: The functional part of the ForgotPassword screen with the states as well as the sendToBackend method

VerifyEmail Screen

The `VerifyEmail` Screen is the screen which the user is navigated to after the `ForgotPassword` Screen is successful, and it is exactly the same format as the Verification Screen. The `actualCode` state is then set to the verification code from the `userdata` parameter in the `useEffect` method. Once the user enters the verification code and it is matches the `actualCode`, the Sendtobackend method in the `VerifyEmail` Screen then alerts to the user that the Email is verified and redirects the user to the `ChangeForgottenPassword` Screen. The users email is also passed to it as a parameter called `udata`.



```
const fdata = {
    Email: userdata[0]?.Email,
}

console.log(fdata);
alert('Email Verified');
navigation.navigate('changePassword', {udata:fdata});
```

Figure 3.29: The VerifyEmail's code to pass the email and navigate to ChangeForgot-Password

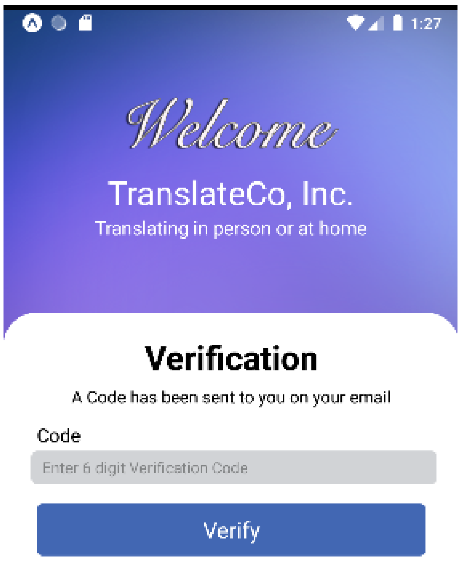

Welcome
TranslateCo, Inc.
Translating in person or at home
Verification
A Code has been sent to you on your email
Code
Enter 6 digit Verification Code
Verify

Figure 3.30: The VerifyEmail screen

ChangeForgottenPassword Screen

The `ChangeForgottenPassword` screen is the page that follows the `VerifyEmail` screen. The screen format has the company logo and trademark at the top of the screen like the previous screens. The screen's main view contains two text boxes for password and confirm password as well as a button to change the password. With that said, the screen also contains three states which are error message, email, and password. The email state is set using `useEffect` with the Email parameter sent from the `VerifyEmail` screen. The password state contains two attributes which are `Password` and `CPassword` which are updated whenever the user types in the password or confirm password text boxes respectively. The password text boxes both have secure text entry. Once the user presses on the `Change Password` button, the `sendToBackend` method is called and first checks if either of the Password or CPassword attributes of the password states are empty strings, and if they are then the error message is set to 'All fields are required'. If they are not empty strings but are not equal to each other then the error message is set to 'Password and Confirm Password must be same'. Finally if they are the same then the `sendToBackend` method calls on the `changepass` Backend PUT method with the values of `Email` and `Password` as parameters. The Backend will then update the password belonging to that email to be the new one and the screen will then alert 'Password Changed Successfully' before being navigated back to the Screen.

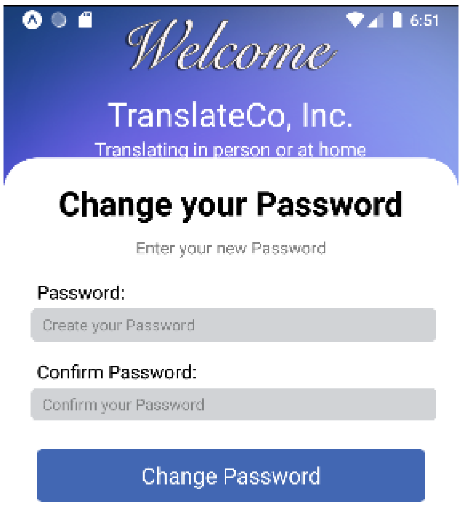


Figure 3.31: The ChangeForgottenPassword screen

IntroScreens Screen

The `IntroScreens` screen is the screen that the user navigates to upon their first time logging in. The `IntroScreens` screen contains introductory slides displaying how to use the application. The `IntroScreens` screen is role specific, and depending on the role, different introduction slides are shown. The introduction slides are creates as constants and are passed to the data prop of the rendered `AppIntroSlider`. The `AppIntroSlider` is the library that allows for the creation of introduction slides. The `IntroScreens` screen is navigated to from the Login screen if the `FirstTimeUser` attribute of the user is true. Upon pressing the done button on the introduction slider, the `FirstTimeUser` attribute for the user will be updated to false, using the Backend method `FirstTime`, and navigate the user to the Home screen. This in turn will make the user not be navigated to the `IntroScreens` screen upon their next login and will be navigated directly to the Home screen.

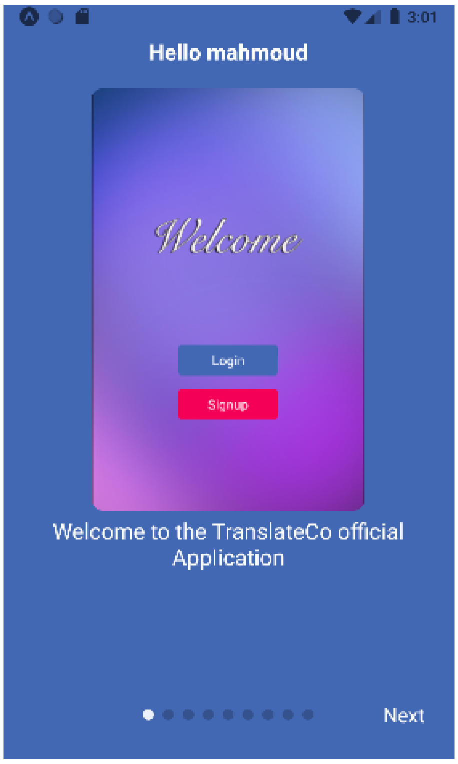


Figure 3.32: The first introductory screen for a new client

3.3.4 Navigators

For all screens that require user information such as role, token, or other details from the User class, all of them are retrieved from the user Redux state and through `useEffect` retrieving the token.

Navbar Navigator

The `Navbar` navigator is the navigation file that is responsible for the bottom tab bar visible in the Home and Settings screens. The `Navbar` navigator is not a role based navigator, as all the screens in the navigator are shared amongst all types of users. The `Navbar` navigator has navigation for both the `Home` and `Settings` screens hence they are only accessible through it. The way it is created is by creating a bottom tab navigator and creating within it tab screens with both the Home screen and the Settings screen stacks. The default screen on the `Navbar` navigator is the `Home` screen hence the first page that it loads on when routed to will be the `Home` screen. The `Navbar` is also decorated with `IonIcons`, which are icon shapes that are provided by the `react-native-vector-icons` library. A bottom tab navigator is created through the statement `createBottomTabNavigator()` [5].

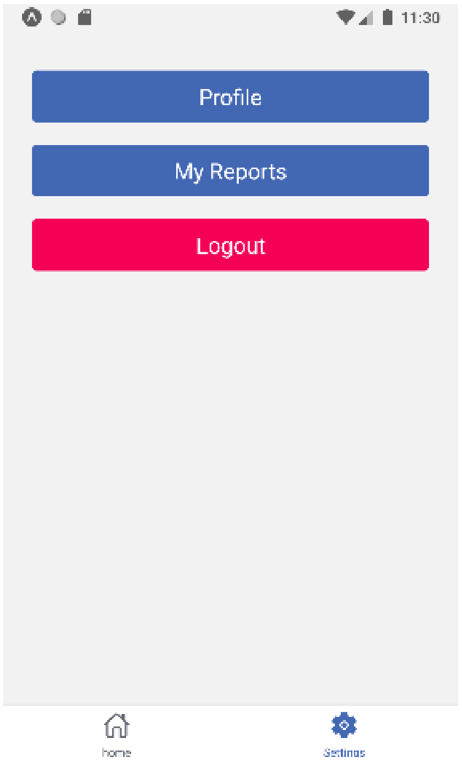


Figure 3.33: The Client Settings screen displaying the Navbar component at the bottom of the Emulator screen.

Topbar Navigator

The `Topbar` navigator is the navigation file that is responsible for displaying sessions lists whenever the user clicks on View Requests in the Home screen. The `Topbar` navigator is role based, and will load different screens depending on the role of the user. The way it is created is by creating a top tabs navigator and creating within in tab screens to accommodate the role of the user. If the role is client, then the `Topbar` navigator will return a top bar with options for `RequestedSession`, `ApprovedTopBar`, and `PreviousSession`. If the role is translator then the navigator will return a top bar with options for `TranslatorRequests`, `ApprovedTopBar`, and `TranslatorPrevious`. If the role is admin then the navigator will return a top bar with options for `RequestsTop`, `AdminAllAccepted`, and `AdminAllPrevious`. A top tab bar is created using the method `createMaterialTopTabNavigator()` from the `react-native/material-top-tabs` library

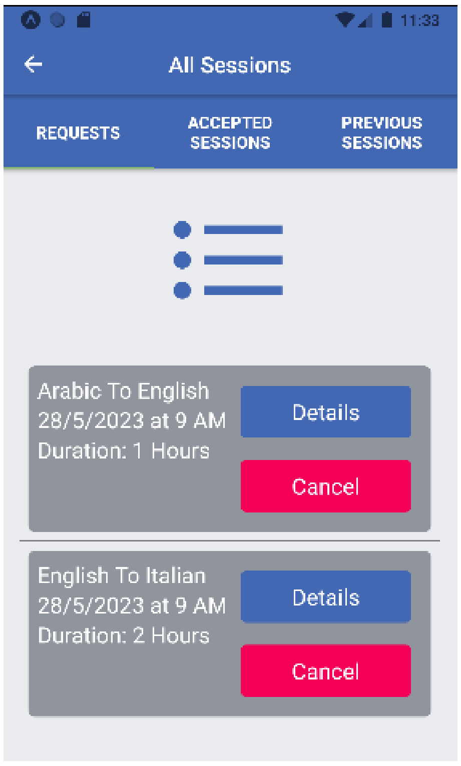
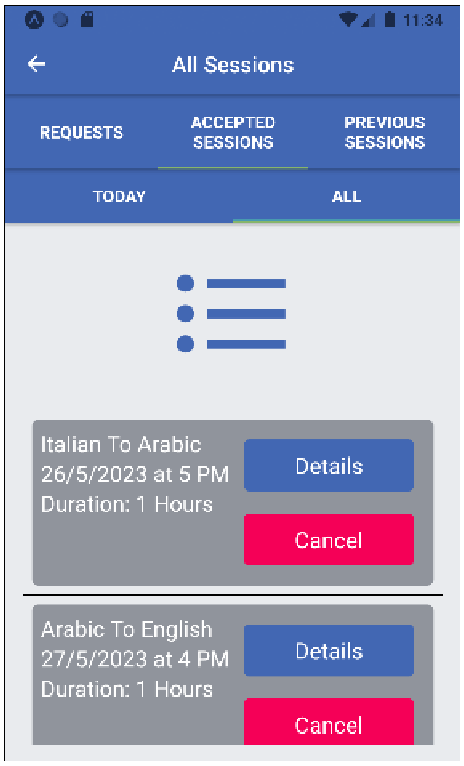


Figure 3.34: The Topbar navigator as displayed on the Emulator screen as a client presses View Requests in their Home screen.

ApprovedTopBar Navigator

The `ApprovedTopBar` navigator is the navigator responsible for displaying two tabs for either approved sessions that are today or later sessions. The `ApprovedTopBar` is added into the `Topbar` navigator as one of the screens to display when either a client or translator clicks on View Requests. The `ApprovedTopBar` has role based returns. If the role is client, then the returned screens would be the `AcceptedSession` with a time parameter of "today", and the `AcceptedSession` again but with a time parameter of "later". If the role is translator, then the returned screens would be `TranslatorApproved` with a time parameter of "today", and the `TranslatorApproved` again but with a time parameter of "later".



Italian To Arabic
26/5/2023 at 5 PM
Duration: 1 Hours

Arabic To English
27/5/2023 at 4 PM
Duration: 1 Hours

Figure 3.35: The ApprovedTopBar navigator as displayed on the Emulator screen as a client swiped to the Approved requests in the Topbar navigator display.

RequestsTop Navigator

The `RequestsTop` navigator is the navigator responsible for displaying two tabs for either unassigned requests or assigned requests. The `RequestsTop` is not role based; however, it is only displayed for one role which is the admin. The `RequestsTop` navigator is added into the `Topbar` navigator as the default screen for the render of the View Requests action by the admin. The `RequestsTop` navigator has two screens/tabs for `AdminAllRequest` and `AdminAllAssigned`.

ReportTopTab Navigator

The `ReportTopTab` navigator is the navigator responsible for displaying report status tabs for either admin or client only. The `ReportTopTab` navigator is role based, and is accessible only from the Settings screen of either admin or client. For the role of client, the `ReportTopTab` has screens for `ClientUnresolved` and `ClientResolved`, with the default being `ClientUnresolved`. For the role of admin, the `ReportTopTab` has screens for `AdminSessionUnresolved`, `AdminSystemUnresolved`, and `AdminResolved`.

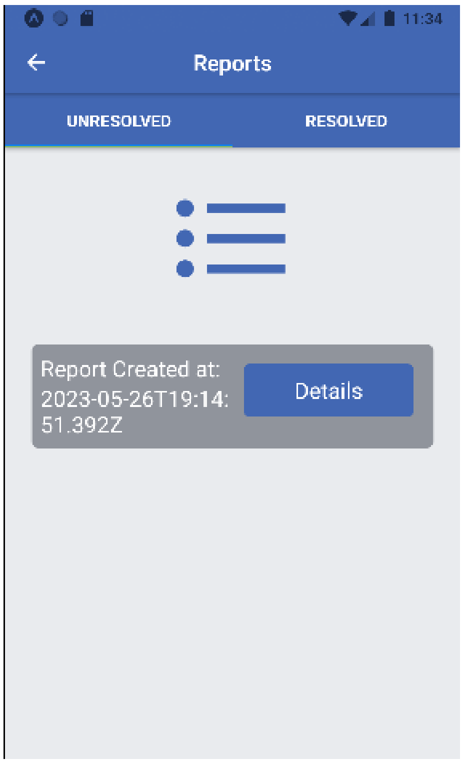


Figure 3.36: The ReportTopTab navigator as displayed on the Emulator screen as a client presses the My Reports button in the Settings screen.

AdminSeeUsersTopBar Navigator

The `AdminSeeUsersTopBar` is the navigator responsible for display list screens of all the users in the system for the admin. The `AdminSeeUsersTopBar` is only accessible by the admin. The `AdminSeeUsersTopBar` has three different stack navigators corresponding to the three different user lists. All of the stacks implement the same screen, which is the `UserList` screen. The difference between them other than title is that each stack passes a parameter called role, which is set to either one, two or three. With that said, the `UserList` screen would now behave as three different screens, as with the parameter, it will load only a list of all users with the specified role parameter. The `AdminSeeUsersTopBar` has screens made using the three Stacks, which are the `ClientScreen, TranslatorScreen, and AdminScreen` stacks. These "Screens" are all, as mentioned, stacks that all point to the same screen but with different parameters.

Draw Navigator

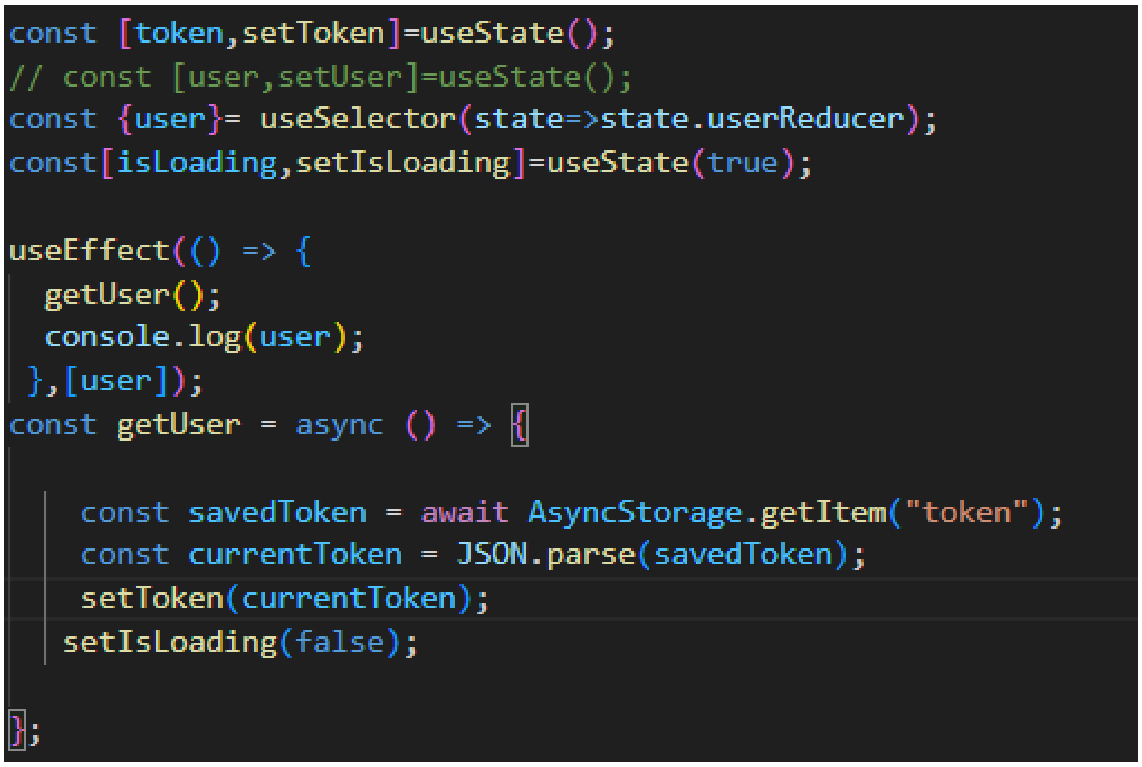
The `Draw` navigator acts as the drawer bar for the application. The `Draw` navigator is role based, and will render a different drawer bar depending on the role of the user. If the role of the user was client, then the drawer will have Home, Profile, Report, and Logout as accessible elements. If the role was either translator or admin, the drawer will have Home, Profile, and Logout as accessible elements. The Report element navigates only the client to the `ReportAll` screen. The Profile element navigates the any user to the `Profile` screen. The Home element navigates the user back to the `Home` screen. The Logout element navigates the user back to the `Welcome` screen.

All of the previously mentioned Navigators also had headers allowed with a back button that takes the user back to the previously opened screen. The `Draw` Navigator; however, does not have that implemented, but rather has a back menu button. The back menu button does not take the user back to the previous screen, but rather opens the drawer bar for the user. The drawer screens are also decorated with `IonIcons`, and to create a drawer navigator the method `createDrawerNavigator()` is used.

3.3.5 Shared Navigator Related Screens

Home Screen

The `Home` Screen is the user's main Hub and it is the screen that users are navigated to when they login to the system. The `Home` Screen is the core of all actions that a user could perform on the application. The `Home` screen creates a constant called user for which it uses `useSelector` to get the user Redux state, as well as has a constant token which is set using the `useEffect` method. The `useEffect` method calls on the getUser method, which retrieves the token from AsyncStorage through a `getItem` method with its input being the string "token. The `getItem` method then returns a JSON object so it is first parsed before it is set to the token state. The `useEffect` method also sets another state, `isLoading`, to false, and this state is what allows a page to not render before the `useEffect` is finished. The `isLoading` state is very important especially in screens that display information of states that are set by `useEffect`, as the screens must not load before `useEffect` is done. Without `isLoading`, state dependent screens will have a high chance of not loading properly. The `useEffect` is also dependent on the user Redux state, which is very important as any changes to the user state now cause the `useEffect` to be called again. This causes the screen to be re-rendered, which is important since it helps keep screens up to date with any change to the user state.



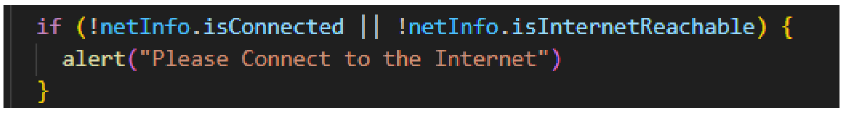
```
const [token,setToken]=useState();
// const [user,setUser]=useState();
const {user}= useSelector(state=>state.userReducer);
const[isLoading,setIsLoading]=useState(true);

useEffect(() => {
  getUser();
  console.log(user);
 },[user]);
const getUser = async () => {

    const savedToken = await AsyncStorage.getItem("token");
    const currentToken = JSON.parse(savedToken);
    setToken(currentToken);
   setIsLoading(false);

};
```

Figure 3.37: useEffect example on how states are loaded and set

The return statement of the Home screen is as mentioned above bounded by a condition for if `isLoading` is false, and only then will it render. However, the `Home` screen has different renders within that condition, as it is conditionally rendered based on the logged in user's role. If the role is equal to one, then the client's home screen is rendered, if it is two then the translator's home screen is rendered, and if it is three then admin's home screen is rendered.

Since the Home Screen is part of the `NavBar` tab navigator, the bottom tab navigator is rendered as well as part of the screen. Through the Tab Navigator, users can either go back to the `Home` screen from the `Settings` screen or go to the `Settings` screen from the `Home` screen.

The `Home` Screen, as well as all other screens in the application, are all always checked for internet connectivity upon any action performance. This is done as if the user is not connected to the internet, actions will not be handled and the screen will just remain frozen. With the use of `NetInfo`, the actions can first check if `NetInfo` is connected, which means if the device is connected to the internet. If the device is not, then instead of having the screen be frozen without a response to inform the user of this issuee, the screen will recieve an alert telling them to "Please connect to the internet".



```
if (!netInfo.isConnected || !netInfo.isInternetReachable) {
  alert("Please Connect to the Internet")
}
```

Figure 3.38: The netInfo code to check for internet connectivity which can be found in all screens that have actions.

The client home screen contains three buttons which allow them to request a session, see their requests, or view their profile. The Request Session button navigates the client to the `EnteredSession` screen. The View Requests button navigates the client to the `TopBar` navigator, which has a default screen as the `RequestedSession` screen and includes tabs to the `AcceptedSession` and `PreviousSession` screens as well. The Profile button navigates that user to the `Profile` screen, which is a draw bar screen that is also accessible through the Draw navigator. The draw bar is made accessible throughout the entire application as `Home` is navigated to through the Drawer navigator and not the `Nabvar` navigator. This decision was made since the tab navigator can only navigate to the screens it contains, while the draw bar navigator is more flexible and has more screens in it than the tab navigator, hence we gave it a higher priority.

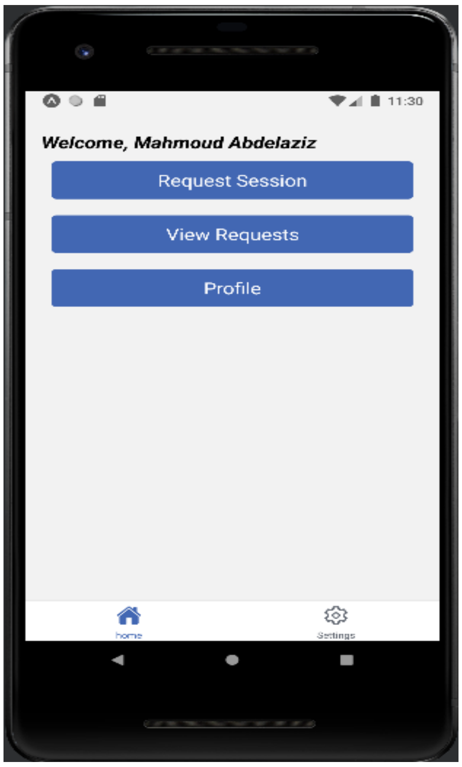


Figure 3.39: The Client Home screen displayed on the Emulator screen.

The translator Home screen contains two buttons which allow a translator to see their requests, and see their profile. The Profile button leads to the `Profile` screen. The View

Requests button navigates the translator to the same `TopBar` navigator; however, not to the same screens, as the `TopBar` navigator is also conditionally rendered based on user role. Therefore, while for client the screens were the `RequestedSession,AcceptedSession, and PreviousSession` screens, for transalator it is `TranslatorRequest, TranslatorApproved, and TranslatorPrevious`. The `TranslatorRequest` screen in this case is the `TopBar`'s default screen.

The admin Home screen contains four buttons which allow an admin to see their requests, add a new user, see all users, and see their profile. The View Requests and Profile buttons provide a similar feature as the previous two screens, with the difference being the screens that are rendered for an admin upon clicking View Requests. The See Users button navigates the admin to the `AdminSeeUsersTopBar` navigator, which has screens for viewing lists of the different types of users. The Add User button navigate the admin to the `AddUserHome` screen, which has three buttons to take the admin to the `AddUser` screen of a specific type of user.

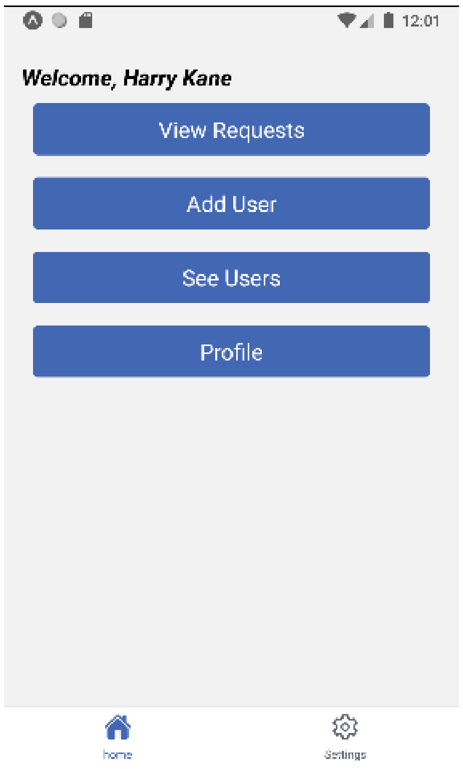
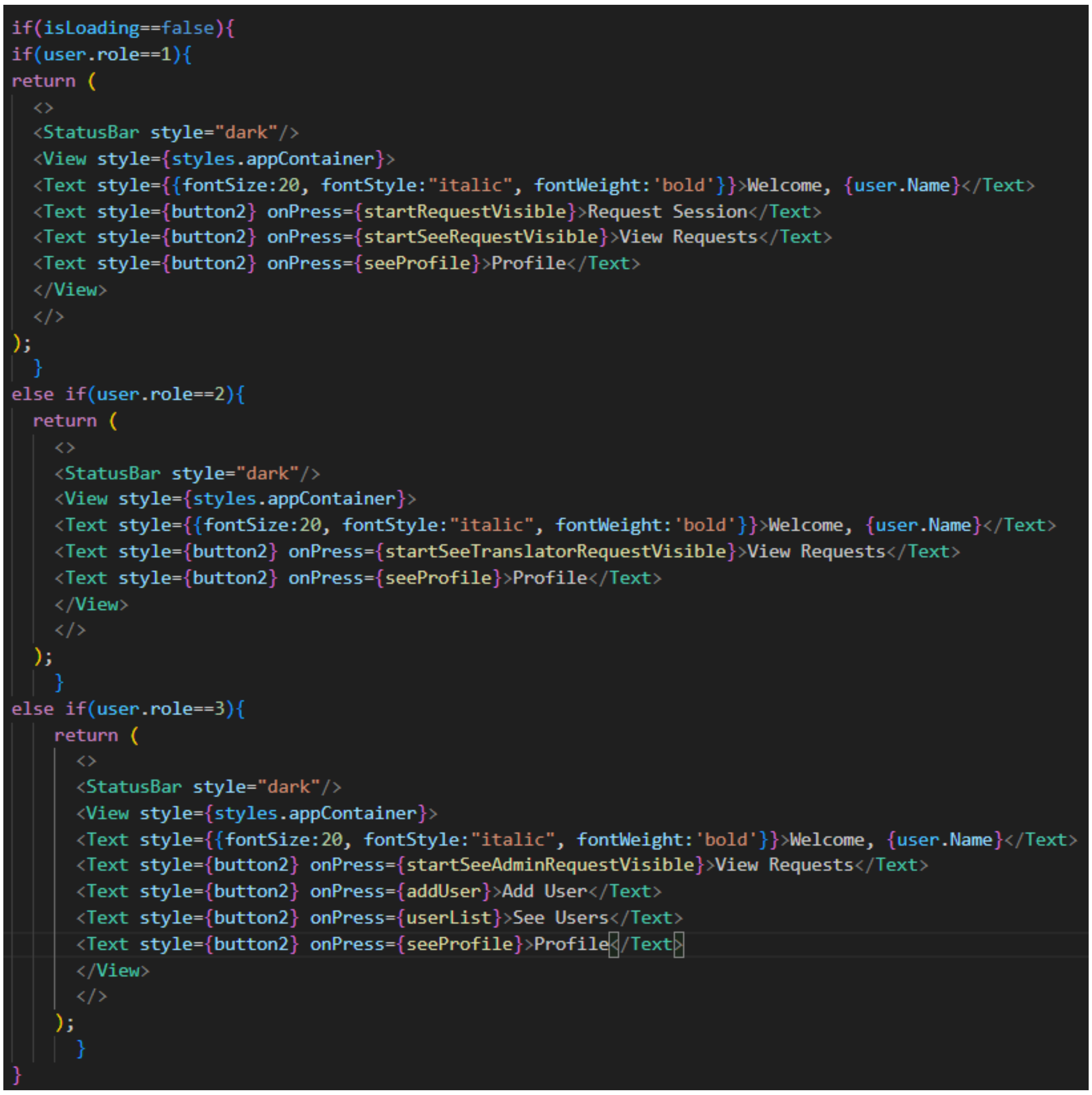


Figure 3.40: The Admin Home screen displayed on the Emulator screen.



```
if(isLoading==false){
if(user.role==1){
return (
  <>
  <StatusBar style="dark"/>
  <View style={styles.appContainer}>
  <Text style={{fontSize:20, fontStyle:"italic", fontWeight:'bold'}}>Welcome, {user.Name}</Text>
  <Text style={button2} onPress={startRequestVisible}>Request Session</Text>
  <Text style={button2} onPress={startSeeRequestVisible}>View Requests</Text>
  <Text style={button2} onPress={seeProfile}>Profile</Text>
  </View>
  </>
);
  }
else if(user.role==2){
  return (
    <>
    <StatusBar style="dark"/>
    <View style={styles.appContainer}>
    <Text style={{fontSize:20, fontStyle:"italic", fontWeight:'bold'}}>Welcome, {user.Name}</Text>
    <Text style={button2} onPress={startSeeTranslatorRequestVisible}>View Requests</Text>
    <Text style={button2} onPress={seeProfile}>Profile</Text>
    </View>
    </>
  );
    }
else if(user.role==3){
    return (
      <>
      <StatusBar style="dark"/>
      <View style={styles.appContainer}>
      <Text style={{fontSize:20, fontStyle:"italic", fontWeight:'bold'}}>Welcome, {user.Name}</Text>
      <Text style={button2} onPress={startSeeAdminRequestVisible}>View Requests</Text>
      <Text style={button2} onPress={addUser}>Add User</Text>
      <Text style={button2} onPress={userList}>See Users</Text>
      <Text style={button2} onPress={seeProfile}>Profile</Text>
      </View>
      </>
    );
      }
}
```

Figure 3.41: The Home screen returned View code

Settings screen

The `Settings` screen is the screen in charge of providing settings related actions to the users. The `Settings` screen is a role based screen, where depending on the user role a different screen will be rendered. If the role is client then the screen will load with buttons for `Profile, My Reports, and Logout`. If the role is admin then the screen will load buttons for `Profile, Reports, Add Language, and Logout`. If the role is translator then the screen will load buttons for `Profile and Logout` only. All of the buttons mentioned before navigate to specific screens. The Profile button navigates the user to the `Profile` screen identified in the Draw navigator. The Reports and My Reports button navigate the user to the `ReportTopTab` navigator, which will load screens specific to the role of the user. The Add Language button will navigate the admin the the

AddLanguage screen. Finally, the Logout button will navigate the user to the Welcome screen.

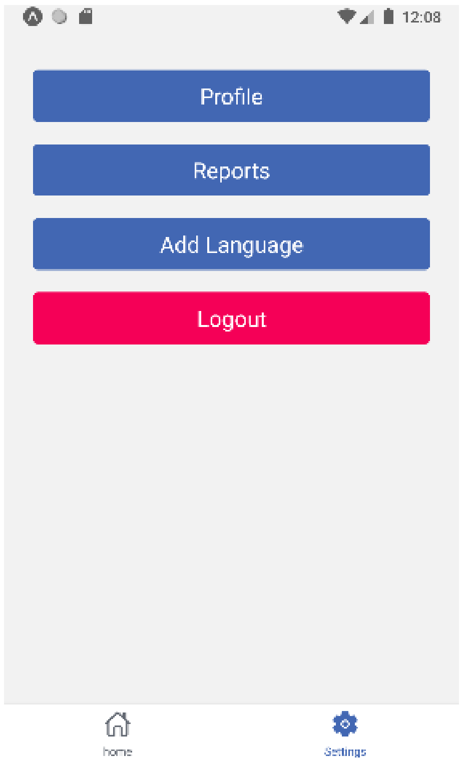


Figure 3.42: The Settings screen from the admin point of view.

Profile Screen

The Profile screen is the screen in charge of displaying the user's profile. The Profile screen is accesible through the Settings and Home screens, and through the Draw navigator. The Profile screen is one of the screens that identified in the Draw navigator's screens, and hence is navigable to through the Draw navigator using the navigation commands. The Profile screen is role specific, as translators have a different screen than admins and clients. The basic Profile screen for admins and clients is a screen that shows their name, profile icon, username, role, phone number, and email. For the translators, there is also one more field added which is a button titled "Add Language". The "Add Language" button navigates the translator to the TranslatorLanguages screen, where the translator can manage their languages. At the top right of the Profile screen, there is an icon for which when a user presses on they will be navigated to the EditProfile screen.

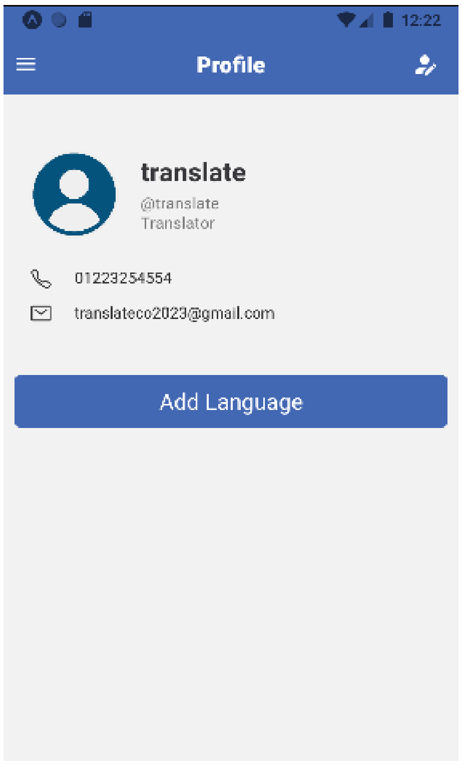


Figure 3.43: The Profile screen from the translator point of view.

3.3.6 Client Navigator Related Screens

RequestedSession Screen

The `RequestedSession` screen is the screen that shows up first whenever a client presses the View Requests button on the Home screen. The `RequestedSession` screen is a screen that provides a list of all the requests that the currently logged in client has requested and have not been accepted/approved by a translator yet. The way a list screen works is that it makes use of `useEffect` to call on a function which loads up a state with an array of the items to display in the list. In this case the `useEffect` method call on the `loadUserRequests` method which calls on the `MyRequests` Backend method. After the `useEffect` is completed, the screen can then be rendered to display the list, which is made using a `Flatlist` component. All list components in this application are also refreshable and have an activity indicator upon refreshing to display to the user that a refresh is occurring. The `Flatlist` component then maps the array of items in a list fashion separated by lines.

Each list screen as well can have its own additional features to the `Flatlist` elements. In the case of `RequestedSession`, the list elements all have buttons to cancel or to view the details of the request. Upon clicking the cancel button, an Axios call is made to the `cancelRequests` Backend method, which in this case cancel the request and refund to the user their money if they paid by card. The `cancelRequests` Backend method also sends a cancellation confirmation email to the client to ensure they know about the cancellation. The Details button navigates the user to the `SessionDetails` screen and passes to it as parameters the details of the session item from the list.

All Backend methods that send Emails to the users in this application have activity indicators loaded while the action is being performed. This is done to let the user know that their action is being handled, rather than having them stare at the same screen until it updates after the action is performed. This is a measure to ensure proper user experience, as while data heavy actions are being performed it takes a while for the effects to be noticed. With this idea in mind, the user without the activity indicator would just be staring at the screen, which might make them feel as if the action did not go through. This might make the user feel prompted to press it again, while in reality the action was actually being handled.

AcceptedSession Screen

The `AcceptedSession` screen is a screen that is used in the `ApprovedTopBar` navigator for the render of the client. The `AcceptedSession` screen is the Screen responsible for the displaying of lists consisting of all the approved requests for the logged in client that have been approved/accepted by a translator. The `AcceptedSession` screen has as input a parameter called time, which is passed to it from its initialization in the `ApprovedTopBar` navigator as an `IntialParams`. Based on the time parameter, the screen will either only display the approved session whose dates are today, or will display those that are not today hence later. The `AcceptedSession` screen is initialized twice in the `ApprovedTopBar` navigator, hence it allows the `ApprovedTopBar` navigator to have two tabs which display different data, even though it is the same screen.

The `AcceptedSession` screen is a list format screen like the `RequestedSession` screen and has exactly the same structure as the `RequestedSession` screen except with the addition of conditionally checking whether the time parameter is "today" or "later" to load different screens. The sessions are loaded into the screen's state through the Backend method `myApprovedRequests`, which returns all of that client's approved sessions. For sessions that are today, the session list items are loaded with only the details button available before the time of the reservation itself. If the current time is the same as or after the time of the session's reservation, then another button will appear displaying accept. The accept button is the button that appears as a response to the translator starting the session from their end. This accept button will only appear; however, in the condition that it is an `InPerson` session. Once the accept button is pressed, an Axios method will be called to use the `AcceptStartSession` Backend method. With this the session will then be removed from this list and will be added to the `PreviousSession` screen's list. The details button does the same as the one in the `RequestedSession` screen. For sessions that are later, the cancel button will now be made visible and upon pressing it the `cancelApprovedRequests` Backend method is called. This also makes a refund to the client if the payment was through card.

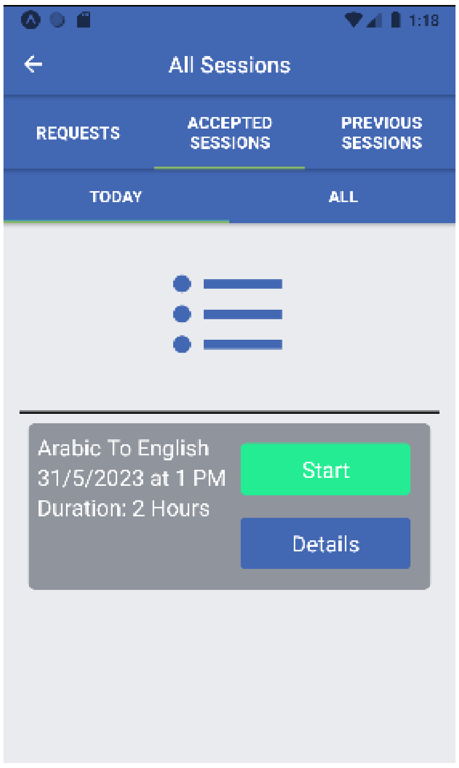


Figure 3.44: The Accepted Sessions screen in the ApprovedTabBar navigator for client screen with the Today sessions tab open.

PreviousSession Screen

The `PreviousSession` screen is also a list format screen, which calls on the Backend method `myPreviousRequests` to set the state that is used mapped by the `FlatList`. This screen simply presents all the previous sessions that belong to the logged in user. All of the session list items can have two buttons, which are Report and Details. If the session has not been previously reported, the both the Report and the Details buttons will be shown. If the session has been reported, then only the Details button will be shown. The details button is the same as it was for the previous two screens. The Report button navigates the client to the `ReportType` screen with the the session item as a parameter as well as another parameter called `ReportType`, which is set to "Session".

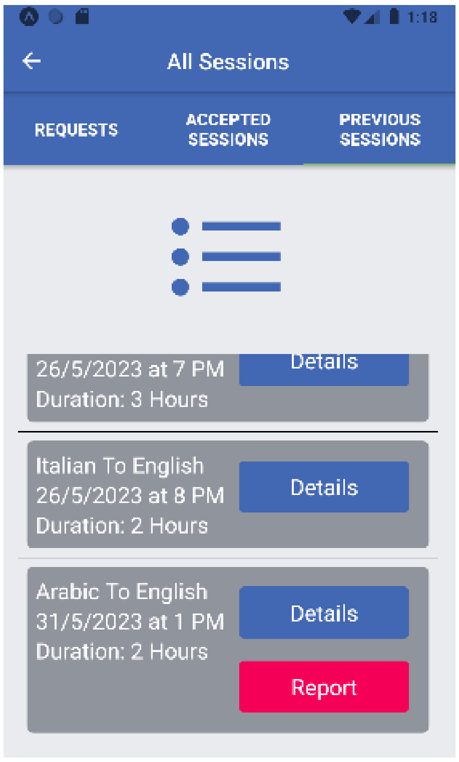


Figure 3.45: The Previous screen in the ApprovedTabBar navigator for client screen.

ClientUnresolved Screen

The `ClientUnresolved` screen is the screen that displays all of the reports, submitted by the logged in client, that have not yet been resolved by an admin. The `ClientUnresolved` is the first screen that renders when the client presses the My Reports button in the settings, as it is the default screen in the `ReportTopTab` navigator for the client role. The `ClientUnresolved` is a list format screen that lists all of the clients unresolved reports. The reports are retrieved and set into the state through `useEffect` like all other list screens in this application. The Backend method that the `useEffect` accesses is the `myNonResolvedReports` method, through the locally defined method `loadUserReports`. The report list items all have only one button, which is the Details button. The Details button navigates the client to the `ReportDetails` screen, and passes to it the report list item's details as parameters.

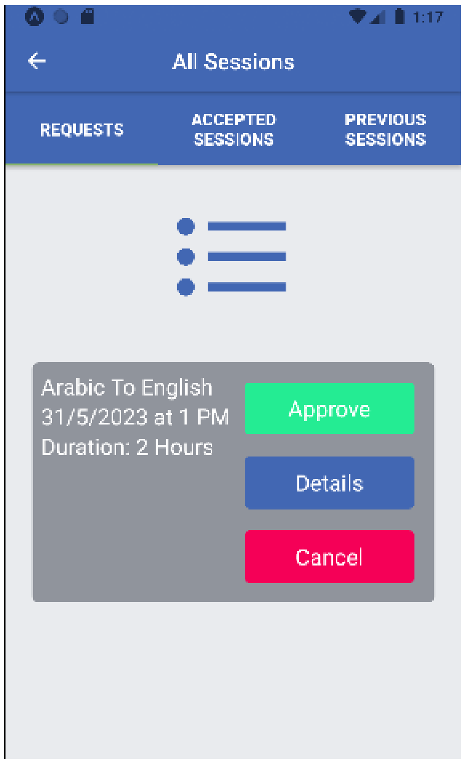
ClientResolved Screen

The `ClientResolved` screen is the screen that is responsible for and displays a list of all reports, submitted by the client, that have been resolved by an admin. The `ClientResolved` is the second screen that is part of the `ReportTopTab` navigator. The Backend method it used to set the reports state is the `myResolvedReports` method. The `ClientResolved` screen is identical to the `ClientUnresolved` screen, with the only difference being the Backend method it calls, since it will return different reports.

3.3.7 Translator Navigator Related Screens

TranslatorRequests Screen

The `TranslatorRequests` screen is the screen that displays all of the requests that have been assigned to the logged in translator that are waiting for approval. The `TranslatorRequests` screen is a list format screen, and gets its sessions to set in the `FlatList` state using the `myRequestsT` Backend method. All of the sessions displayed in the list have buttons for Approve, Details, and Cancel. The Approve button makes a call to the `acceptSession` Backend method and passes to it as parameters the session details. The Details button does the same action as all sessions Detail buttons have done before. The Cancel button makes a call to the `declineRequest` Backend method and passes to it as a parameter the session id.

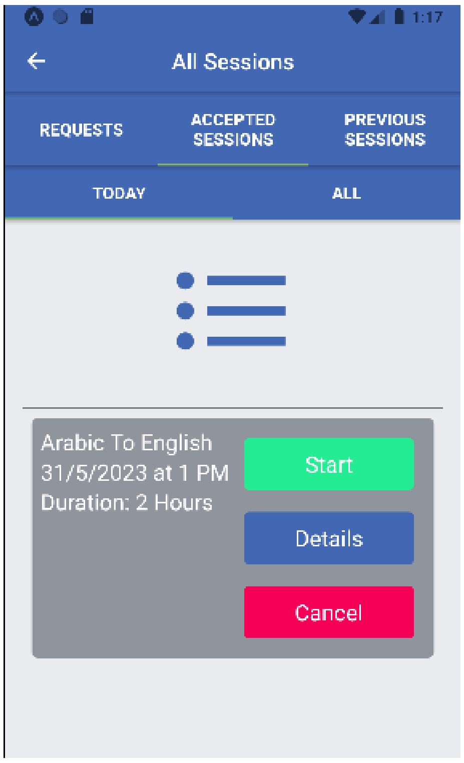


Arabic To English
31/5/2023 at 1 PM
Duration: 2 Hours

Figure 3.46: The TranslatorRequests screen in the Topbar navigator.

TranslatorApproved Screen

The `TranslatorApproved` screen is the screen that displays all of the sessions that the logged in translator has approved of. This screen is almost identical to the `AcceptedSession` screen in the client side, as it also receives the time parameter to load two different screens for the `ApprovedTopBar`. The only key difference between them is the Start button, where here the translator will receive a start button on the session that should start at the current time. If the type of session is `OverThePhone`, then pressing the start button will call on the `startSession` Backend method and pass to it the request id and the type of the session. In addition, the translator's phone dial app will open up with the client's number ready to dial. If the session type was `InPerson`, then all the start button will do is call on the `startSession` Backend method. This will in turn also affect the client's AcceptedSession screen as now that session will have an Accept button on it.



Arabic To English
31/5/2023 at 1 PM
Duration: 2 Hours

Figure 3.47: The TranslatorApproved screen in the ApprovedTabBar navigator for translator screen.

TranslatorPrevious Screen

The `TranslatorPrevious` screen is the screen that loads up all of the previous session of the logged in translator. The `TranslatorPrevious` screen is also almost identical to the `PreviousSession` screen in the client side. The only difference between them, other than the Backend method called and the set states, is that the translator does not have a Report button on their sessions.

3.3.8 Admin Navigator Related Screens

AdminAllRequest Screen

The `AdminAllRequest` screen is the screen that displays all of the sessions' requests made by all clients in the system that have not been assigned yet. The `AdminAllRequest` is the screen first viewed once an admin presses the View Requests button on the home page, as it is the default screen in the `RequestsTop` navigator. The `AdminAllRequest` is a list format screen, and is very similar to the `RequestedSession` screen in the client view. The major differences between them is that each session list item has an additional button which is the Assign button, and that only requests without assigned translators are loaded.

The Assign button navigates the admin to the `AssignTranslator` screen, where they can assign a translator to said session. The Assign button also sends as parameters to the `AssignTranslator` screen the session details. The cancel button on all the list elements there also call on a different Backend method which is the `cancelRequestsA` method. The only difference between this one and the one in the client view is the purpose of it. The purpose of the cancel button in the `AdminAllRequest` screen is to cancel a session given that there are no translators that wish to accept the session or if there no translators available. The email that is sent to the client is also different to the one sent when a client cancels in the `RequestedSession` screen. The Details button does the same action as before.

The Sessions are loaded by using the `Requests` Backend method in the `useEffect`. This method returns only the unassigned sessions back.

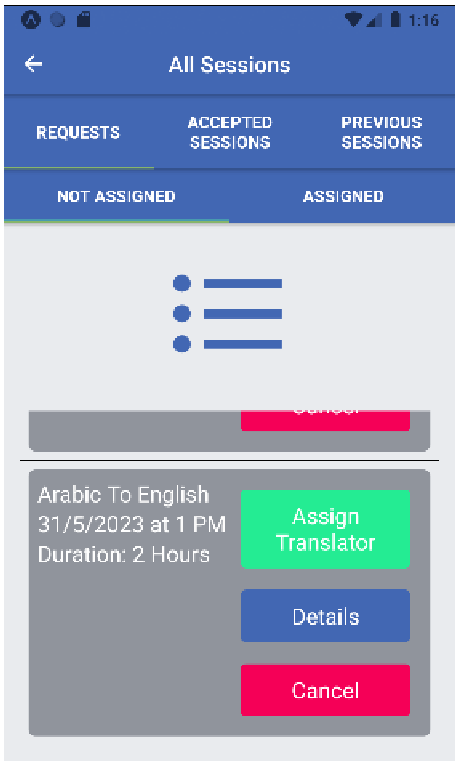
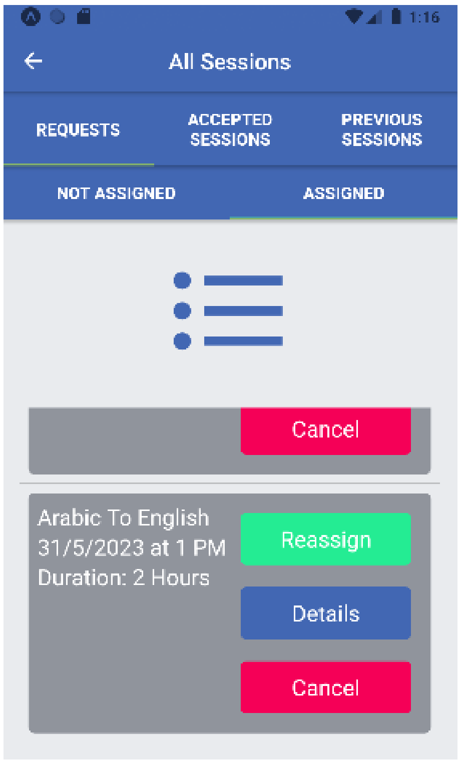


Figure 3.48: The AdminAllRequest screen in the RequestsTop navigator.

AdminAllAssigned Screen

The `AdminAllAssigned` screen is the screen that displays all of the requests made by all clients in the system that have already been assigned. The `AdminAllAssigned` is the second screen accessible from the `RequestsTop` navigator. The `AdminAllAssigned` screen is very similar to the `AdminAllRequest` screen, where the only differences are that the Assign button is replaced by the Reassign button, and that only assigned requests are loaded. The Reassign button should only be used if the request had not been accepted for a while, as a normal decline from a translator would simply remove it from this screen and place it in the `AdminAllRequest` screen. The Reassign button makes a call to the `Reassign` method, which only removes the translator's email from the session and places it in the request's `CancelledBy` array. The Reassign button then navigates the admin to the `AssignTranslator` screen. The requests are loaded by using the Backend method `AssignedRequests`.



Arabic To English
31/5/2023 at 1 PM
Duration: 2 Hours

Figure 3.49: The AdminAllAssigned screen in the RequestsTop navigator.

AdminAllAccepted Screen

The `AdminAllAccepted` screen is the screen responsible for loading all the accepted/approved sessions in the system. The sessions are loaded using the `ApprovedRequests` Backend method. The screen does not have any other feature to it other than a Details button just like all the previous session list screens.

AdminAllPrevious Screen

The `AdminAllPrevious` screen is the screen that is responsible for loading all the previous sessions in the system. The sessions are loaded using the `PreviousRequests` Backend method. The]`AdminAllPrevious` is identical to the `TranslatorPrevious` screen. The only difference is the Backend method it calls on, thus it will show more session list items than the `TranslatorPrevious` screen since it will load all of the previous sessions in the system.

AdminSessionUnresolved Screen

The `AdminSessionUnresolved` is the screen responsible for displaying all the session reports submitted to the system by all clients. The `AdminSessionUnresolved` is a list type screen, and it calls on the Backend method `SessionUnresolved` to set the state mapped by the FlatList. The `AdminSessionUnresolved` screen's list items all have a Resolve button. The Resolve button is the button that is used to navigate to the `ResolveReport` screen, and passes to it as parameters the report's details. The `ResolveReport` screen in this case will load up a resolve session form for the admin to resolve the session report.

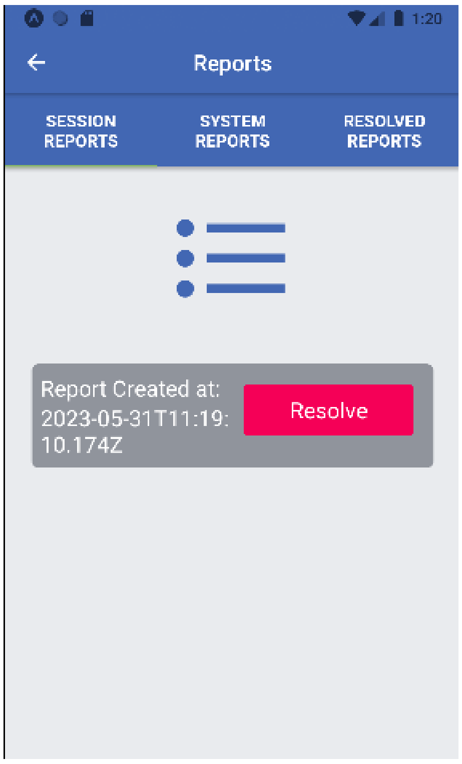


Figure 3.50: The AdminSessionUnresolved screen in the ReportTopTab navigator.

AdminSystemUnresolved Screen

The `AdminSystemUnresolved` is the screen responsible for displaying all the session reports submitted to the system by all clients. The `AdminSystemUnresolved` is a list type screen, and it calls on the Backend method `SystemUnresolved` to set the state mapped by the `FlatList`. The `AdminSystemUnresolved` screen's list items all have a Resolve button, like the `AdminSessionUnresolved` screen. The Resolve button here also navigates to the `ResolveReport` screen with the report details as parameters. The `ResolveReport` screen in this case will load up a resolve system form for the admin to resolve the system report.

AdminResolved Screen

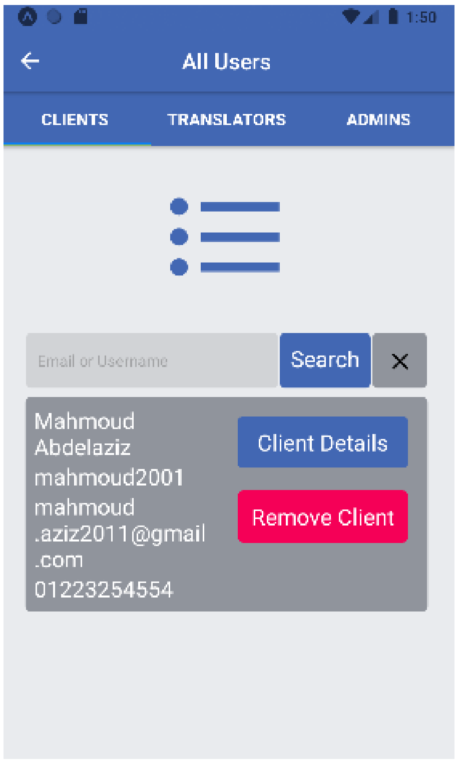
The `AdminResolved` screen is the screen that displays all of the resolved reports in the system. It is a list format screen and calls the Backend method `ResolvedReports` to fill in the state mapped by the `FlatList` component. The `AdminResolved` screen is identical to the ClientResolved screen in the client view, with the exception being the Backend method called. The `AdminResolved` screen also has the Details button, which also navigates the admin to the `ReportDetails` screen with the report's details as parameters.

UserList Screen

The `UserList` screen is the screen that is accessed from the admin home screen by pressing the button "See Users". The `UserList` screen is also the screen that is used in the `AdminSeeUsersTopBar` navigator. The `UserList` screen is a list format screen, and the way it retrieves the users to put into the `FlatList`'s state is through the Backend method `getUsers`. The `getUsers` method is also passed the role required is a parameter called role. The role parameter is retrieved from the route parameter, which is the parameter that is passed to it from the `AdminSeeUsersTopBar`.

As previously mentioned, the `AdminSeeUsersTopBar` creates three screens of the `UserList` screen, for listing clients, translators, and admins respectively. Each list item has two buttons which are Details and Remove. The Details button navigates to the `UserDetails` screen and passes to it as parameters the user list item's details. The Remove button calls on the Backend method `removeUser` and passes to it as a parameter the id of the user list object.

The `UserList` screen not only has the listing element, but it also has a search bar above the list. The search bar element is a text box with a button next to it to search. The way it works is that once the admin enters either a user's username or email and presses the search button, the `SearchUser` Backend method is called and is passed to as parameters the entered email or username as well as the role of the current list/version of `UserList`. The Backend then will either respond with a message saying the user is not in the system if the entered data doesnt match the role or is not a user found, or will respond with a user object. This user object is then mapped to the state that the list uses and is the only object that is present in the list at that point. After the search as well, a button with an "X" icon appears next to the search button, and upon pressing it, the `getUsers` Backend method is called again. By the Backend method is called, like previously mentioned, we mean that an Axios method calls the Backend method. Thus the list is refilled with all the users belonging to that role.



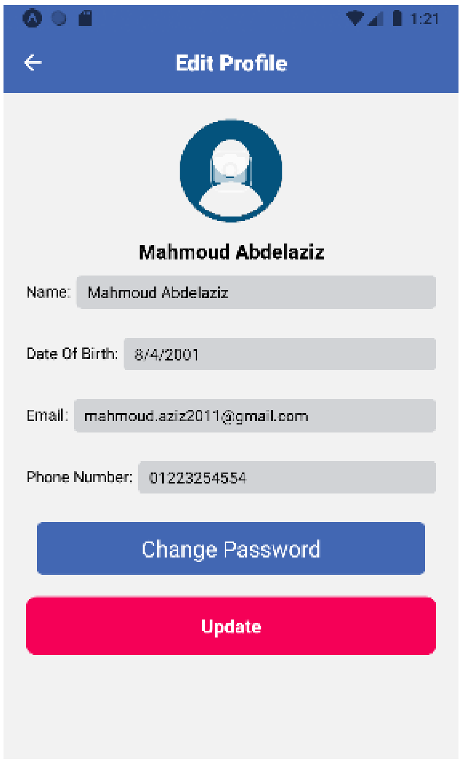
Mahmoud Abdelaziz
mahmoud2001
mahmoud.aziz2011@gmail.com
01223254554

Figure 3.51: The UserList screen on the client tab with a search query for the user with the username `mahmoud2001`

3.3.9 User Shared Screens

EditProfile Screen

The `EditProfile` screen is the screen responsible for allowing the user to edit their personal information. The `EditProfile` screen is accessible through the icon at the top right of the Profile screen. In the `EditProfile` screen, the user is presented with four different fields for which they could modify, and those are name, email, date of birth, and phone number. At the bottom of the `EditProfile` screen is a button titled update, and upon pressing it the Backend method `updateProfile` is called. The response of the method is then stored in the Redux user state, so that the changes are made global on the user's side.



Mahmoud Abdelaziz

Name: Mahmoud Abdelaziz

Date Of Birth: 8/4/2001

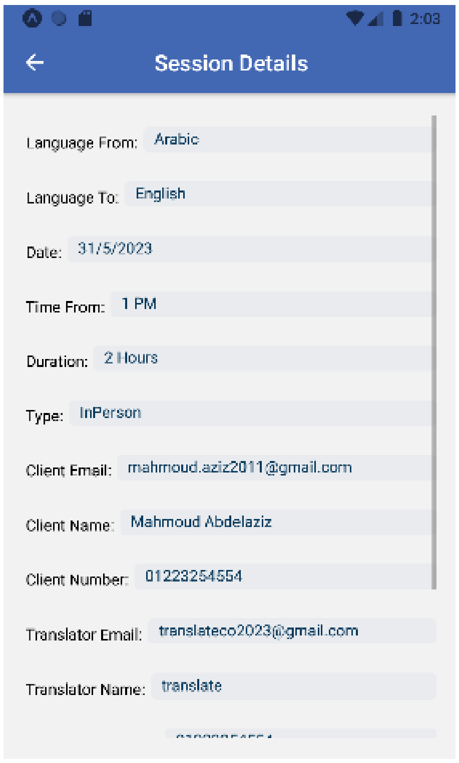
Email: mahmoud.aziz2011@gmail.com

Phone Number: 01223254554

Figure 3.52: The EditProfile screen from the client point of view.

SessionDetails Screen

The `SessionDetails` screen is the screen that displays all the details belonging to a specific session. The details that the `SessionDetails` screen receives are passed to it as parameters from any of the session listing screens. The `SessionDetails` screen is role based, as not all details of a session should be revealed to all users. For example, the client does not need to see their email, name, and phone number in the details. Using the same example, the translator does not need to see their own information as well. The only role that gets to see all of the details is the admin role. However, none of the roles get to see the `pid` of any of the sessions, as it is confidential payment information that should only be known to Stripe. The `SessionDetails` screen is simply just a scrollable View with all the details of a session represented in text fields.

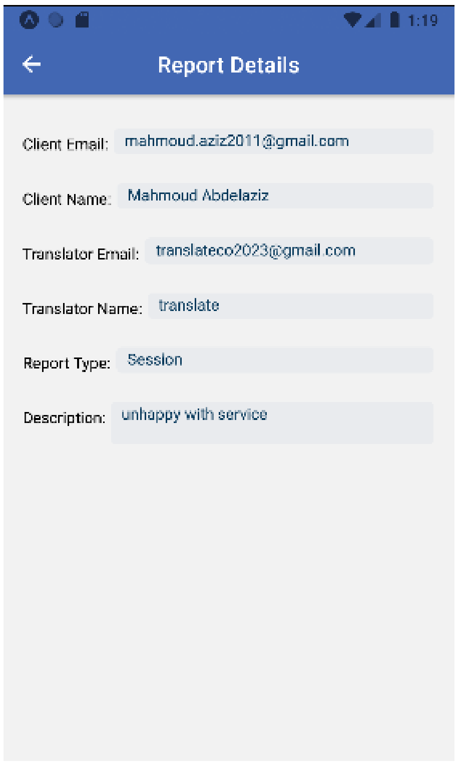


Language From: Arabic
Language To: English
Date: 31/5/2023
Time From: 1 PM
Duration: 2 Hours
Type: InPerson
Client Email: mahmoud.aziz2011@gmail.com
Client Name: Mahmoud Abdelaziz
Client Number: 01223254554
Translator Email: translateco2023@gmail.com
Translator Name: translate

Figure 3.53: The SessionDetails screen from the admin point of view.

ReportDetails Screen

The `ReportDetails` screen is the screen responsible for displaying the details of a specific report. The `ReportDetails` screen receives the details of the report from any of the reports listing screens. The `ReportDetails` screen is role based, as it is not available in general to translators as well as that it hides certain information from clients. The `ReportDetails` screen follows the same format as the `SessionDetails` screen. The `ReportDetails` is not only conditionally rendered based on role, but it is also base on the type of report, where if it is a system report then not all of the report attributes should be shown, else all would be shown. For the client's view, if the report is of type "Session", the `TranslatorWarnStatus` attribute is hidden from the client. For the admin all attributes are mapped to the fields.



Client Email:	mahmoud.aziz2011@gmail.com
Client Name:	Mahmoud Abdelaziz
Translator Email:	translateco2023@gmail.com
Translator Name:	translate
Report Type:	Session
Description:	unhappy with service

Figure 3.54: The ReportDetails screen from the client point of view.

DrawerDesign Screen

The `DrawerDesign` screen is the screen that contains the style of the `Draw` navigator. The `DrawerDesign` screen is separates the drawer into three sections: a header, a body, and a footer. The header component includes the user profile icon as well as their name. The body components loads in all of the properties of the Draw navigator, which are the options that are routable to from the Draw navigator. The footer contains the contact information of the company, as well as a logout feature. The `DrawerDesign` screen is what makes the Draw navigator look that way it looks, and is applied only the Draw navigator through the `drawerContent` prop in the Draw navigator.

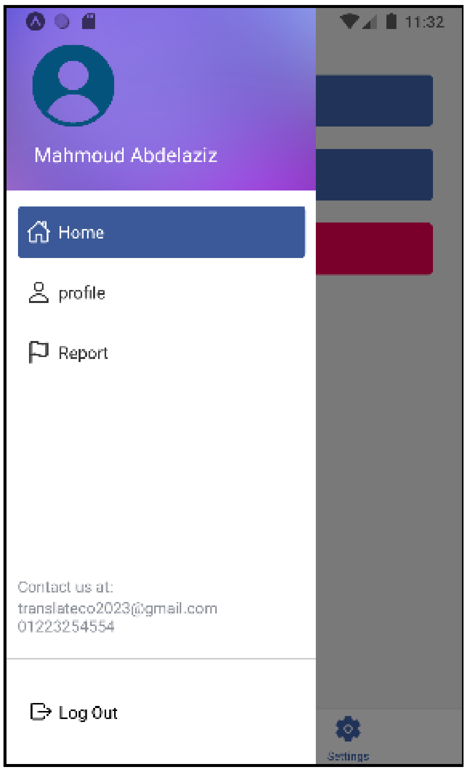


Figure 3.55: The Draw Navigator element displayed on the Emulator screen.

3.3.10 Client Specific Screens

EnteredSession Screen

The `EnteredSession` screen is the screen that allows a client to create a new request for a session. This screen is the most important one out of all of the screens, as without it request creation would not be possible. The `EnteredSession` screen is a form format screen, which has a form with four drop down lists and two buttons which enable a date picker and a time picker. There are two drop down lists which represent the Language-From and LanguageTo attributes of a request. These two drop down pickers are filled with all the languages in the system by calling the `getLanguages` Backend method in the `useEffect`. The other two drop down lists are for the duration of the session and the type of session.

The date and time pickers are highly constricted, as whenever a user picks a date or time, the `onPick` method is called. We designed the `onPick` method to check for the cases of if the day is from today and onward and not a past day, if the time is atleast an hour ahead. It also makes sure to alert the user to pick the date first before picking the time. This is done to be able to detect the time constraints correctly based on if the day is today or in the future.

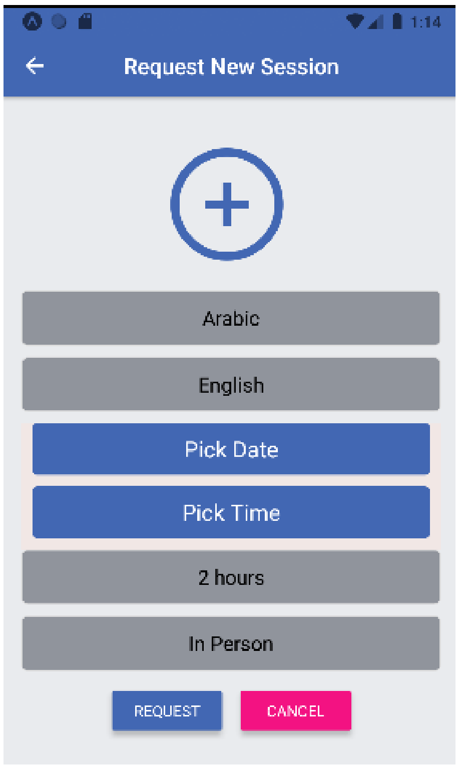


Figure 3.56: The EnteredSession screen.

The Price of the session as well is affected by the duration as well as the type of it. As depending on the length and type of the session the price will change. The prices that were set courtesy of our own thought, and the general idea of it is `OverThePhone` sessions are cheaper than `InPerson` sessions.

Once the client presses on the request button, they are shown one of two modal renders. If the request type is `OverThePhone`, then the client would get a modal asking to for payment by card, and if `InPerson` then they get the option of either cash or card. If the client chooses card, the `intent` Backend method is first called to make a payment intent, then the Stripe payment sheet is opened for the user to enter their details. After a successful payment, the request is created and the `pid` from the payment intent is stored along with it.

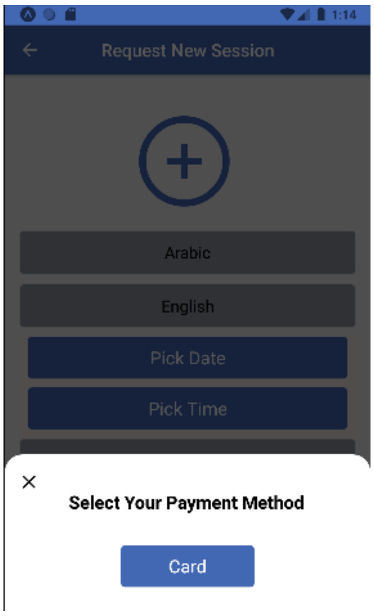


Figure 3.57: The payment modal that appears when a client presses the Request button for an `InPerson` session in the EnteredSession screen.

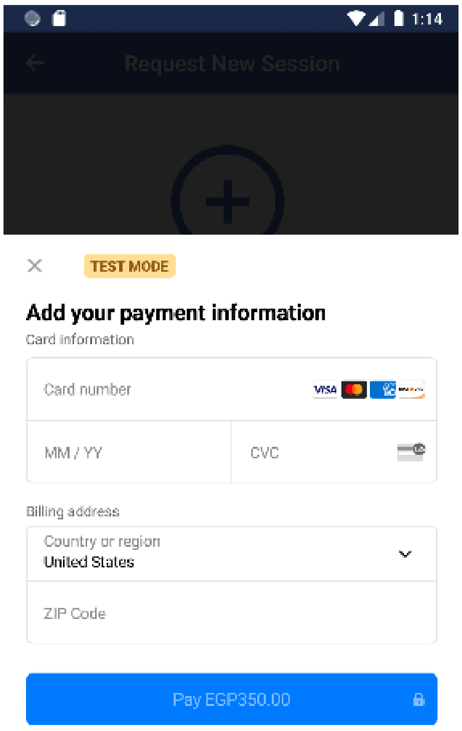


Figure 3.58: The payment sheet that appears on the EnteredSession screen if the client chooses to pay with card.

ReportAll Screen

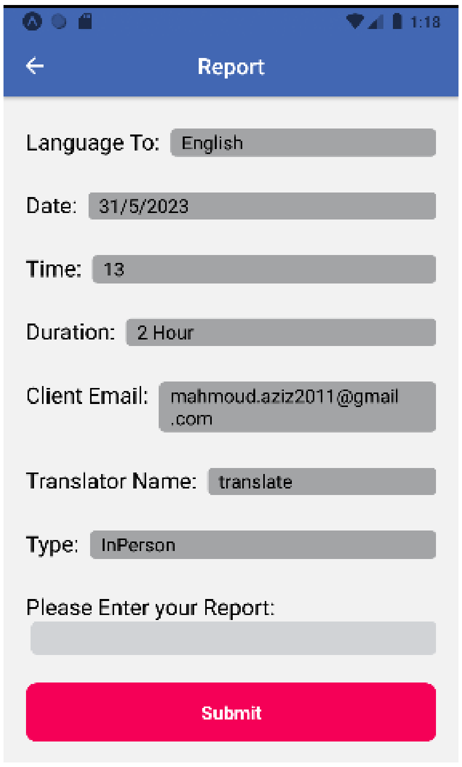
The `ReportAll` screen is the screen that the client is navigated to upon clicking on the report option in the Draw navigator. The `ReportAll` screen is a simple screen that has two buttons, one for system reports and one for session reports. Upon pressing on the Report System button, the user is navigated to the `ReportType` screen with the parameter `ReportType` being equal to "System". If the client was to press on the Report Session button, the client would then be navigated to the `UnreportedPrevious` screen.

UnreportedPrevious Screen

The `UnreportedPrevious` screen is the screen that follows after pressing the Report Session button in the `ReportAll` screen. The `UnreportedPrevious` screen is a list format screen and it is an identical copy of the `PreviousSession` screen with a minor change. The difference between the two screens is that the `PreviousSession` screen would load all of the previous sessions, reported or unreported. The `UnreportedPrevious` screen on the other hand displays only the unreported previous sessions, for which the client can then choose from to report. The way to report one of the sessions is through the Report button on each list item, and upon pressing it the client is navigated to the `ReportType` screen with the parameters including the session details as well as the `ReportType` parameter being equal to "Session".

ReportType Screen

The `ReportType` screen is the screen for which a client can report any issue, whether session or system. The ReportType screen is navigated to from the `PreviousSession`, `UnreportedPrevious`, and the `ReportAll` screens. The `ReportType` screen is conditionally rendered based on the `ReportType` parameter. If The `ReportType` parameter is equal to "Session", then the session details would be displayed to the client in a scrollable form along with a text box to enter their report description. If the `ReportType` parameter was "System", then only the text box would be rendered. Upon filling in the description text box and pressing submit, the session details and the entered report description are sent to the Backend method `Report`, for which it creates a new report that is yet to be resolved.



Language To: English

Date: 31/5/2023

Time: 13

Duration: 2 Hour

Client Email: mahmoud.aziz2011@gmail.com

Translator Name: translate

Type: InPerson

Please Enter your Report:

Figure 3.59: The ReportType screen with the render of reporting a Session.

3.3.11 Translator Specific Screens

TranslatorLanguages Screen

The `TranslatorLanguages` screen is the screen that allows the translator to add or remove languages from their language set. The `TranslatorLanguages` screen is a list type screen that is accessed from the translator Profile screen. The `TranslatorLanguages` screen first renders all of the languages that currently belong to the translator in a `FlatList` using the `myLanguages` Backend method to retrieve those langauges. The `TranslatorLanguages` screen also has a drop down list with a button next to it which contains all of the languages that the translator has not yet added to their language array. upon selecting and pressing the add button, the Backend method `addLanguages` is called and the translator's language array is updated to include the selected language.

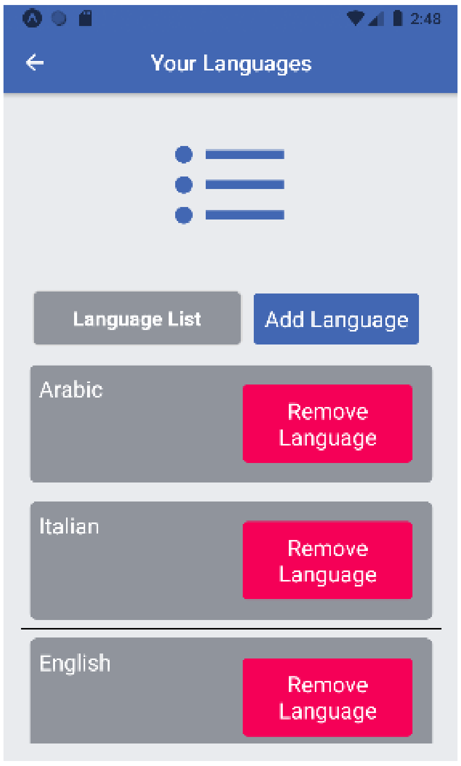


Figure 3.60: The TranslatorLanguages screen.

3.3.12 Admin Specific Screens

AssignTranslator Screen

The `AssignTranslator` screen is the screen that the admin is navigated to upon pressing Assign in the `AdminAllRequest` screen or Reassign in the `AdminAllAssigned` screen. The `AssignTranslator` screen receives as parameters the `LanguageFrom, LanguageTo, SessionId, and the CancelledBy` attribute of the request in question from either of the previously mentioned screens. The `AssignTranslator` screen is a list format screen, and the way it loads the translator's into the state that is mapped to the `FlatList` is through the response of the Backend method `listTranslators`. The `listTranslators` Backend method takes as parameters the `LanguageFrom, LanguageTo, and CancelledBy` attributes and returns an array of all the users that speak these languages and are not in the CancelledBy array. This array of users is then mapped to the displayed `FlatList` The `FlatList` elements all contain a button called Assign. The Assign button is used to call on the `assignTranslator` Backend method, and takes as parameters the `SessionId` and the translator list object's email. The Backend then assigns the translator and upon completion the admin is alerted that the translator has been assigned and is navigated back to the screen from which they initially were in.

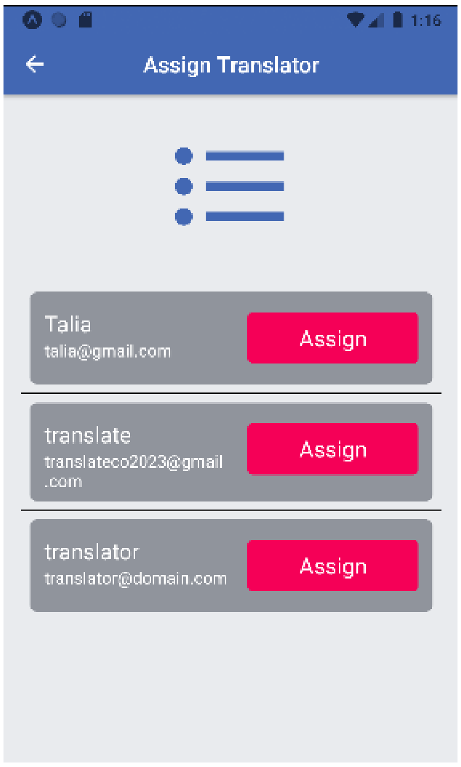


Figure 3.61: The TranslatorLanguages screen.

AddLanguage Screen

The `AddLanguage` screen is the screen responsible for adding new languages to the system. The screen is accessible from the admin Home screen. The format of the screen is a form with a text field and a button. The admin is then instructed to enter the name of the language to be added in the text field. Once the admin presses the add language button, a call is made to the Backend method `addLanguage` and takes as a parameter the entered language. After the language is successfully added, the admin is taken back to their Home screen. With this in mind, now any translator wishing to add this language to their list of language can simply find it in the language drop down menu in their `TranslatorLanguages` screen.

UserDetails Screen

The `UserDetails` screen is the screen responsible for showing the details of a user and is accessed from the `UserList` screen. The `UserDetails` screen receives as parameters the details of the user list item from the `UserList` screen for which the Details button had been pressed. The `UserDetails` screen then displays these details in a scrollable form format similar to the `ReportDetails` and `SessionDetails` screens. This screen does not have role based rendering as only the admin can access it.

AddUserHome Screen

The `AddUserHome` screen is the screen that the admin is navigated to upon pressing the Add User button on the Home screen. The `AddUserHome` screen only serves the purpose of having three buttons which upon pressing them turns the visibility state of a modal to true. The modal in question is the `AddUser` screen, which is a prop receiving screen that is initiated in the `AddUserHome` screen as an object and is not navigated to. The `AddUserHome` screen has three versions of the `AddUser` objects, which are one with a role prop of one, one with a role prop of two, and one with a role prop of three. The `AddUser` objects are initially set to have a visibility of false so as to not render them; however, upon pressing the button to access the screen it will be turned to true and the specific rendition of the screen will be rendered.

AddUser Screen

The `AddUser` screen is the screen that allows an admin to add a new user to the system. The `AddUser` screen has props based rendering, as the placeholder per the text fields are changed depending on the role of the user to be added. As mentioned above, the `AddUser` screen is created as an object in the `AddUserHome` screen and is set to be visible upon pressing the specific button relating to the role. Given that it was mapped as an object, that means that it renders upon navigation to the `AddUserHome AddUserHome` screen, but is made invisible until needed.
The `AddUser` screen has text fields to accommodate the number of attributes required to create a new user, and upon pressing the Add button the specified Backend method is called. The Backend methods are either`addClient, or addTranslator, or addAdmin`. Once the Backend outputs the response, the visibility of the `AddUser` screen is set to false and the admin is taken back to the `AddUserHome` screen.

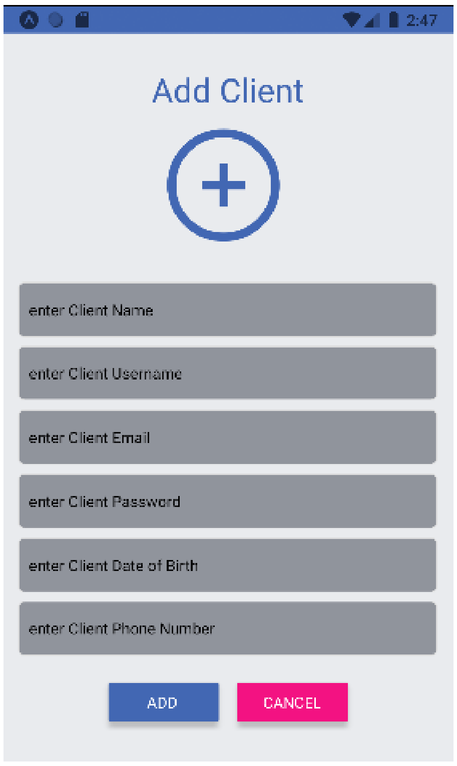
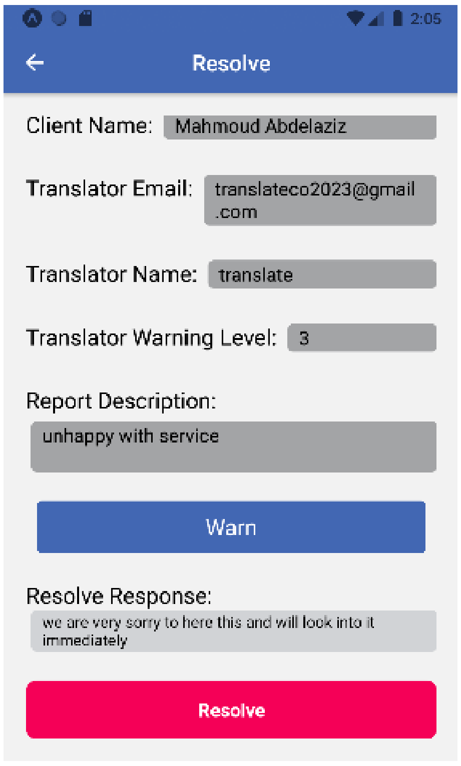


Figure 3.62: The AddUser screen with the client mode render.

ResolveReport Screen

The `ResolveReport` screen is the screen that is navigated to upon pressing on the Resolve button in either of the `AdminSessionUnresolved` or `AdminSystemUnresolved` screens. The `ResolveReport` screen is a scrollable form format screen, and it takes as parameters the report list items details upon navigation. Depending on the type of the report, a different screen will be rendered. If the type was a system report, only the client email, and report description will be rendered along with the admin's response text box. If it was a session report then all the fields would be rendered along with a button that allows the admin to increase the admin's warning status.

After the admin fills in the response text box and presses the Resolve button, the `RespondToReport` Backend method is called and given to it some of the report's details. The Backend method then would turn the `Resolved` attribute of the report to true, which allow it to be displayed in the resolved reports screens. The method also sends emails to the client and the translator.



Client Name: Mahmoud Abdelaziz

Translator Email: translateco2023@gmail.com

Translator Name: translate

Translator Warning Level: 3

Report Description: unhappy with service

Resolve Response: we are very sorry to here this and will look into it immediately

Figure 3.63: The ResolveReport screen with the Session type render.

3.3.13 Common Styles

The Common folder contains JavaScript files button and form, which contain stylesheets/styles which are commonly used by multiple pages. The way to use them is by simply importing them in the page needed and adding them to the tag that will take that style, such as a `View` tag or a `Button` tag or a `TextInput` tag. The way to do so is by using the style prop and passing to it the style needed. The form file is what we used mainly in the `Signup` and `Login` forms and they contain styles for inputs as well as error messages like for incorrect input format. The button file was used for the design of four different buttons which have four different color. The blue button represents normal actions done in the application. The red button represents critical actions such as Logout or cancel. The green button represents success actions like paying for a session, and approving, assigning, or starting a session. The grey button represents unpressable buttons.

3.3.14 Assets

The Assets folder contains the images which are used throughout the application such as the logo and trademark, the background, as well as the intro slider pictures used in the application. These images are located within the images folder in the assets folder, and the images outside are used for loading screen and the startup screen.

3.3.15 Miscellaneous

App.js File

The App.js file is the root file for all the navigation that takes place in the system. It contains a main Stack Navigator, which is created using the statement `createNativeStackNavigator()`, and it's first page is the Welcome Screen hence why it is always the first page to load upon startup of the application. The Welcome screen is referred to as a Stack Screen and the other Stack Screens in the Stack Navigator are Login, Signup and all screens that are not referenced in any of the navigation files mentioned above. The App.js file contains the root identification for all of the navigators in the system as well. Each navigator mentioned throughout the sections above has their own stack screen identification in the App.js file. This is the concept of Nesting Navigators [6], where for example the NavBar Navigator in the NavBar screen is nested into the Draw Navigator drawer screen called Tab. The Draw navigator is also created in the App.js as Draw, thus the Draw navigator is nested into a stack navigator and the Navbar navigator is nested in the drawer navigator. This is the reason why when navigating to Home, the navigation is using `navigation.navigate('Tab',{screen:'home'})`, where we first mention the stack screen name of the Tab Navigator then the Screen to navigate to within the Tab Navigator.
you

3.3.16 Testing

Postman API

Postman is an API testing tool which can be used to run requests to the API to affect the Database. An API is an application programming interface and it is basically the layer between the Backend and the Database which acts as a way to communicate requests to the Database from the Backend and it takes the form of HTTP request methods as in for example the authcontroller. However, without a Frontend to call on those HTTP requests, we cant virtually make any calls to the Database from the Backend alone. Postman solves this issue by using these HTTP Requests to make calls to the Database by acting as a testing Frontend for the application's Backend, and this is really useful in the testing phases of the Backend files if the Frontend has not yet been created. To see if changes were made to the Database collections, an HTTP GET method on that specific collection code be called from Postman and the results could be compared with the previous results, instead of the old way of doing it which would be by opening the collection on the MongoDB website or the MongoDB Atlas application to see if the collection had been updated or not, hence Postman eliminates the need to have to open the collection itself and allows for ease of communication with the database during the testing phase.

Chapter 4

Conclusion and Future Work

The purpose of this project is to provide a digital transformation for TranslateCo's way of business. The way that this was handled in our development was by providing an interface that has an ease of use for both Clients and Translators, allowing Clients to easily book a session whenever and wherever they are, as well as allowing Translators to accept incoming sessions in a mobile manner. This ease allows faster acceptance rates as well as faster assignment rates by the Admins since now they get updated lists on what translators are best suited for the session based on languages they are fluent in. This ensures high-quality translations and greater customer satisfaction. In conclusion, this digital translation platform provides an efficient and reliable solution for Clients looking for fast, accurate, and on time translations, as well as allowing Translators to receive more requests faster and in a more organized scheduled manner. With its advanced features, Clients can be sure that their needs will be met with the highest level of professionalism and expertise and Translators can relax knowing that their needs are met as well.

Through the Background chapter, we discussed all of the technologies used in the creation of this application. We discussed certain features as well of each technology such as the React Life Cycle hooks, which are key methodologies when it comes to the responsiveness of any React application. We also discussed where our data is stored, on MongoDB, as well as how it is communicated to from our Backend, which was using Express JS. Throughout the Implementation chapter, we thoroughly discussed every component that was created in the application, from startup screens all the way to role specific screens. Through the Implementation chapter, we have shown what each screen functionally does in our application as well as what Backend methods they communicate with. We have also gone through the Backend methodologies and their controller classes and discussed the choices made when creating them as well as the thought process behind most of them. Through the previous chapters, we have managed to successfully achieve our aim of providing a digital transformation to the company, and created the perfect application.

In the future, The application should reveive deployment for its Backend server on any platform hosting service, as well as be uploaded onto the Google Play store. Furthur work could also be added to the application to further improve responsiveness, as well receive various updates to further enhance the Frontend style.

List of Figures

Bibliography

[1] React Development. Built-in react hooks. https://react.dev/reference/react, 2023.

[2] Axios Docs. Axios docs. https://axios-http.com/docs/intro, 2023.

[3] Expo Docs. Expo docs. https://docs.expo.dev/get-started/installation/, 2023.

[4] MongoDB Docs. Mongodb docs. https://www.mongodb.com/docs/, 2023.

[5] React Navigation Docs. Bottom tabs navigator. https://reactnavigation.org/docs/bottom-tab-navigator/, 2023.

[6] React Navigation Docs. Nesting navigators. https://reactnavigation.org/docs/nesting-navigators, 2023.

[7] Radek Fabisiak. React native lifecycle methods with hooks guide. https://www.blog.duomly.com/react-native-lifecycle-methods-with-hooks/, March 2022.

[8] Gilshaan Jabbar. React native hooks how to use usestate and useeffect. https://gilshaan.medium.com/react-native-hooks-how-to-use-usestate-and-useeffect-3a10fd3e760c, April 2020.

[9] Inc. Mongoose. Mongoose docs. https://mongoosejs.com/docs/, 2023.

[10] NPMJS. @react-native-community/datetimepicker. https://www.npmjs.com/package/@react-native-community/datetimepicker, 2023.

[11] Inc. Postman. Postman learning docs. https://learning.postman.com/docs/introduction/overview/, 2023.

[12] Andris Reinman. Nodemailer. https://nodemailer.com/about/, 2010.

[13] Stripe. Stripe docs. https://stripe.com/docs, 2023.