



**Kazakh-British Technical University**  
**School of Information Technologies and Engineering**

**Assignment #2**  
Mobile programming  
Basic pages of Instagram

Prepared by: Maratuly T.  
Checked by: Serek A.

**Almaty, 2024**

## CONTENT

Introduction .....	3
1. Project setup .....	4
2. Layouts .....	6
2.1 Home feed page .....	8
2.2 Search page .....	10
2.3 Profile page .....	13
2.4 Add post page .....	16
2.5 Notifications page .....	21
3. React Native development features .....	23
3.1 Navigation .....	23
3.2 User interaction .....	25
3.3 Data handling .....	25
Conclusion .....	26

## INTRODUCTION

This work is responsible for creating a real application with multi pages. We need to study how navigate between different pages, use navigation and work with forms. Ubuntu was an operational system (OS) which we used to cover this lab. Pay attention to a user's username (here 'nekofetishist' is a username of Maratuly Temirbolat with ID 23MD0409). Instead of a virtual emulator, real android smartphone was used (Xiaomi Redmi Note 12 Pro). The whole project was done using React Native which key difference that it can be applied to create applications for iOS, android as well as for the web interfaces.

## 1. Project setup

Before the beginning of the lab's process pay attention that username 'nekofetishist' belongs to Maratuly Temirbolat (ID 23MD0409). Figure 1.1 illustrates who is the owner of the account which will be used to cover the work.

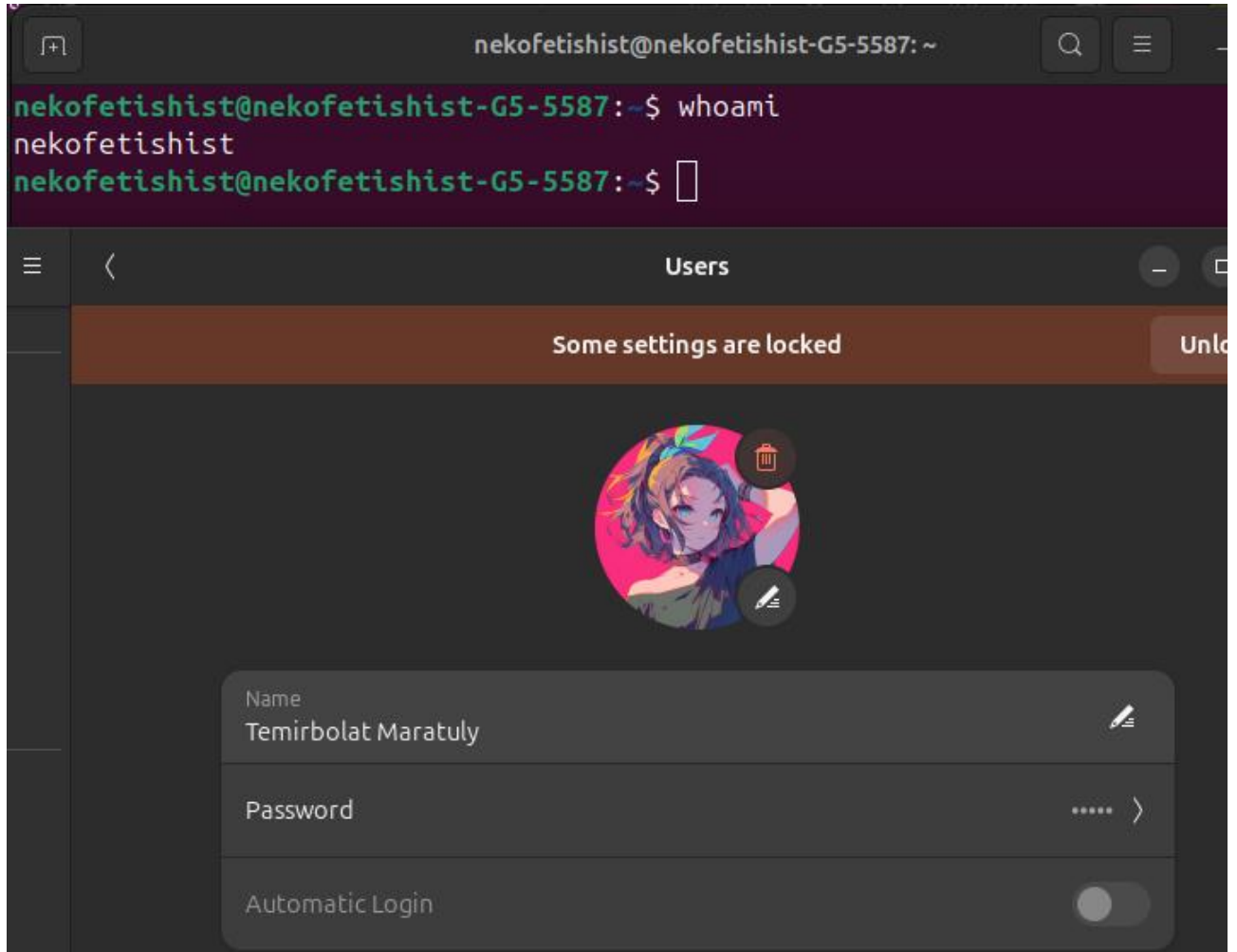


Figure 1.1 - Username and account's owner credentials information

Thus, according to a figure, the command 'whoami' from the Ubuntu command line shows the current user, moreover, this username is used in the whole path (marked with green letters).

Now, let's look at the project's setup. The project was installed using expo. Expo is a production-grade React Native Framework. It provides developer tooling that makes developing apps easier, such as file-based routing, a standard library of native modules and much more. Another advantage that it is free and open source, it lets the developer to run an application in a terminal, then if a developer does not have an installed emulator for android or iOS, the generated QR code could be scanned and been connected to the same network we can open the started project in expo mobile application. To create a sample application using expo run the command "npx create-expo-app@latest". The crucial factor that by default expo or standard react native use

TypeScript to generate a code which is also a great approach as we use data types implicitly. Additionally, to use emulators, we need to install Android Studio for android devices and Xcode for iOS devices respectively. Let's now examine the generated working tree of a project which is illustrated in Figure 1.2.

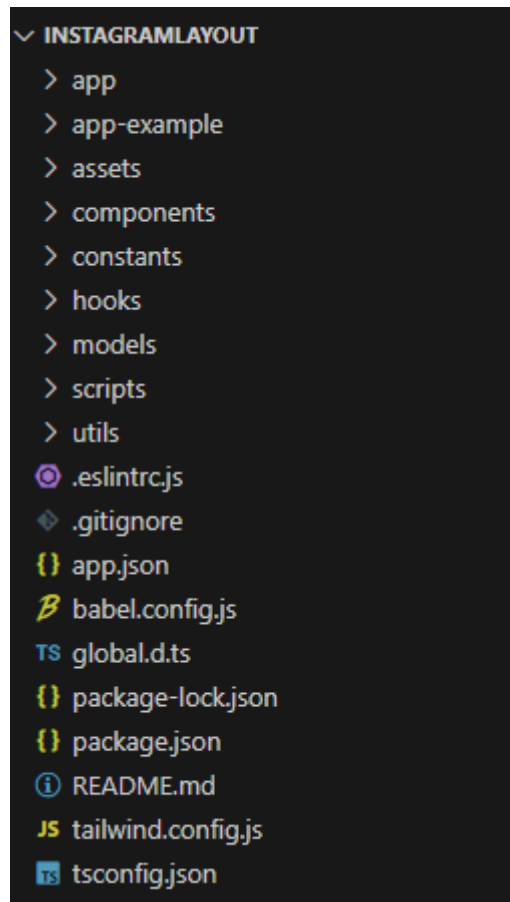


Figure 1.2 – Working tree of the Instagram layout project

This React Native project, built with **Expo**, contains several important folders. The **"app/"** directory holds the code for screens and navigation, while **"app-example/"** is generated by default with Expo and provides example implementations. Reusable UI components, such as buttons or headers, are located in **"components/"**. The **"assets/"** folder stores static files like images and fonts. **"hooks/"** contains custom React logic for managing state and behavior; it is also generated by default with Expo as a starting point for developers. **"models/"** defines data structures used across the app, **"utils/"** organizes common utility functions, and **"scripts/"** holds automation tools or setup scripts, another default folder provided by Expo for demonstration purposes.

Several key configuration files are included in the project. **"app.json"** defines the app's configuration, and **"package.json"** lists its dependencies. **"babel.config.js"** allows the app to use modern JavaScript, and **"tsconfig.json"** provides settings for TypeScript. **".eslintrc.js"** enforces code style rules, and **"tailwind.config.js"** sets up Tailwind CSS for styling. These elements, combined with Expo, make the development

process streamlined and manageable. Then, each folder will be unwrapped to see its content along with the code, but not in this section as it would be unclear. To run an application in browser use the command: “npm run web”. You will obtain about the same results in Figure 1.3.

```
[17:13:55] Starting project at /Users/work/Desktop/ExpoTester
[17:13:55] Expo DevTools is running at http://localhost:19002
[17:13:55] Opening DevTools in the browser... (press shift-d to disable)
[17:14:00] Starting Metro Bundler on port 19001.
[17:14:01] Tunnel ready.

exp://192.168.1.178:19000
```



```
To run the app with live reloading, choose one of:
• Scan the QR code above with the Expo app (Android) or the Camera app (iOS).
• Press a for Android emulator, or i for iOS simulator.
• Press e to send a link to your phone with email/SMS.
• Press s to sign in and enable more options.

Press ? to show a list of all available commands.
Logs for your project will appear below. Press Ctrl+C to exit.
```

Figure 1.3 – Result of running web version of an application

The provided QR code is used here to open the application from your smartphones (but you must be in the same network). Additionally, web version is visible on the localhost with specified port (here 19002).

## 2. Layouts

This section is the most important and responsible for developing main pages of our Instagram like application. It consists of the following pages: home, profile, search, add new post, notifications pages. All these pages have a navigation bar and header, so, we can move from one page to another. The subsections below contain information about each page separately with corresponding screenshots to see the results. The results will be shown only for android devices. All the pages for our application are

hold in “app” folder which is in the root of the project. Figure 1.4 demonstrates the unwrapped “app” folder with project’s pages.

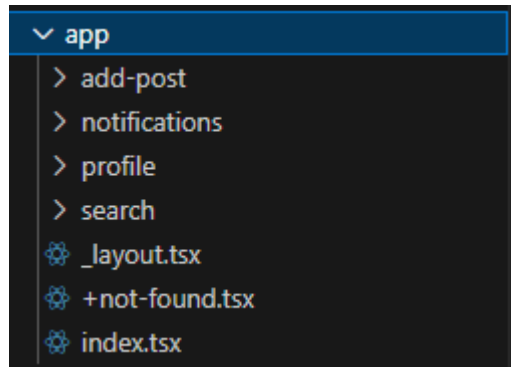
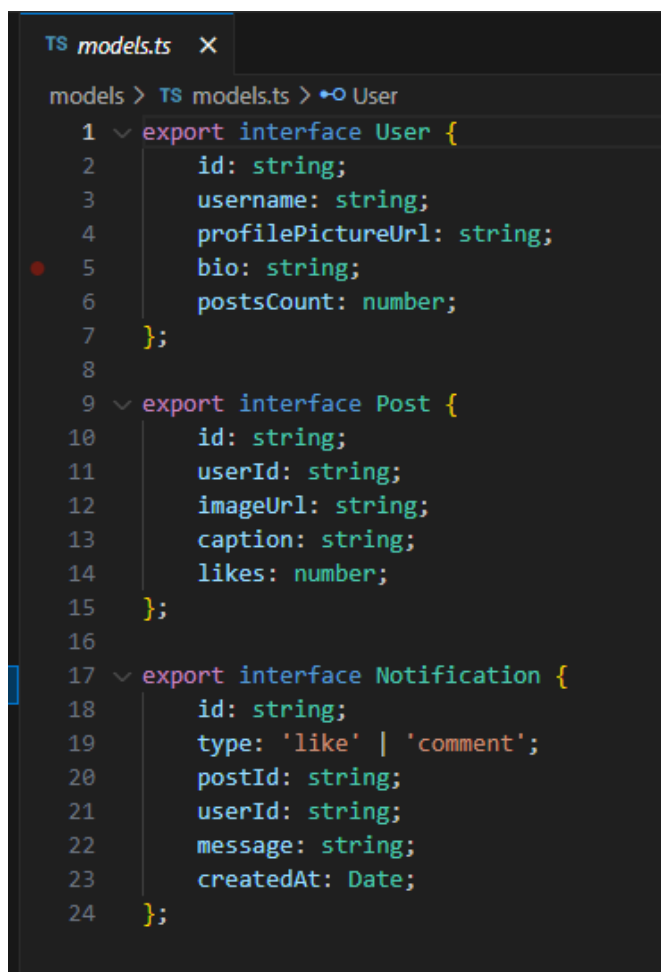


Figure 1.4 – Structure of main application pages

The app folder consists of the following folders: “add-post”, “notifications”, “profile”, “search”. There are present such files: “\_layout.tsx”, “+not-found.tsx”, and “index.tsx”. It is quite necessary to show the models of the project. Figure 1.5 represents the code of models which used in the project.



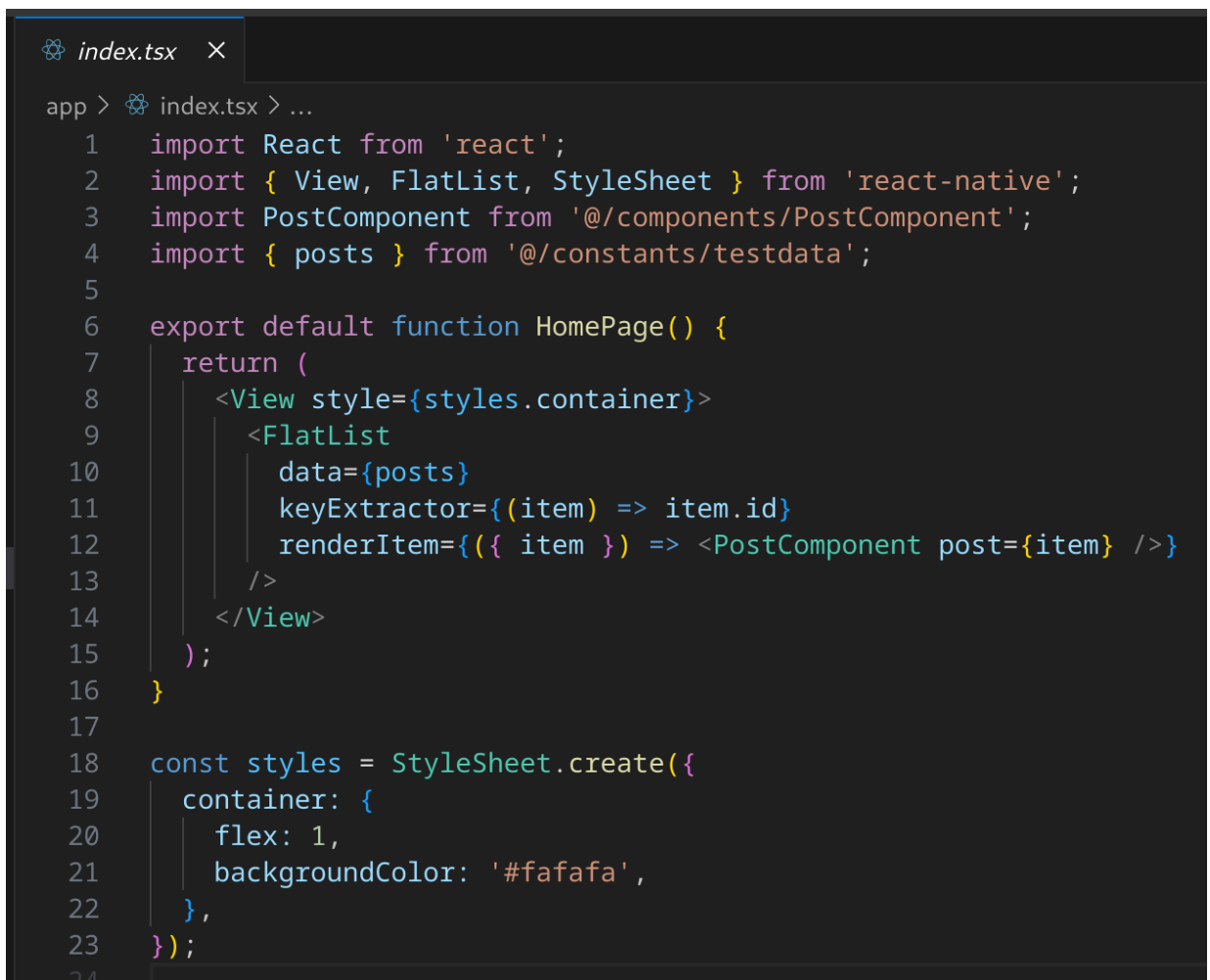
```
TS models.ts x
models > TS models.ts > User
1 export interface User {
2   id: string;
3   username: string;
4   profilePictureUrl: string;
5   bio: string;
6   postsCount: number;
7 };
8
9 export interface Post {
10  id: string;
11  userId: string;
12  imageUrl: string;
13  caption: string;
14  likes: number;
15 };
16
17 export interface Notification {
18  id: string;
19  type: 'like' | 'comment';
20  postId: string;
21  userId: string;
22  message: string;
23  createdAt: Date;
24 };
```

Figure 1.5 – Used models of the project

We have “User”, “Post” and “Notification” models in “models.ts” file (“models” folder). “User” model has the following fields: “id” (integer number), “username” (string), “profilePictureUrl” (url string to a photo), “bio” (short string description) and “postsCount” (number of user’s posts). “Post” model consists of the next fields: “id” (integer number), “userId” (integer of the post’s owner), “imageUrl” (url string to a photo), “caption” (short post’s description) and “likes” (integer number of likes). “Notification” model has: “id” (integer number), “type” (type of a notification which is new like or new comment), “postId” (integer identifier of a post under which notification is left), “userId” (integer identifier of a user who left a comment), “message” (content of a notification), “createdAt” (date when the notification was created).

## 2.1. Home feed page

The home page is a page that is responsible for displaying a list of posts in scrolled vertical view. “Index.tsx” is navigated as main page under each folder if no any other files are specified. Figure 1.6 contains the main code of the “index.tsx” page which is a home page.



```
index.tsx X
app > index.tsx > ...
1  import React from 'react';
2  import { View, FlatList, StyleSheet } from 'react-native';
3  import PostComponent from '@components/PostComponent';
4  import { posts } from '@constants/testdata';
5
6  export default function HomePage() {
7    return (
8      <View style={styles.container}>
9        <FlatList
10         data={posts}
11         keyExtractor={(item) => item.id}
12         renderItem={({ item }) => <PostComponent post={item} /> }
13       />
14     </View>
15   );
16 }
17
18 const styles = StyleSheet.create({
19   container: {
20     flex: 1,
21     backgroundColor: '#fafafa',
22   },
23 });
24
```

Figure 1.6 – Code of a home page



The page looks so small because it consists of the components to reuse the view logic and isolate it. You can think about them as functions (reused code parts) but in a layout. It imports necessary modules from React, React Native, and two local files: “PostComponent” from the components folder and some sample data (posts) from the constants folder. The component uses a “FlatList” to display a list of posts by mapping each post to a “PostComponent”. The “keyExtractor” ensures each post has a unique key, based on its id. The “renderItem” function renders each item in the list as a “PostComponent” with the post data passed as a prop. The View wrapper applies basic styling, with a background color of “#fafafa” using a style object created by “StyleSheet.create”. The view of the home page could be seen in Figure 1.7.



Figure 1.7 – Home feed page on android

It can be clearly seen that application home page view looks almost the same as in Instagram. We can also scroll the page vertically to see other posts of other people. There is not only one user, but plenty of them. The page’s header represents at what page we are currently located. Figure 1.8 gives proof that we can scroll through the posts.



Figure 1.8 – Scroll through the posts on home page

So, we have can scroll through the posts, and our “FlatList” component loads not all the data immediately but in chunks (some part of them) to balance the load and system. “PostComponent” could be found in “components” folder.

## 2.2. Search page

The current page is used to search users by their usernames. As a searching parameter we have text input where we input username. Let’s firstly take a look where the “Search” page is located (Figure 1.9).

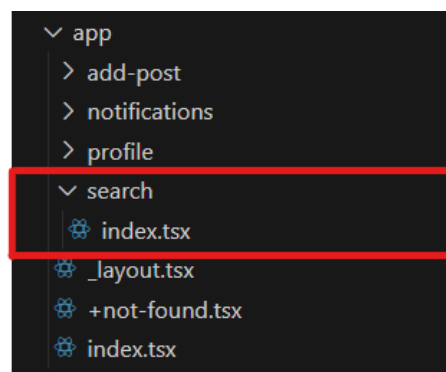
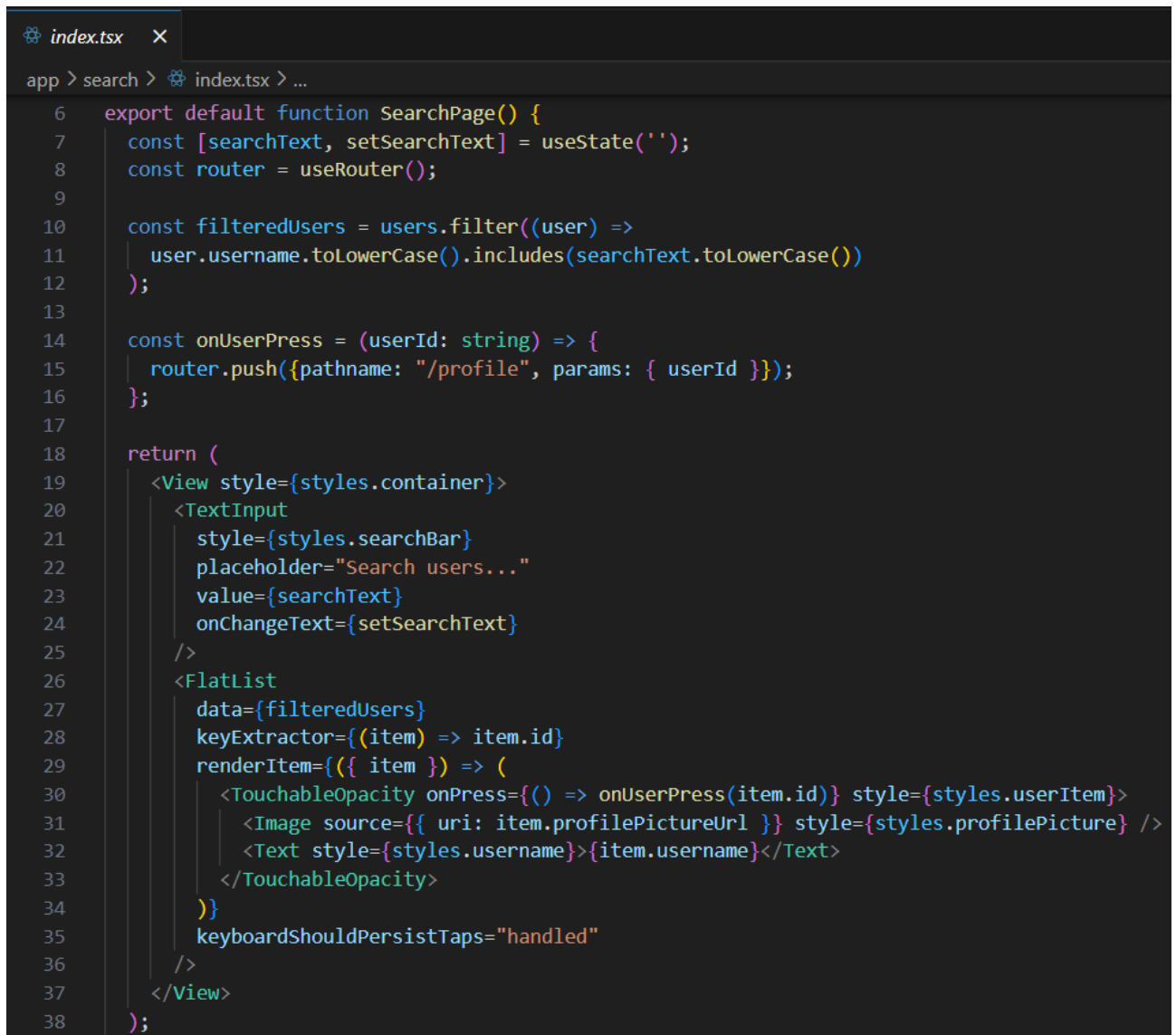


Figure 1.9 – Search page location

Our search page file is “index.tsx”, however, located in the “search” folder. Detailed description about navigation will be explained in “Navigation” section. Let’s just remember that this file is loaded as we want to move to search page. Figure 1.10 resembles the code of a page.



```
index.tsx X
app > search > index.tsx > ...
6  export default function SearchPage() {
7    const [searchText, setSearchText] = useState('');
8    const router = useRouter();
9
10   const filteredUsers = users.filter((user) =>
11     user.username.toLowerCase().includes(searchText.toLowerCase())
12   );
13
14   const onUserPress = (userId: string) => {
15     router.push({pathname: "/profile", params: { userId }});
16   };
17
18   return (
19     <View style={styles.container}>
20       <TextInput
21         style={styles.searchBar}
22         placeholder="Search users..."
23         value={searchText}
24         onChangeText={setSearchText}
25       />
26       <FlatList
27         data={filteredUsers}
28         keyExtractor={(item) => item.id}
29         renderItem={({ item }) => (
30           <TouchableOpacity onPress={() => onUserPress(item.id)} style={styles.userItem}>
31             <Image source={{ uri: item.profilePictureUrl }} style={styles.profilePicture} />
32             <Text style={styles.username}>{item.username}</Text>
33           </TouchableOpacity>
34         )}
35         keyboardShouldPersistTaps="handled"
36       />
37     </View>
38   );
```

Figure 1.10 – Code of the search page

This code creates a simple search screen page. It lets users type in a search box to look for other users by username. The search input is managed using “useState”, which keeps track of what the user types. As the user types, the list of users is filtered to show only those whose usernames match the search text (ignoring case).

Each matching user is displayed in a list. Every item in the list shows the user's profile picture and username. The items are wrapped in “TouchableOpacity”, which makes them clickable. When you tap on a user, the app navigates to that user’s profile page, passing their “userId” as a parameter to open the right profile. The search input is built with “TextInput”. It updates the search results as you type, keeping everything

smooth and in real time. Let's see the view of the search page before typing any text (Figure 1.11).

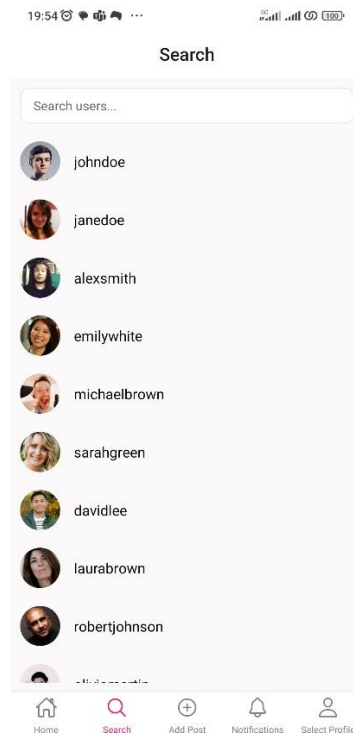


Figure 1.11 – Search page listing users before filtering

Then, we input some text and see the changed list (Figure 1.12)

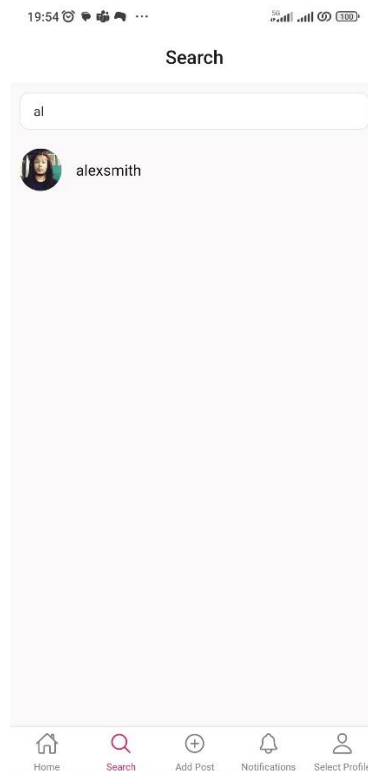


Figure 1.12 – Search page listing users after filtering

It can be clearly seen that we can easily search for the users by just typing a username which we want to find. Additionally, we can click on any user to move to its detailed page. The next section is about detailed users' pages.

### 2.3. Profile page

The current section is about user's profile page. It consists of 2 files or pages. Figure 1.13 represents its location in the project tree.

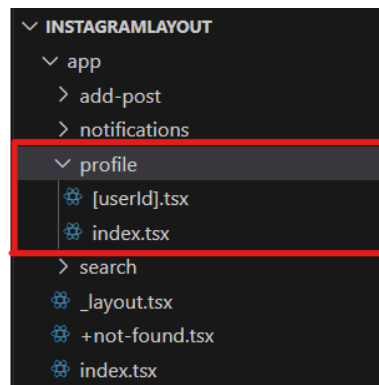


Figure 1.13 – Profile page location

There are 2 files “[userId].tsx” and “index.tsx”. As we move to a profile page, initially we come to the “index.tsx” file. Its code is visible in Figure 1.14.

```
index.tsx x
app > profile > index.tsx > ...
12
13 export default function ProfileSelectionPage() {
14   const router = useRouter();
15
16   const onUserPress = (userId: string) => {
17     router.push(`/profile/${userId}`);
18   };
19
20   return (
21     <View style={styles.container}>
22       <FlatList
23         data={users}
24         keyExtractor={(item) => item.id}
25         renderItem={({ item }) => (
26           <TouchableOpacity
27             onPress={() => onUserPress(item.id)}
28             style={styles.userItem}
29           >
30             <Image
31               source={{ uri: item.profilePictureUrl }}
32               style={styles.profilePicture}
33             />
34             <Text style={styles.username}>{item.username}</Text>
35           </TouchableOpacity>
36         )}
37         keyboardShouldPersistTaps="handled"
38       />
39     </View>
40   );
41 }
```

Figure 1.14 – Code of the initial profile page

It displays a list of users, and when you tap on a user, it takes you to their profile page. When a user is clicked, the “onUserPress” function is called, which navigates to the profile of that specific user by adding their “userId” to the URL. As a result, we get the following page view (Figure 1.15).

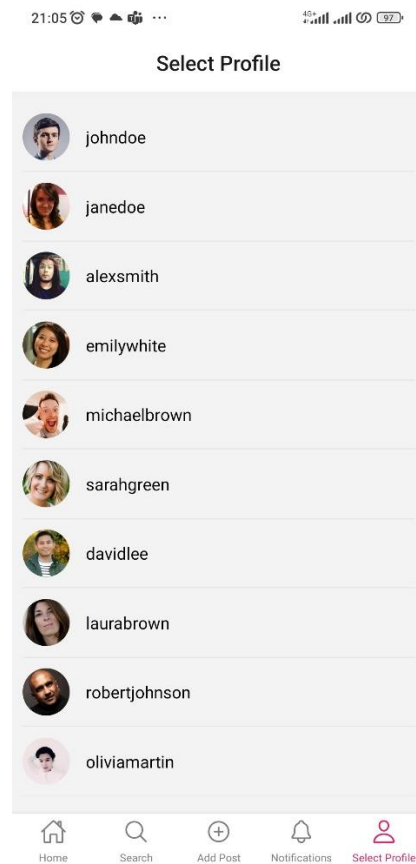


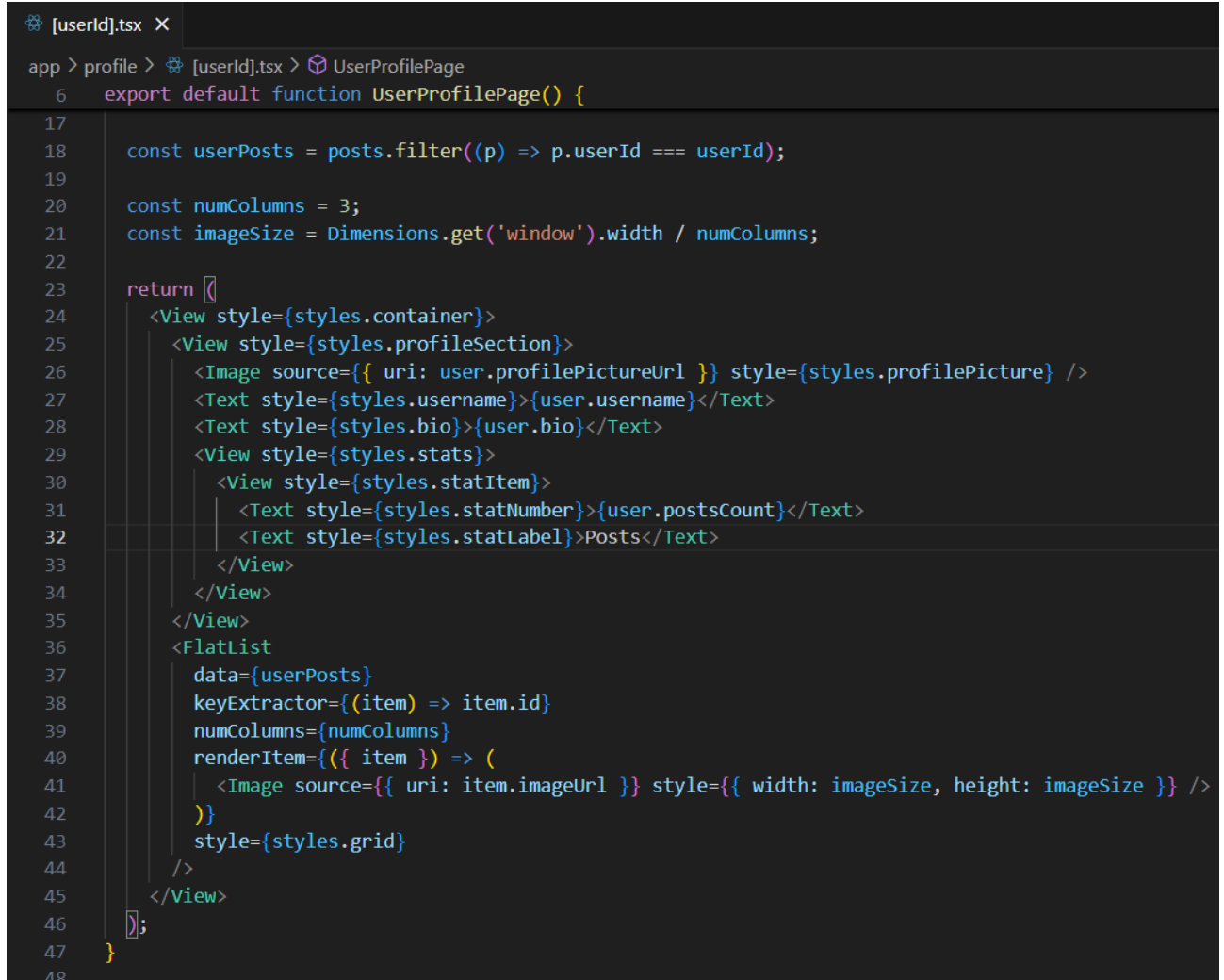
Figure 1.15 – Select Profile page

Here we choose one of the users from the list whose profile we want to see (or from the search page). After selecting a user from the list we come to a “[userId.tsx]” page where id is used as a parameter. The search code is shown in Figure 1.16.

```
[userId].tsx X
app > profile > [userId].tsx > UserProfilePage
6 export default function UserProfilePage() {
7   const { userId } = useLocalSearchParams<{ userId: string }>();
8   const user = users.find((u) => u.id === userId);
9
10  if (!user) {
11    return (
12      <View style={styles.container}>
13        <Text>User not found</Text>
14      </View>
15    );
16  }
```

Figure 1.16 – Code for searching a user

Here we apply “useLocalSearchParams” hook function to extract “userId” from the url. Moreover, we immediately start looking for the user by the id and as we find we return it. If there is no found user, we return “User not found” text as a response to a user. Figure 1.17 shows the continuation of the code from the “[userId].tsx” file which is triggered as we find the user correctly.



```
[userId].tsx X
app > profile > [userId].tsx > UserProfilePage
6  export default function UserProfilePage() {
17
18    const userPosts = posts.filter((p) => p.userId === userId);
19
20    const numColumns = 3;
21    const imageSize = Dimensions.get('window').width / numColumns;
22
23    return (
24      <View style={styles.container}>
25        <View style={styles.profileSection}>
26          <Image source={{ uri: user.profilePictureUrl }} style={styles.profilePicture} />
27          <Text style={styles.username}>{user.username}</Text>
28          <Text style={styles.bio}>{user.bio}</Text>
29          <View style={styles.stats}>
30            <View style={styles.statItem}>
31              <Text style={styles.statNumber}>{user.postsCount}</Text>
32              <Text style={styles.statLabel}>Posts</Text>
33            </View>
34          </View>
35        </View>
36        <FlatList
37          data={userPosts}
38          keyExtractor={({item}) => item.id}
39          numColumns={numColumns}
40          renderItem={({ item }) => (
41            <Image source={{ uri: item.imageUrl }} style={{ width: imageSize, height: imageSize }} />
42          )}
43          style={styles.grid}
44        />
45      </View>
46    );
47  }
48
```

Figure 1.17 – Continuation of a user’s profile page

This code creates a user profile page that shows the user’s profile picture, username, bio, and total number of posts. It filters through the posts to display only those made by the current user. This way, the profile shows only relevant content.

The posts are arranged in a grid with 3 columns to keep everything organized and easy to browse. The size of each image is automatically adjusted to fit the screen’s width, making sure all posts align nicely. A “FlatList” is used to show the posts efficiently, even if there are a lot of them, ensuring smooth scrolling.

At the top, the user’s personal details are displayed, followed by their posts in a neat, scrollable grid. This layout provides a simple and clear way to view both the user’s info and their content in one place, making it easy to explore the profile. The view of this page is shown in Figure 1.18.

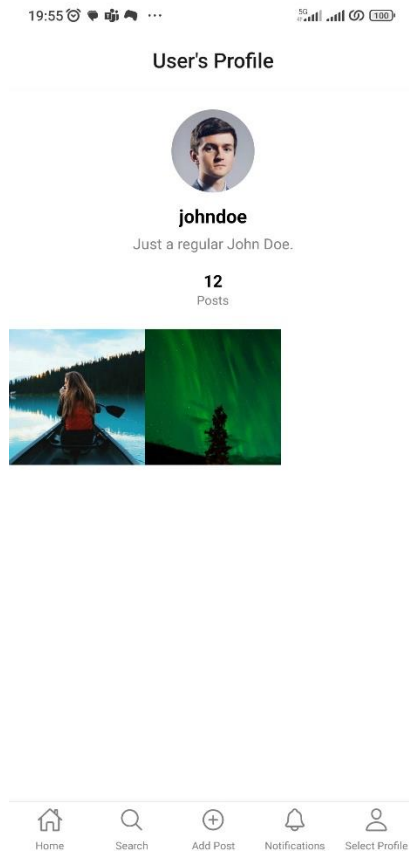


Figure 1.18 – Detailed user's profile page view

Here we see the page's name (User's profile), the user's main photo, username, biography, number of posts and grid of posts separated by 3 columns per row.

## 2.4. Add post page

As we finish looking at the pages where we can only get data at see it, let's move to a page where we upload new posts. Figure 1.19 demonstrates the location of this page in a working tree explorer.

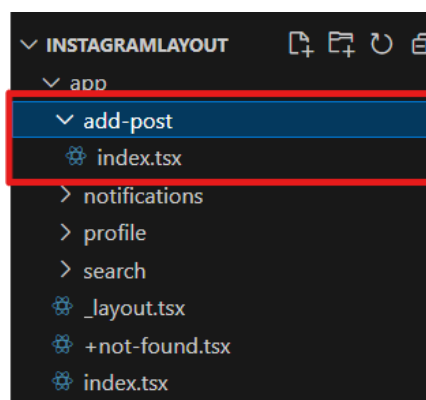


Figure 1.19 – Location of a page to upload new posts



The code of this page is shown in Figure 1.20. As we remember it is still “index.tsx” file as it is an initial file under each folder to be loaded first.

```
export default function AddPostPage() {  
  return (  
    <ScrollView contentContainerStyle={styles.container}>  
      <Text style={styles.label}>Select User</Text>  
      <View style={styles.pickerContainer}>  
        <Picker  
          selectedValue={selectedUserId}  
          onValueChange={(itemValue) => setSelectedUserId(itemValue)}  
          style={styles.picker}  
        >  
          {users.map((user) => (  
            <Picker.Item key={user.id} label={user.username} value={user.id} />  
          ))}  
        </Picker>  
      </View>  
  
      <TextInput  
        style={styles.input}  
        placeholder="Image URL"  
        value={imageUrl}  
        onChangeText={setImageUrl}  
      />  
  
      <TextInput  
        style={[styles.input, styles.captionInput]}  
        placeholder="Caption"  
        value={caption}  
        onChangeText={setCaption}  
        multiline  
      />  
  
      <Button title="Upload Post" onPress={onAddPost} />  
      {imageUrl ? (  
        <Image source={{ uri: imageUrl }} style={styles.previewImage} />  
      ) : null}  
    </ScrollView>  
  );  
}
```

Figure 1.20 – Code for uploading a new Post

This code creates a page where users can upload a new post. It lets the user select which account they are posting from, enter the image URL, and write a caption for the post.

At the top, a “Picker” component displays a list of users, allowing the user to choose which account will make the post. The selected user’s ID is stored and updated when the user changes their selection. Below that, two “TextInput” fields let the user enter the image URL and a caption. The caption field supports multiple lines for longer text. When the user fills in the details, they can click the “Upload Post” button to submit the post. If an image URL is entered, a preview of the image will appear below the form. The entire page is wrapped in a “ScrollView” to make sure the content is scrollable, ensuring a smooth user experience on smaller screens. This design makes it easy for users to fill in all necessary details and upload a post quickly and efficiently.

The logic for uploading a new post as inputs are filled is represented in Figure 1.21.

```

export default function AddPostPage() {
  const [imageUrl, setImageUrl] = useState('');
  const [caption, setCaption] = useState('');
  const [selectedUserId, setSelectedUserId] = useState(users[0].id);

  const router = useRouter();

  const onAddPost = () => {
    if (!imageUrl || !caption) {
      Alert.alert('Error', 'Please fill in all fields');
      return;
    }

    const newPost: Post = {
      id: uuid.v4().toString(),
      userId: selectedUserId,
      imageUrl,
      caption,
      likes: 0,
    };

    posts.unshift(newPost);
    setImageUrl('');
    setCaption('');
    Alert.alert('Success', 'Post added successfully!', [
      {
        text: 'OK',
        onPress: () => router.push('/'),
      },
    ]);
  };
};

```

Figure 1.21 – Logic code for uploading a new post

This code is part of the "Add Post" page and handles the logic for uploading a new post. It uses “useState” to keep track of the image URL, caption, and the selected user. By default, the selected user is the first one from the user list.

The “onAddPost” function is triggered when the user clicks the "Upload Post" button. It first checks if both the image URL and caption fields are filled. If not, it shows an alert asking the user to complete the form. If all fields are filled, a new post object is created, including the user ID, image URL, caption, and setting the likes count to 0.

The new post is added to the beginning of the posts list using unshift, which ensures the most recent post appears first. After adding the post, the form fields are cleared, and a success message is displayed using an alert. When the user presses "OK" in the alert, they are redirected back to the home screen using “router.push('/')”. This approach ensures the post is properly created, added, and the user is smoothly returned to the main page. Figure 1.22 shows the view for uploading new post before filling information.

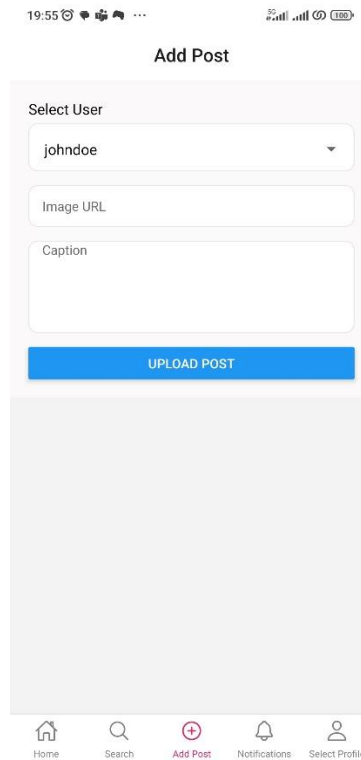


Figure 1.22 – Page for uploading new post before filling information

Figure 1.23 demonstrates the view of the page after filling information.

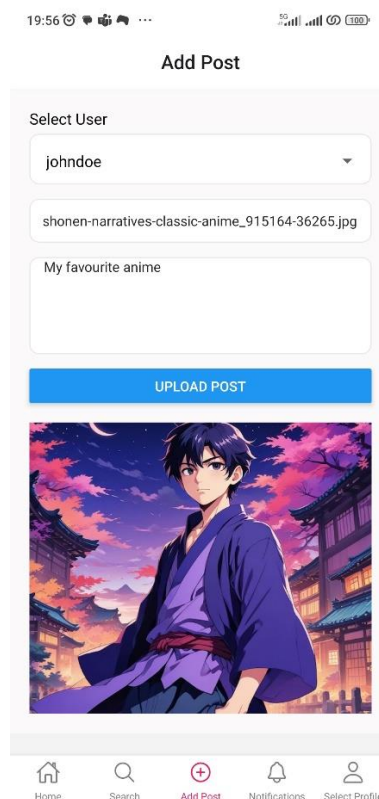


Figure 1.23 – Page for uploading information after filling information

After filling information, we can press the button “Upload Post” and if it is successful we obtain a popup message. Figure 1.24 demonstrates this behavior.

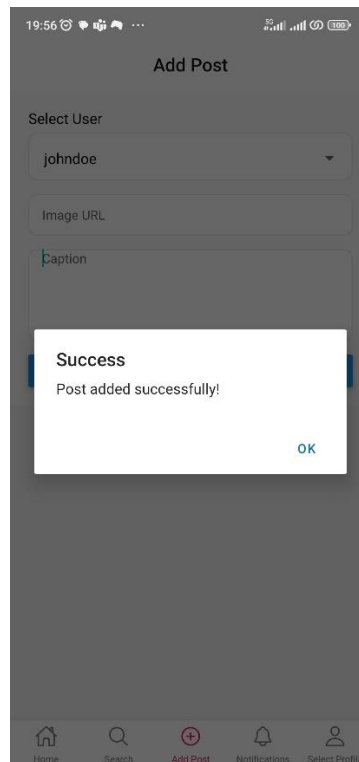


Figure 1.24 – Successful post uploading popup

After moving to a home page, we can see that new post appeared (Figure 1.25).



Figure 1.25 – New uploaded post on a home page

If we open a profile page of the user with username “johndoe” we can see that post is present (Figure 1.26).

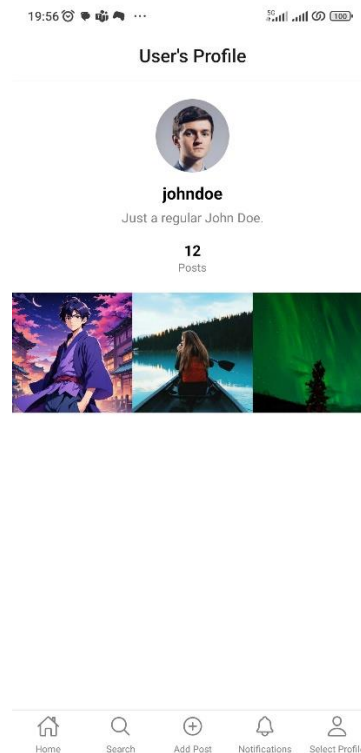


Figure 1.26 – View of a profile page with added new post

So, that means the page for adding a new post works as expected.

## 2.5. Notifications page

The “notification page” section is the last section about main pages as we covered all of them. The Notification page is a page where we can view all the notifications which the current user has. The notifications are related to likes or new comments. Figure 1.27 the location of the page in the file explorer.

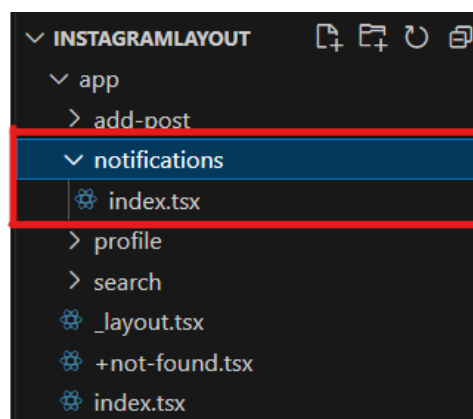
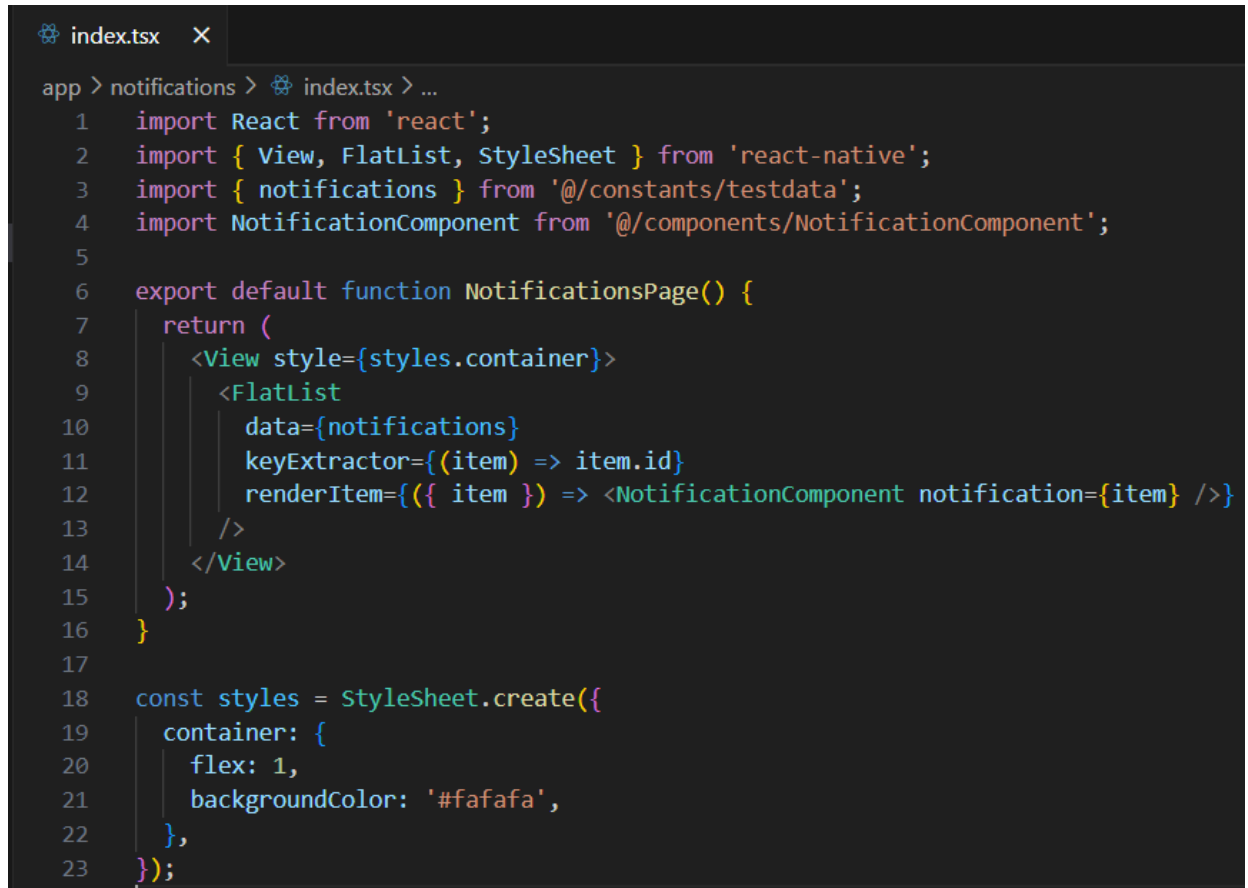


Figure 1.27 – Location of the page for notifications

As it was expected all the code about notifications page under the URL “https://hostname/notifications/” we come to the “index.tsx” page. Figure 1.28 shares the code about the page.



```
index.tsx X
app > notifications > index.tsx > ...
1  import React from 'react';
2  import { View, FlatList, StyleSheet } from 'react-native';
3  import { notifications } from '@/constants/testdata';
4  import NotificationComponent from '@/components/NotificationComponent';
5
6  export default function NotificationsPage() {
7    return (
8      <View style={styles.container}>
9        <FlatList
10          data={notifications}
11          keyExtractor={(item) => item.id}
12          renderItem={({ item }) => <NotificationComponent notification={item} />}
13        />
14      </View>
15    );
16  }
17
18  const styles = StyleSheet.create({
19    container: {
20      flex: 1,
21      backgroundColor: '#fafafa',
22    },
23  });
```

Figure 1.28 – Notification’s page code

This code creates a notifications page that shows a list of notifications to the user. It imports a list of notifications from the “testdata” file and displays them using a “FlatList”. Each notification is rendered with a custom component called “NotificationComponent”.

The “FlatList” takes the notifications as data and uses a “keyExtractor” to assign a unique key to each notification, ensuring smooth rendering. For every item in the list, the “renderItem” function passes the notification data to the “NotificationComponent” to display it properly. The component is in the “components” folder.

The whole list is wrapped in a View with some simple styling. The background color is set to “#fafafa” using the “StyleSheet”, and the container takes up the full screen using “flex” attribute style set to 1. This approach ensures the notifications are displayed clearly in a scrollable list, with each item shown in a structured way using the custom component. Let’s look at the result of the code considering Figure 1.29.

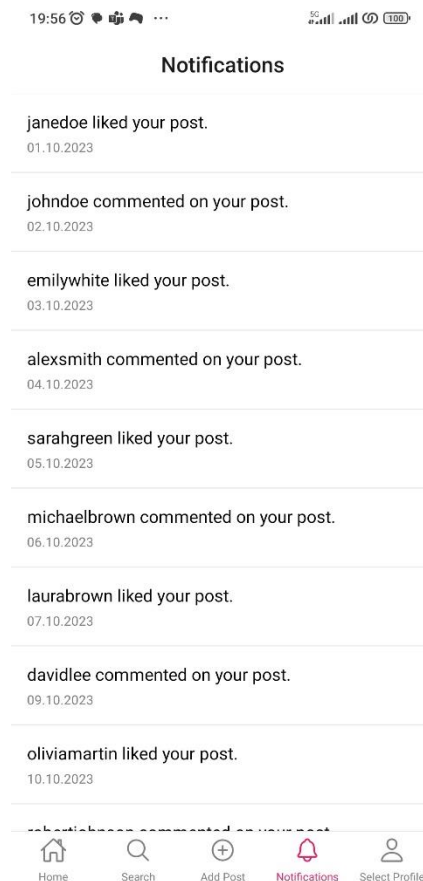


Figure 1.29 – Notification page view

According to a figure, we have a scrollable list of items where each item is a notification. Here notifications were created randomly depending on the type (comment or like) of a notification.

### 3. React Native development features

This chapter is about how user interaction, navigation and data handling implemented in the react native, but in scope of this project only.

#### 3.1 Navigation

In modern React Native apps, navigation is typically handled using the React Navigation library. It supports several types of navigation patterns, including stack navigation, tab navigation, drawer navigation, and deep linking. In the current project we used tab navigation. Tab navigation creates a view in the bottom of the screen, for example, from Figure 1.29 where you can easily navigate through your pages been at any of them. So, tab navigation approach provides a tab bar at the bottom to switch between main sections of the app. Let's examine the code of the tab navigation implementation. However, firstly understand what file it is located. Figure 1.30

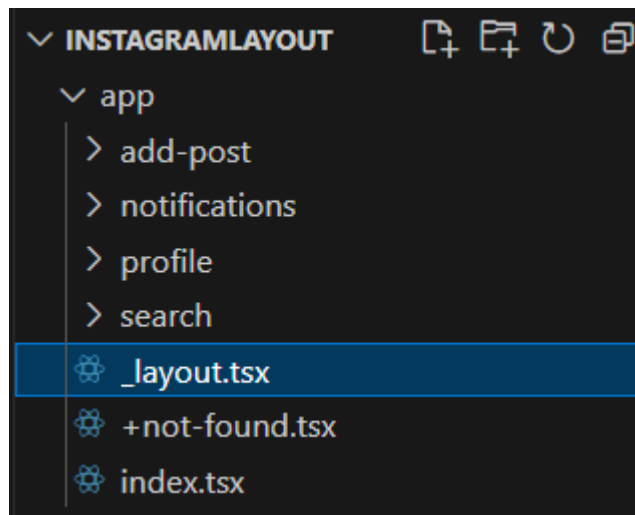


Figure 1.30 – Location of the router’s configuration for each tab

Thus, the location of the file is in the “\_layout.tsx” which was not explained before. The name of the file is constant to let React Native read it and process automatically. Basically, this file is responsible for pages’ layout. Now, consider the code from Figure 1.31.

```

_layout.tsx x
app > _layout.tsx > RootLayout
1  import { Tabs } from 'expo-router';
2  import { Ionicons } from '@expo/vector-icons';
3
4  export default function RootLayout() {
5    return (
6      <Tabs
7        screenOptions={{
8          tabBarActiveTintColor: '#e91e63',
9          tabBarInactiveTintColor: 'gray',
10         headerTitleAlign: 'center',
11       }}
12      >
13        <Tabs.Screen
14          name="index"
15          options={{
16            title: 'Home',
17            tabBarIcon: ({ color, size }) => (
18              <Ionicons name="home-outline" color={color} size={size} />
19            ),
20          }}
21        />
22        <Tabs.Screen
23          name="search/index"
24          options={{
25            title: 'Search',
26            tabBarIcon: ({ color, size }) => (
27              <Ionicons name="search-outline" color={color} size={size} />
28            ),
29          }}
30        />
31      </Tabs>
32    );
33  }

```

Figure 1.31 – Configuration of the layout file.



By default, React Native via expo framework has its own page handling router. It works with such approach that if you want to go to the “https://hostname/contacts” page you need to create a file in “app” directory “contacts.tsx” or create a folder “contacts” and create “index.tsx” file there to follow it. React Native will process this path by itself automatically. We used the last approach since it is more customizable for bigger projects (e.g. having 2 files in one directory for profile page). In tabs you only need to see the name of the tabs (paths to the files), their icons to be displayed (here default were imported) and title name.

### 3.2. User interaction

One of the most common things are forms, buttons, links and so on. They let users interact with applications to create something, follow or change. React Native offers components that detect taps and touches: “TouchableOpacity” (Makes elements tappable and changes their opacity when pressed), “TouchableHighlight” (similar to the previous one but it also changes the background) and “Pressable” (component that responds to multiple types of user interactions such as pressing, long pressing, etc.).

As for the forms, React Native has some basic components: “TextInput” (it collects text input from users), “Picker” (allows users to select an option from a dropdown), “Switch” (just a toggle component for on/off values). All of them has a callback as a parameter to be executed after the event (action) is involved. All the above introduced approaches were demonstrated in the screenshots of the codes above.

### 3.3 Data handling

Data handling is more about getting, posting and storing data inside of the application or on a remote server. Generally, if we consider the requests to any API we use “fetch”, “axios” or any other third-party library. These mentioned functions are used in React applications or any JavaScript related code since React Native is still a react family solution. In the current project we simulated data uploading and fetching by creating local lists of constants where we store all the data. It doesn’t mean that we cheated somehow, no, since the logic of the code maintains the same. Figure 1.32 represents the location of the “testdata.ts” file containing all the constants.

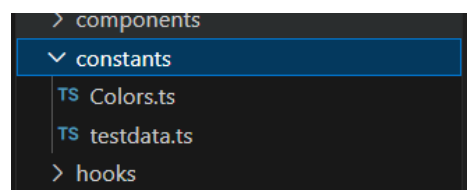


Figure 1.32 – Location of the project constants

Basically, we have constants of already existed users, posts as well as notifications, but in the page for adding new posts we increased the number of them as it could be done correctly, not manually.

## **CONCLUSION**

In conclusion, we managed to build an application for creating new posts, viewing all the posts, notifications, and user's detailed profile page in scope of the React Native environment.