

Book

Shopping

App

report

TABLE OF CONTENTS

PART

A	SCREENS, COMPONENTS AND REACT-NAVIGATION	3
	A.1 Screens & Layout	3
	A.2 Use of Components	15
	A.3 Reusability (Custom Components)	19
	A.4 Implementation of React-Navigation	21
B	DATA PERSISTENCE	25
	B.1 Usage of Data Persistence	25
	B.2 Implementation of Data Persistence	27
	B.3 CRUD Operations & Input Validation	32
C	CLOUD CONNECTIVITY	36
	C.1 Usage of Cloud Connectivity	36
	C.2 Implementation of Cloud Connectivity	39
	REFERENCES	40

BRIEF SUMMARY

Presenting the ABC bookstore app, the perfect gathering place for bibliophiles! With our intuitive app, explore, find, and buy your favourite books with convenience. From the comfort of your device, take advantage of safe payments, quick delivery, and personalized recommendations. Come explore our community and go on an unparalleled literary journey. Welcome to the ABC bookstore app, where you may access any book with just a tap!

KEY FEATURES

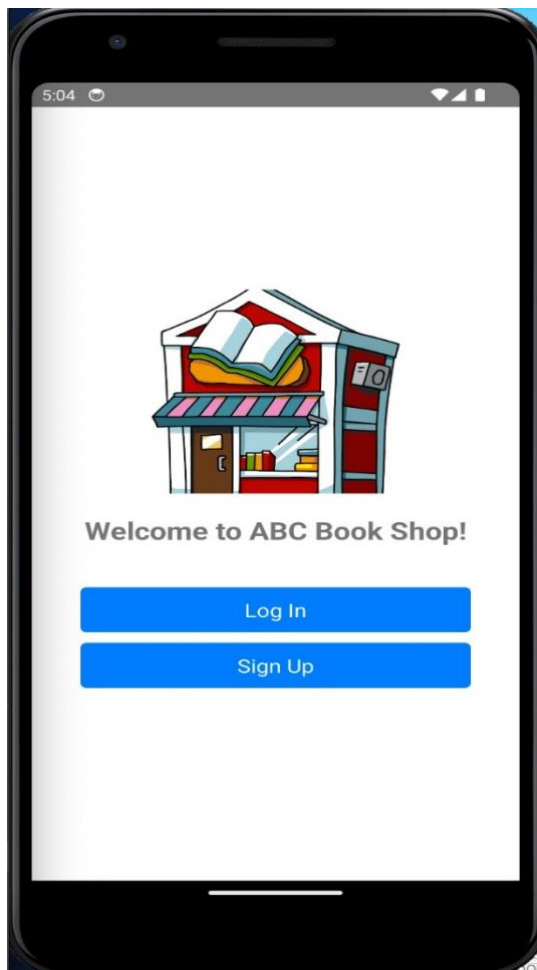
- 1) User Authentication: A secure login system ensures that enable users to login to their account
- 2) Search: Allow user to search their book they want by the book tittle
- 3) Add to cart: This feature allow user to add the book they want to buy to the cart in their account
- 4) Sign Up: Allow user to create their account after filling up their personal information
- 5) Profile Settings: Allow user to change their personal details.

Part A

SCREENS, COMPONENTS AND REACT-NAVIGATION

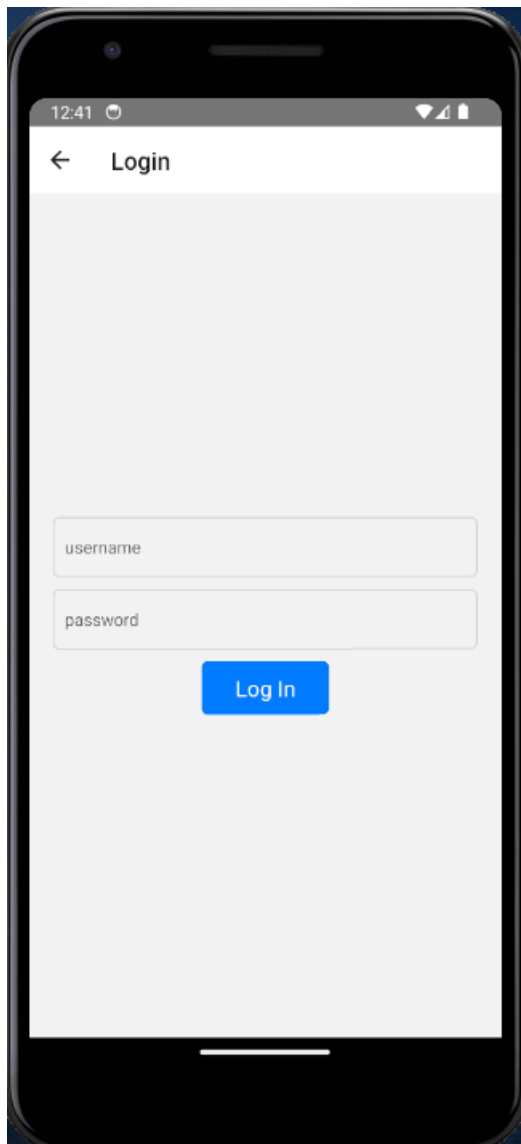
A.1 Screens & Layout

A.1.1 Start Up Screen



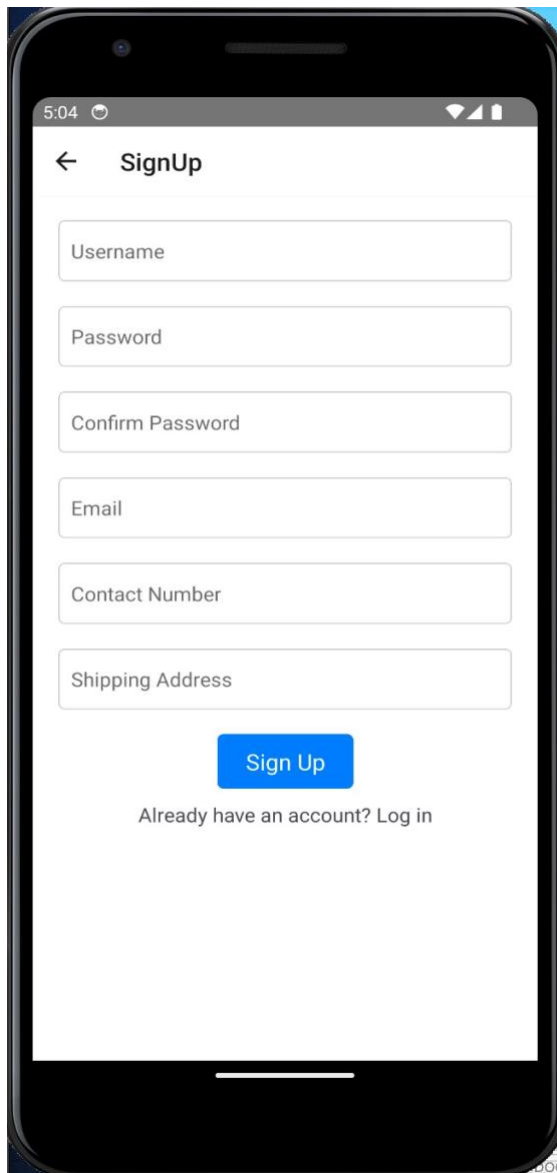
A Screen that displays the login and signup button

A.1.2 Login Screen



A screen that allows existing users to log in their account.

A.1.3 Sign Up Screen



5:04

← SignUp

Username

Password

Confirm Password

Email

Contact Number

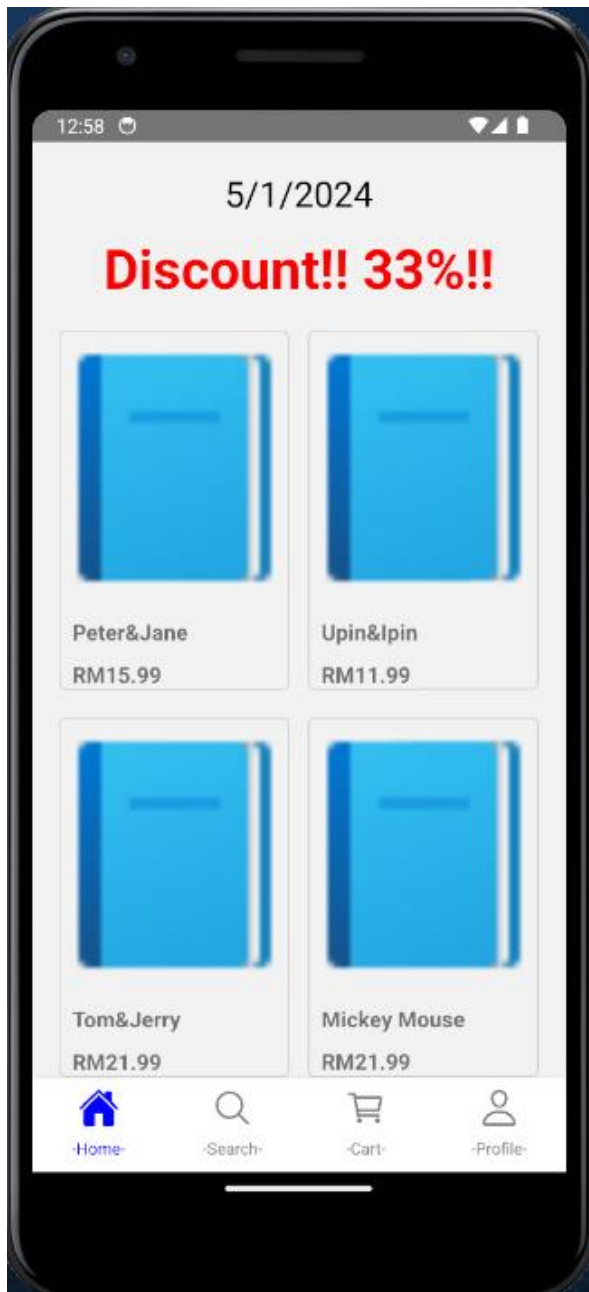
Shipping Address

Sign Up

Already have an account? Log in

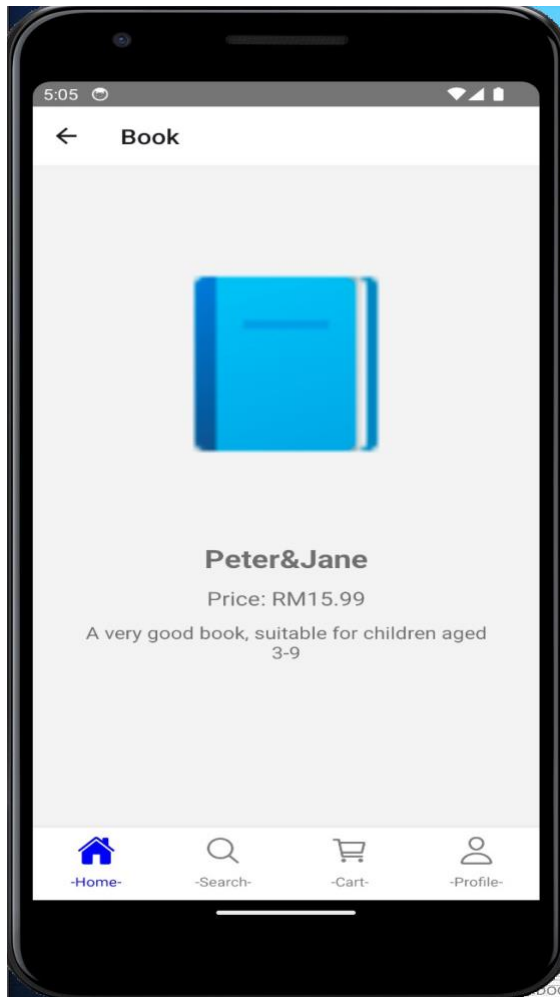
A screen that allows users to register their account by filling up their personal information.

A.1.4 Home Screen



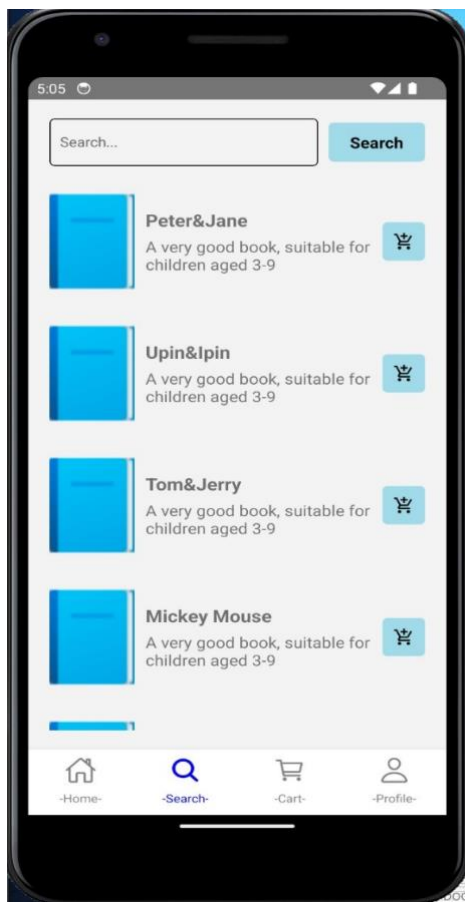
A screen that displays some books for users to select. Users can scroll down to see more books. Bottom tab is implemented for easy navigation.

A.1.5 Book Screen



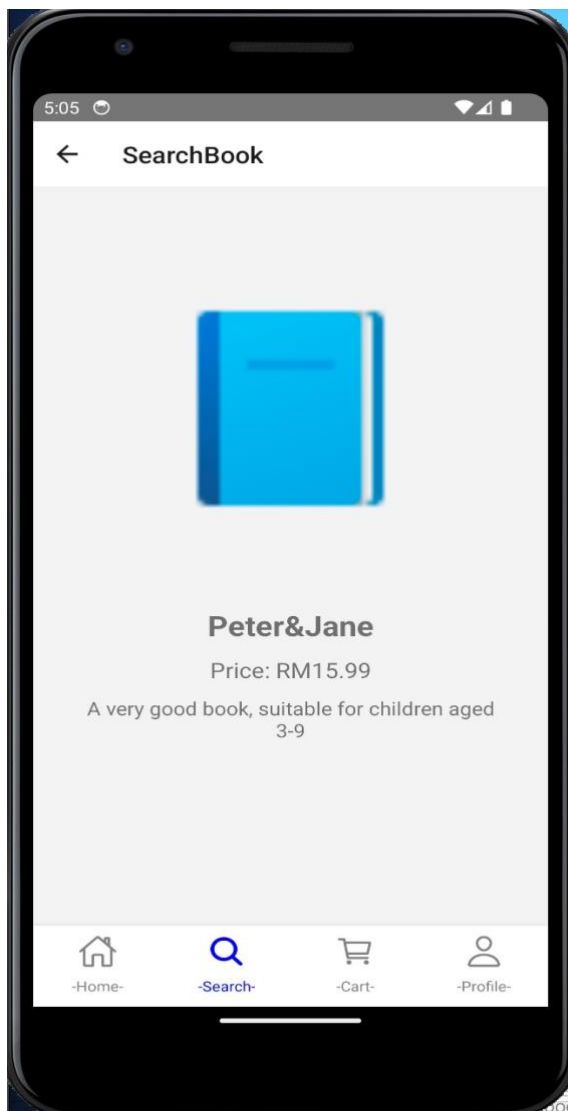
When user clicks the book icons in home screen, it will navigate to book screen where users can see more details about the book.

A.1.6 Search Screen



User can search book by name and use the add to cart button to place books into their cart.

A.1.7 Search Book Screen



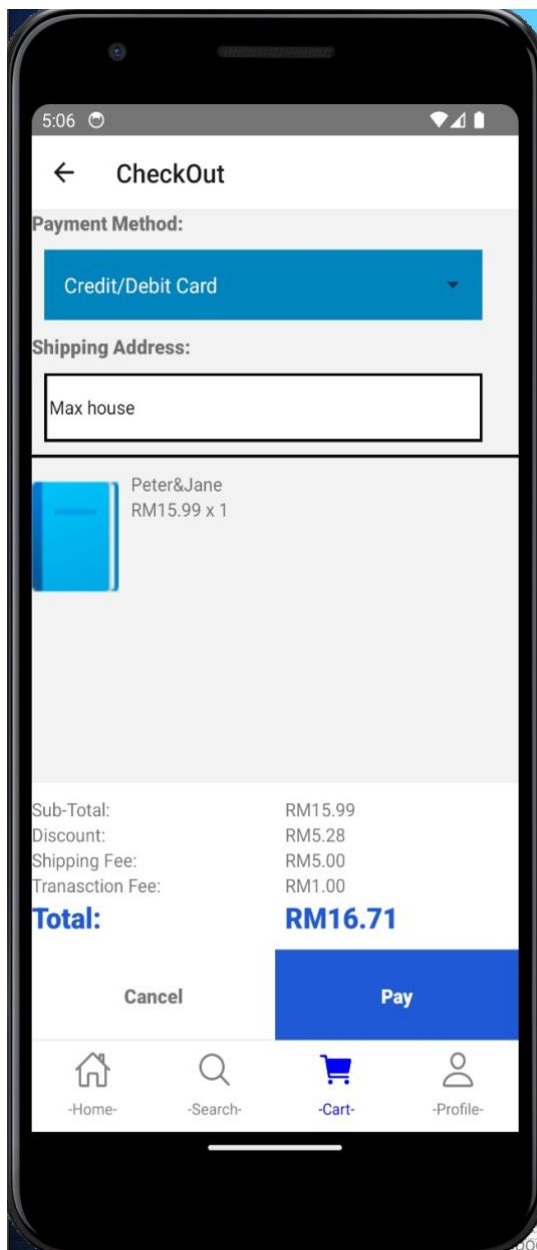
If user clicks on book icon in search screen, it will navigate to Search Book screen, displaying more details to user.

A.1.8 Cart Screen



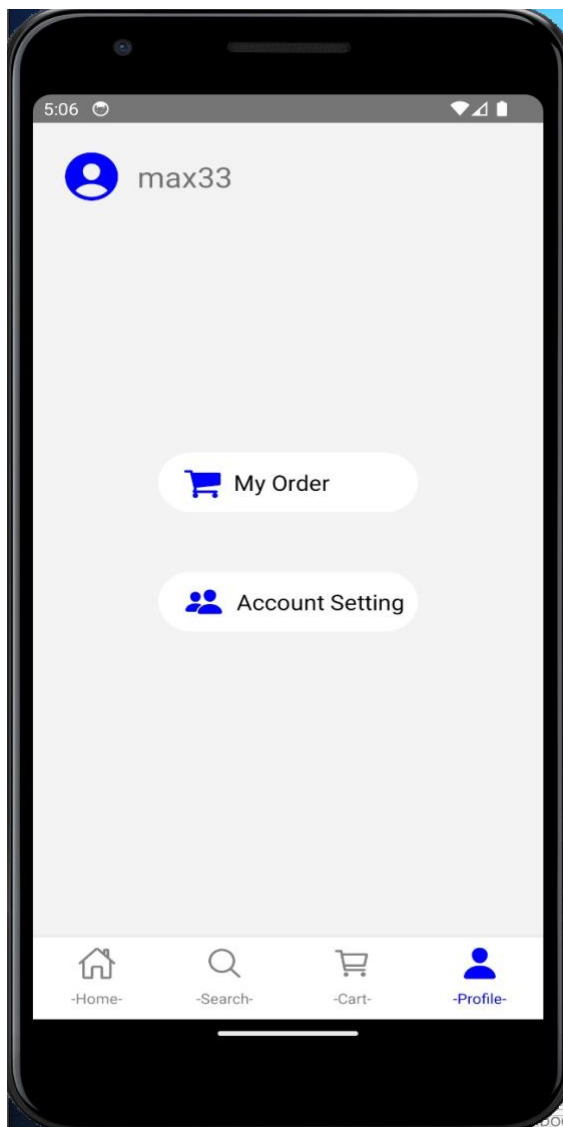
Books added to cart will be displayed here, user can change quantity, remove books and select books to checkout.

A.1.9 Check Out Screen



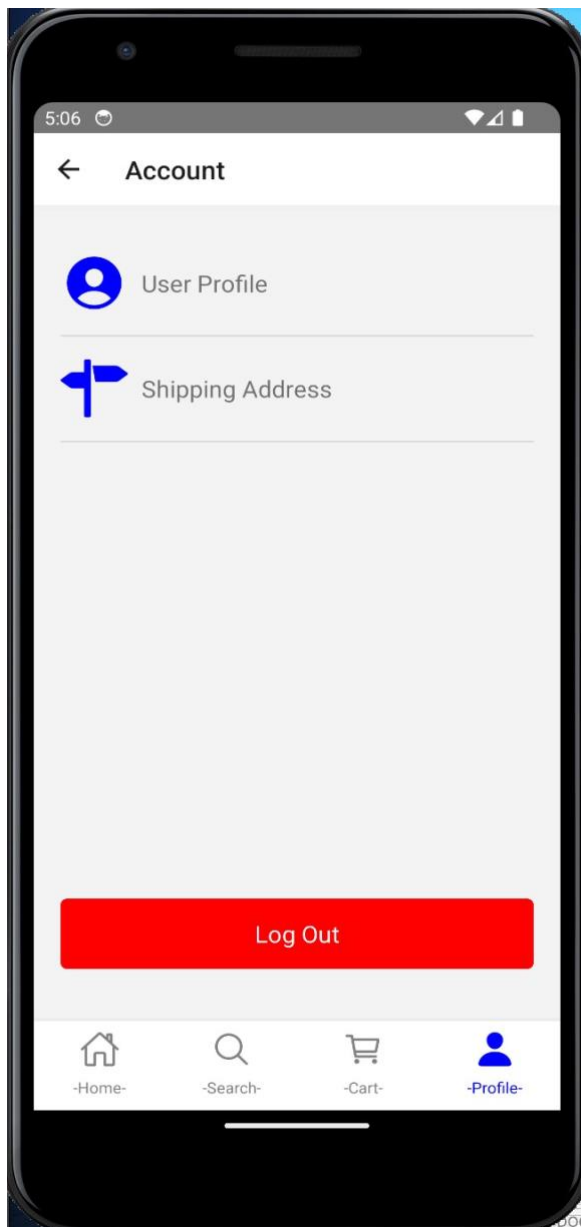
User can make payments in check out screen, user make select payment method and edit their shipping address.

A.1.10 Profile Screen



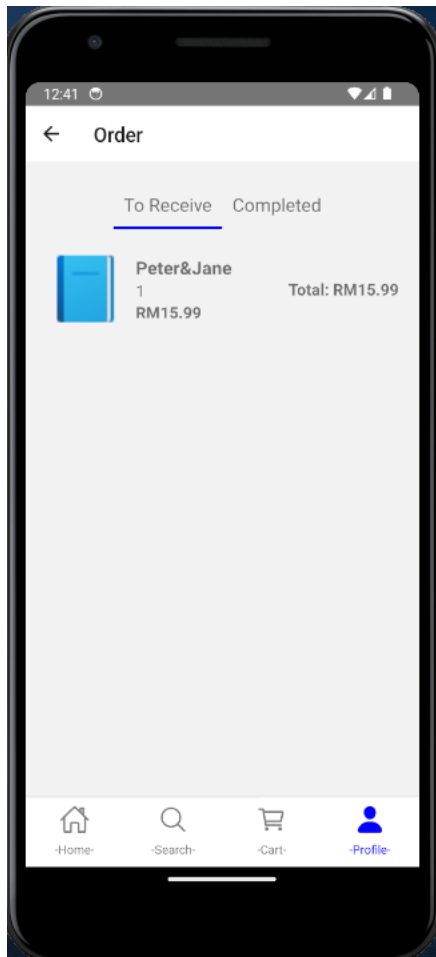
User can access personal details and data in profile screen

A.1.11 Account Screen



User can edit their personal information and log out in account screen.

A.1.12 Order screen



Users may track their order and order history

A.2 Use of Components

A.2.1 Combination of internal style and external style sheet

A.2.2.1 Internal style

```
const BookDetails = ({ book }) => {
  return (
    <View style={styles.container}>
      <View style={styles.imageContainer}>
        <Image source={require('../bookImg/bookcover.jpg')} style={styles.bookImage}
      />
      </View>
      <View style={styles.bookDetails}>
        <Text style={styles.bookName}>{book.bookName}</Text>
        <Text style={styles.bookPrice}>Price: RM{book.price}</Text>
        <Text style={styles.bookDescription}>{book.bookDes}</Text>
      </View>
    </View>
  );
};
```

In the book details screen the functional component is used to return the book cover image and the books details.

A.2.2.2 External Stylesheet

- External Stylesheet is used frequently in multiple screen such as account screen, Book Screen, Book Details Screen, and others.
- This stylesheet is aim for Performance, Consistency, and flexibility.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20
  },
  header: {
    flexDirection: 'row',
    justifyContent: 'flex-start',
    marginBottom: 20
  },
  content: {
    flex: 1
  },
});
```



```
settingItem: {
  flexDirection: 'row',
  alignItems: 'center',
  paddingVertical: 15,
  borderBottomWidth: 1,
  borderBottomColor: 'lightgray'
},
iconContainer: {
  marginRight: 10
},
settingText: {
  fontSize: 18
},
footer: {
  marginBottom: 20
},
logoutButton: {
  backgroundColor: 'red',
  paddingVertical: 15,
  borderRadius: 5
},
logoutButtonText: {
  color: 'white',
  fontSize: 18,
  textAlign: 'center'
},
modalContainer: {
  flex: 1,
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: 'rgba(0, 0, 0, 0.5)'
},
modalContent: {
  backgroundColor: 'white',
  padding: 20,
  borderRadius: 10,
  width: '80%'
},
modalTitle: {
  fontSize: 20,
  marginBottom: 10,
  color: 'black'
},
label: {
  fontSize: 16,
  marginBottom: 5,
  color: 'blue'
},
```

```
user: {
  fontSize: 16,
  marginBottom: 5,
  color: 'red'
},
address: {
  fontSize: 16,
  marginBottom: 5,
  color: 'blue'
},
input: {
  borderWidth: 1,
  borderColor: 'lightgray',
  borderRadius: 5,
  padding: 10,
  marginBottom: 10
},
buttonContainer: {
  flexDirection: 'row',
  justifyContent: 'space-between',
  marginTop: 20
}
});
```

Sample of stylesheet used in Account screen

A.2.2 Third Party Component

A.2.2.1 React-native vector icons

```
import Icon from 'react-native-vector-icons/Ionicons';  
profile Screen.js
```

- This component enhances usability and visual view with icons

A.2.2.2 @react-native-async-storage/async-storage

```
import AsyncStorage from '@react-native-async-storage/async-storage';  
App.js
```

- Multiple screen such as Account screen, Login screen, order screen and others use @react-native-async-storage/async-storage for managing user data storage
- By facilitating data persistence and retrieval, this component enhances the functioning of the program.

A.2.2.3 Text Input

```
import { TextInput } from "react-native-gesture-handler";
```

- Utilize a specialized text input component that provides enhanced gesture handling capabilities for improved user interactions, such as smoother scrolling and gestures like swiping or pinching.
- The text input from gesture handler is applied in the checkout screen.

A.2.2.4 Use Focus Effect

```
import { useFocusEffect } from '@react-navigation/native';
```

- Serves the purpose of utilizing a hook that allows you to perform side effects when the screen focuses. This is particularly useful for scenarios where you need to execute certain actions, such as fetching data or updating the UI, when a specific screen gain focus within a navigation stack.
- The focus effect is used in the profile screen.

A.3 Reusability (Custom Components)

Custom components are stored in UI.js and exported to screens that require it.

A.3.1 CheckOutItem

```
export function CheckOutItem({ item }) {
  return (
    <View style={{ flexDirection: 'row', marginVertical: 10 }}>
      <Image source={require('./bookImg/bookcover.jpg')} style={{ width: 70, height: 100,
marginRight: 10 }} />
      <View>
        <Text>{item.name}</Text>
        <Text>RM{item.price} x {item.quantity}</Text>
      </View>
    </View>
  );
}
```

Used in CheckOutScreen to print out list of books to check out

A.3.2 OrderItem

```
export function OrderItem ({ book }) {
  return (
    <View style={styles.bookItem}>
      <Image source={require("./bookImg/bookcover.jpg")} style={styles.bookImage} />
      <View style={styles.bookDetailsLeft}>
        <Text style={styles.bookName}>{book.bookName}</Text>
        <Text style={styles.bookQuantity}>{book.quantity}</Text>
        <Text style={styles.bookPrice}>RM{book.price}</Text>
      </View>
      <Text style={styles.bookPrice}>Total: RM{(book.price * book.quantity).toFixed(2)}</Text>
    </View>
  );
};
```

Used in order screen to display order history of user.

A.3.3 HomeItem

```
export function HomeItem ({ book }) {  
  return (  
    <View>  
      <Image source={require('./bookImg/bookcover.jpg')} style={styles.bookImage1} />  
      <Text style={styles.bookName1}>{book.bookName}</Text>  
      <Text style={styles.bookName1}>RM{book.price}</Text>  
    </View>  
  );  
};
```

Used in Home Screen to display books

A.4 Implementation of React-Navigation

A.4.1 Stack Navigation

A.4.1.1 Implementation of Stack Navigation

```
import { createStackNavigator } from '@react-navigation/stack';
```

- The `createStackNavigator` from React Navigation is used to create a stack-based navigation system.

A.4.1.2 Usage of Stack Navigation

```
const HomeStack = () => {  
  return (  
  
    <Stack.Navigator>  
      <Stack.Screen name="Home" component={HomeScreen} options={{headerShown:false}}  
initialParams={{ initialLoad: true }}/>  
      <Stack.Screen name="Book" component={BookScreen} options={{tabBarVisible:  
false}} />  
    </Stack.Navigator>  
  
  );  
};  
  
const SearchStack = () => {  
  return (  
  
    <Stack.Navigator>  
      <Stack.Screen name="Search" component={SearchScreen}  
options={{headerShown:false}} initialParams={{ initialLoad: true }}/>  
      <Stack.Screen name="SearchBook" component={SearchBookScreen}  
options={{tabBarVisible: false}} />  
    </Stack.Navigator>  
  
  );  
};  
  
const CartStack = () => {  
  return (  
  
    <Stack.Navigator>
```

```
        <Stack.Screen name="Cart" component={CartScreen} options={{headerShown:false}}
initialParams={{ initialLoad: true }}/>
        <Stack.Screen name="CheckOut" component={CheckOutScreen}
options={{tabBarVisible: false}} />
    </Stack.Navigator>

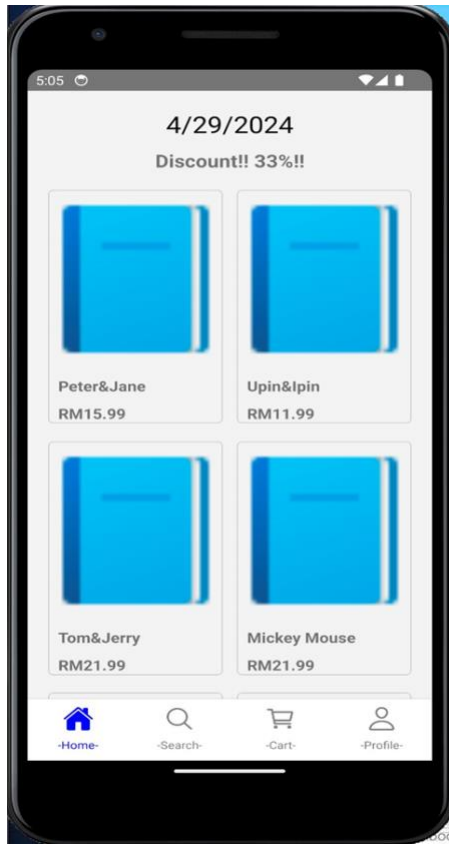
    );
};

const ProfileStack = () => {
    return (

        <Stack.Navigator>
            <Stack.Screen name="Profile" component={ProfileScreen}
options={{headerShown:false}} initialParams={{ initialLoad: true }}/>
            <Stack.Screen name="Order" component={MyOrderScreen} options={{tabBarVisible:
false}} />
            <Stack.Screen name="Account" component={AccountScreen} options={{tabBarVisible:
false}} />
        </Stack.Navigator>

    );
};
```

A.4.2 Bottom tab navigator



Home Screen

The bottom tab navigator allows user to switch different features screen

A.4.2.1 Implementation of bottom tab navigator

```
import { createBottomTabNavigator } from "@react-navigation/bottom-tabs";
```

- The createBottomTabNavigator from React Navigation is used to create a bottom-tab based navigation system.

A.4.2.2 usage of bottom tab navigator

```
<Tab.Navigator
  initialRouteName='Home'
  screenOptions={({ route }) => ({
    headerShown:false,
    tabBarActiveTintColor: 'blue',
```



```

    tabBarInactiveTintColor:'grey',
    tabBarLabelStyle:{fontSize:12},
    tabBarStyle:{paddingBottom:10,height:70},
    tabBarIcon: ({ focused, color, size }) => {
      let iconName;

      if (route.name === '-Home-') {
        iconName = focused ? 'home' : 'home-outline';
      }else if (route.name === '-Search-'){
        iconName = focused ? 'search' : 'search-outline';
      }else if (route.name === '-Cart-'){
        iconName = focused ? 'cart' : 'cart-outline';
      }
      else if (route.name === '-Profile-'){
        iconName = focused ? 'person' : 'person-outline';
      }

      return <Ionicons name={iconName} size={30} color={color} />;
    },

  )}}
>
<Tab.Screen name="-Home-" component={HomeStack} />
<Tab.Screen name="-Search-" component={SearchStack} />
<Tab.Screen name="-Cart-" component={CartStack} />
<Tab.Screen name="-Profile-" component={ProfileStack} />

</Tab.Navigator>

```

Part B

DATA PERSISTENCE

B.1 Usage of Data Persistence

B.1.1 async storage

B.1.1.1 Account Screen -async storage

```
const handleLogout = async() => {  
    await AsyncStorage.setItem('cus_id', '0');  
    navigation.navigate('Startup2');  
};
```

- In the Account Screen AsyncStorage is utilized for data persistence.
- AsyncStorage is a local storage feature that enables the user's device to store data.
- This data is stored in AsyncStorage and retrieved when the user accesses the Login Screen, Account Screen, order screen and others.
- This usage ensures that customer id remains accessible across different screen.

B.1.2 Database

```
const result = await db.executeSql(query, [input]);
console.log(result[0].rows.item(0))
// Check if any rows were returned
if (result[0].rows.length > 0) {
    // Get the user object
    const user = result[0].rows.item(0);

    // Get the password associated with the username
    const storedPassword=user.password;

    // Check if the entered password matches the stored password
    if (password === storedPassword) {
        console.log('cus_id:'+ user.id)
        //Upon successful login, store customer id
        await AsyncStorage.setItem('cus_id', user.id.toString());
        // Redirect the user to the main app screen
        navigation.navigate("Main2");
    } else {
        alert("Invalid password.");
    }
} else {
    alert("User not found.");
}
} catch (error) {
    console.error(error);
    alert("Failed to authenticate user.");
}
```

- This code handles user authentication by querying the database for the entered username, retrieving the stored password, comparing it with the entered password, and taking appropriate actions based on the comparison result.

B.2 Implementation of Data Persistence

B.2.1 SQLite Database Setup

```
import sqlite3
db = sqlite3.connect('bookshop.sqlite')

db.execute('DROP TABLE IF EXISTS customer')
db.execute(
    '''CREATE TABLE customer(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL,
        password TEXT NOT NULL,
        email TEXT NOT NULL,
        phone TEXT NOT NULL,
        shippingAddress TEXT NOT NULL
    )'''
)

db.execute(
    '''CREATE TABLE book(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        bookName TEXT NOT NULL,
        bookDes TEXT NOT NULL,
        price DOUBLE,
        imgLink TEXT NOT NULL
    )'''
)

db.execute(
    '''CREATE TABLE cart(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        cusID INTEGER,
        bookID INTEGER,
        quantity INTEGER
    )'''
)

db.execute(
    '''CREATE TABLE orders(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        cusID INTEGER,
        bookID INTEGER,
        quantity INTEGER,
        status INTEGER,
        shippingAddress TEXT NOT NULL
    )'''
)

db.execute('''
```

```

        INSERT INTO orders
(cusID,bookID,quantity,status,shippingAddress)
VALUES (1,1,1,1,'Max house')
'''

db.execute('''
        INSERT INTO orders
(cusID,bookID,quantity,status,shippingAddress)
VALUES (2,2,2,1,'Lewis house')
''')

db.execute('''
        INSERT INTO
customer(username,password,email,phone,shippingAddress)
VALUES('max33','abc123','mv33@gmail.com','011-1234567','Max
house')
''')

db.execute('''
        INSERT INTO
customer(username,password,email,phone,shippingAddress)
VALUES('lewis44','123abc','lh44@gmail.com','011-
1234567','Lewis house')
''')

db.execute('''
        INSERT INTO book(bookName,bookDes,price,imgLink)
VALUES ('Peter&Jane','A very good book, suitable for children
aged 3-9',15.99,'../bookImg/p1.jpg')
''')

db.execute('''
        INSERT INTO book(bookName,bookDes,price,imgLink)
VALUES ('Upin&Ipin','A very good book, suitable for children
aged 3-9',11.99,'../bookImg/p2.jpg')
''')

db.execute('''
        INSERT INTO book(bookName,bookDes,price,imgLink)
VALUES ('Tom&Jerry','A very good book, suitable for children
aged 3-9',21.99,'../bookImg/p3.jpg')
''')

db.execute('''
        INSERT INTO book(bookName,bookDes,price,imgLink)
VALUES ('Mickey Mouse','A very good book, suitable for
children aged 3-9',21.99,'../bookImg/p3.jpg')
''')

```

```
db.execute('''
    INSERT INTO book(bookName,bookDes,price,imgLink)
    VALUES ('Toy Story','A very good book, suitable for children
aged 3-9',15.99,'../bookImg/p3.jpg')
''')

db.execute('''
    INSERT INTO book(bookName,bookDes,price,imgLink)
    VALUES ('Captain Malaysia','A very good book, suitable for
children aged 3-9',15.99,'../bookImg/p3.jpg')
''')

db.execute('''
    INSERT INTO book(bookName,bookDes,price,imgLink)
    VALUES ('Family guy','A very good book, suitable for children
aged 3-9',20.99,'../bookImg/p3.jpg')
''')

db.commit()
db.close()
```

- This section of code creates tables to hold user information, book information and order status in addition to initializing an SQLite database. The createTables function is used to design the database schema, and the db variable is used to represent the SQLite database instance.

B2.2 SQLite

B.2.2.1 sign up

```
export const signUp = async(db: SQLiteDatabase ,
username:any,password:any,email:any,contactNumber:any,shippingAddress:any)=>{
  try{
    const query = 'INSERT INTO customer (username, password, email, phone,
shippingAddress) VALUES (?, ?, ?, ?, ?)';
    const parameters = [username,password,email,contactNumber,shippingAddress]
    await db.executeSql(query,parameters);
  } catch (error) {
    console.error(error);
    throw Error('Failed to insert customer !!!');
  }
}
```

- this function inserts a new customer record into an SQLite database table named customer, using the provided data, and handles any errors that occur during the process.

B.2.2.2 Update User Address

```
export const updateUserAddressByID = async(db: SQLiteDatabase , id:any,
address:any)=>{
  try {
    const query = `UPDATE customer SET shippingAddress=? WHERE id = ?`;
    await db.executeSql(query, [address,id]);
    console.log(`User shipping address updated successfully!`);
  } catch (error) {
    console.error('Failed to update user:', error);
    throw new Error('Failed to update user!');
  }
}
```

- This function updates the shipping address of a user in the customer table of an SQLite database based on their ID, handles any errors that occur during the process, and provides appropriate feedback via console logging.

B.2.2.3 Get Book by Id

```
export const getBookById = async( db: SQLiteDatabase , id:any ): Promise<any> => {  
  try{  
  
    const query = `SELECT * FROM book WHERE id = ?`;   
  
    let result = await db.executeSql(query,[id]);  
  
    return result;  
  } catch (error) {  
    console.error(error);  
    throw Error('Failed to get searched book !!!');  
  }  
}
```

- This function retrieves book information from the book table in an SQLite database based on the provided book ID, handles any errors that occur during the process, and returns the result or throws an error accordingly

B.3 CRUD Operations & Input Validation

B.3.1 CRUD Operations

B.3.1.1 Create

```
export const signUp = async(db: SQLiteDatabase ,
username:any,password:any,email:any,contactNumber:any,shippingAddress:any)=>{
  try{
    const query = 'INSERT INTO customer (username, password, email, phone,
shippingAddress) VALUES (?, ?, ?, ?, ?)';
    const parameters = [username,password,email,contactNumber,shippingAddress]
    await db.executeSql(query,parameters);
  } catch (error) {
    console.error(error);
    throw Error('Failed to insert customer !!!');
  }
}
```

- User can create their personal account by filling up their username, password, email, phone, and shipping address by clicking the sign-up button in the startup page
- If the system failed to get the data, it will throw an error says that failed to insert customer.

B.3.1.2 Read

```
export const getOrder = async( db: SQLiteDatabase, cusID:any ): Promise<any> => {
  try{
    const books : any = [];
    const query = `SELECT * FROM orders WHERE cusID = ?`;
    const results = await db.executeSql(query,[cusID]);
    results.forEach(result => {
      (result.rows.raw()).forEach(( item:any ) => {
        books.push(item);
      })
    });
    console.log('orders loaded');
    return books;
  } catch (error) {
    console.error(error);
    throw Error('Failed to get orders !!!');
  }
}
```

- Read operation is used to fetch the customer's order based on their id

- It constructs a SQL query to select matching records, executes it using the `db` object, and returns the results. If successful, it logs "orders loaded"; otherwise, it logs an error and throws an exception.

Update

```
export const updateUserAddressByID = async(db: SQLiteDatabase , id:any,
address:any)=>{
  try {
    const query = `UPDATE customer SET shippingAddress=? WHERE id = ?`;
    await db.executeSql(query, [address,id]);
    console.log(`User shipping address updated successfully!`);
  } catch (error) {
    console.error('Failed to update user:', error);
    throw new Error('Failed to update user!');
  }
}
```

- Update operation is used to set the customer's shipping address when their shipping address is change.
- It constructs and executes a SQL query to update the "customer" table with the new address based on the provided user ID. If successful, it logs a confirmation message; otherwise, it logs an error and throws an exception.

Delete

```
export const removeCart = async(
  db: SQLiteDatabase,
  cusID: number,
  bookID:number
) => {
  try{
    const query = 'DELETE FROM cart WHERE cusID = ? AND bookID = ?' ;
    await db.executeSql(query,[cusID,bookID]);
    console.log("remove successful");

  } catch (error) {
    console.error(error);
    throw Error('Failed to remove book !!!');
  }
}
```

- Delete operation is used which allow user to remove the books from their cart if they decided not to buy it

- This function removes a record from a SQLite database table named "cart" based on the provided customer ID (cusID) and book ID (bookID). It constructs and executes a SQL query to delete the record. Upon successful deletion, it logs "remove successful". If an error occurs, it logs the error and throws an exception with the message 'Failed to remove book !!!'

B.3.2 Input Validation

B.3.2.1 Error Input Validation

```
// Check if the entered password matches the stored password
if (password === storedPassword) {
  console.log('cus_id:' + user.id)
  //Upon successful login, store customer id
  await AsyncStorage.setItem('cus_id', user.id.toString());
  // Redirect the user to the main app screen
  navigation.navigate("Main2");
} else {
  alert("Invalid password.");
}
} else {
  alert("User not found.");
}
} catch (error) {
  console.error(error);
  alert("Failed to authenticate user.");
}
```

Login Screen

- In the login screen the system will be validated is that the password matches with the password store in the databased if it is wrong then the system will alert invalid password

B.3.2.2 Empty Input validation

```
if (!username.trim()) {
    console.error("Username is required");
    return; //Stop execution if validation fails
};

if (!email.trim()) {
    console.error("Email is required");
    return;
}

else if (!isValidEmail(email)){
    console.error("Invalid email format");
    return;
};

if (!password.trim()) {
    console.error("Password is required");
    return;
};

if (!confirmPassword.trim()) {
    console.error("Confirm password is required");
    return;
};

if (!contactNumber.trim()) {
    console.error("Contact number is required");
    return;
};

if (!shippingAddress.trim()) {
    console.error("Shipping address is required");
    return;
};

if (password !== confirmPassword) {
    console.error("Passwords do not match");
    return;
}
```

Sign up screen

- If the user leave empty adds the contact number, it will display an error message says that contact number is required
- When the e-mail format doesn't match the validation error message will also show that invalid e-mail format

Part C

CLOUD CONNECTIVITY

C.1 Usage of Cloud Connectivity

C.1.1 API 1-Compute Shipping Fee

```
from flask import Flask
from flask_socketio import SocketIO, emit
import json
import math

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)

@socketio.on('connect', namespace='/shipping')
def handle_connect_shipping():
    print('Connected to /shipping')

@socketio.on('client_connected', namespace='/shipping')
def handle_client_connected_shipping(data):
    print('Connection Status: {}'.format(data['connected']))

@socketio.on('client_send', namespace='/shipping')
def handle_client_send_shipping(data):
    quantity = data['quantity']

    # Compute shipping fee
    shipping = 5*quantity

    # Emit result to client
    emit('server_send', json.dumps({
        'shipping':shipping
    }), namespace='/shipping')

if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5001, debug='true')
```

C.1.2 API 2-Calculate processing fee

```
from flask import Flask
from flask_socketio import SocketIO, emit
import json
import math

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)

@socketio.on('connect', namespace='/fee')
def handle_connect_fee():
    print('Connected to /fee')

@socketio.on('client_connected', namespace='/fee')
def handle_client_connected_fee(data):
    print('Connection Status: {}'.format(data['connected']))

@socketio.on('client_send', namespace='/fee')
def handle_client_send_fee(data):
    quantity = data['quantity']

    # Compute shipping fee
    fee = 1*quantity

    # Emit result to client
    emit('server_send', json.dumps({
        'fee':fee
    }), namespace='/fee')

if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5002, debug='true')
```

C.1.3 API 3 -Calculate Discount

```
from flask import Flask
from flask_socketio import SocketIO, emit
import json
import math

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)

@socketio.on('connect', namespace='/discount')
def handle_connect_discount():
    print('Connected to /discount')

@socketio.on('client_connected', namespace='/discount')
def handle_client_connected_discount(data):
    print('Connection Status: {}'.format(data['connected']))

@socketio.on('client_send', namespace='/discount')
def handle_client_send_discount(data):
    amount = data['amount']

    # Compute shipping fee
    discount = 0.33*amount

    # Emit result to client
    emit('server_send', json.dumps({
        'discount':discount
    }), namespace='/discount')

if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5003, debug='true')
```

C.2 Implementation of Cloud Connectivity

C.2.1 API 1-Compute Shipping Fee

- This API allows clients to connect to a WebSocket server (/shipping namespace) and send requests to calculate shipping fees based on the quantity provided. The server responds with the calculated shipping fee.
- When user navigates to the checkout screen the shipping fess will be calculated and show on the screen

C.2.2 API 2-Calculate Transaction fee

- This API establishes a WebSocket connection with clients in the /fee namespace, receives quantity data from clients, calculates Transaction fees based on that quantity, and sends the fee back to the client.
- When user navigates to the checkout screen the processing fees will be calculated and show on the screen

C.2.3 API 3 -Calculate Discount

- This API establishes a WebSocket connection with clients in the /discount namespace, receives amount data from clients, calculates a discount based on that amount, and sends the discount back to the client.
- When user navigates to the checkout screen the discount amount will be calculated and show on the screen

REFERENCES

React native · learn once, write anywhere (no date) React Native RSS. Available at: <https://reactnative.dev/> (Accessed:).

END OF DOCUMENT