LAZY LOADING

What is our GOAL for this MODULE?

In this class, you learned to display items in a list using the FlatList component and build a search bar in the search screen to display the list of transactions queried by the user.

What did we ACHIEVE in the class TODAY?

- Displayed transactions in a FlatList.
- Built a search bar to query transactions collection to search for a field value.
- Displayed the query results in a FlatList.

Which CONCEPTS/CODING BLOCKS did we cover today?

- ScrollView
- FlatList
- Search bar

How did we DO the activities?

Loading data in bulk takes time and is resource-consuming, to tackle this problem you use a technique called **Lazy Loading**. **Lazy Loading** is the technique of rendering critical user-interface items first, then unrolling/loading the non-critical items later. Hence, you can say Lazy Loading is a technique that can be used in FlatList.

1. Create a state called **allTransactions** as an empty array.

```
export default class SearchScreen extends Component {
  constructor(props) {
    super(props);
    this.state = {
      allTransactions: [],
    };
  }
```

2. Store the entire list of transactions inside this state when the component mounts. **Import** db from **config.js** and write the code to modify the state **allTransactions** inside the **componentDidMount()** function.

screens > JS Search.js > ...

```
1  import React, { Component } from "react";
2  import {
3    View,
4    StyleSheet,
5    TextInput,
6    TouchableOpacity,
7    Text,
8    FlatList
9  } from "react-native";
10
11 import db from "../config";
12
13 export default class SearchScreen extends Component {
14   constructor(props) {
15     super(props);
16     this.state = {
17       allTransactions: [],
18     };
19   }
```

```
getTransactions = () => {
  db.collection("transactions")
    .get()
    .then(snapshot => {
      snapshot.docs.map(doc => {
        this.setState({
          allTransactions: [...this.state.allTransactions, doc.data()]
        });
      });
    });
};
```

Now that you have the data that you want, but you don't have a way to present it in the app. You'll be using FlatList to showcase this data.

3. Use **FlatList** to list down all the transactions.

   FlatList has three key props:

   - **data**: This contains all the data in the array which needs to be rendered.
   - **renderItem:** This takes each item from the data array and renders it as described using JSX. This should return a JSX component.
   - **keyExtractor:** It gives a unique key prop to each item in the list. The unique key prop should be a string.

```
<View style={styles.lowerContainer}>
  <FlatList
    data={allTransactions}
    renderItem={this.renderItem}
    keyExtractor={(item, index) => index.toString()}
  />
</View>
```

4. Explore the code for **renderItem()** function.

```
renderItem = ({ item, i }) => {
  var date = item.date
    .toDate()
    .toString()
    .split(" ")
    .splice(0, 4)
    .join(" ");

  var transactionType =
    item.transaction_type === "issue" ? "issued" : "returned";
  return (
    <View style={{ borderWidth: 1 }}>
      <ListItem key={i} bottomDivider>
        <Icon type={"antdesign"} name={"book"} size={40} />
        <ListItem.Content>
          <ListItem.Title style={styles.title}>
            {`${item.book_name} ( ${item.book_id} )`}
          </ListItem.Title>
          <ListItem.Subtitle style={styles.subtitle}>
            {`This book ${transactionType} by ${item.student_name}`}
          </ListItem.Subtitle>
          <View style={styles.lowerLeftContaiiner}>
            <View style={styles.transactionContainer}>
              <Text
                style={[
                  styles.transactionText,
                  {
                    color:
                      item.transaction_type === "issue"
                        ? "#78D304"
                        : "#0364F4"
                  }
                ]}
              >
                {item.transaction_type.charAt(0).toUpperCase() +
                  item.transaction_type.slice(1)}
              </Text>
              <Icon
                type={"ionicon"}
                name={
                  item.transaction_type === "issue"
                    ? "checkmark-circle-outline"
                    : "arrow-redo-circle-outline"
                }
                color={
                  item.transaction_type === "issue" ? "#78D304" : "#0364F4"
                }
              />
            </View>
            <Text style={styles.date}>{date}</Text>
          </View>
        </ListItem.Content>
      </ListItem>
    </View>
  );
};
```

Output:

Now you have all the transactions in our database, what if you want to check a transaction for a specific student or the book? To solve this issue, you use the **handleSearch** functionality, which allows you to search for any **bookId** or **studentId**.

5. Write a **handleSearch()** function to search for the desired book ID or student ID.
    - Get the entered text from the user and split it to get the first letter of the word.
    - If the first letter is "**B**" then make a query on the books collection using the given **bookId**.
    - If the first letter is "**S**" then make a query on the students' collections using the given **studentId**.
    - Set the data to the **allTransactions** filed in state.

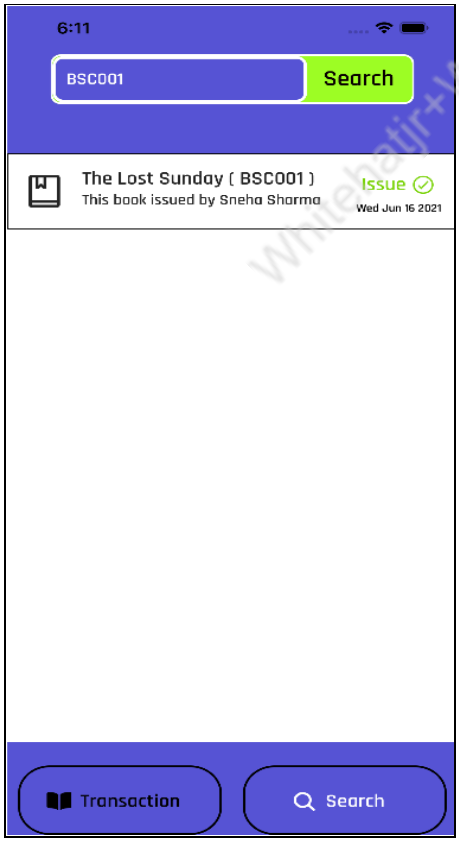
```
handleSearch = async text => {
  var enteredText = text.toUpperCase().split("");
  text = text.toUpperCase();
  this.setState({
    allTransactions: []
  });
  if (!text) {
    this.getTransactions();
  }

  if (enteredText[0] === "B") {
    db.collection("transactions")
      .where("book_id", "==", text)
      .get()
      .then(snapshot => {
        snapshot.docs.map(doc => {
          this.setState({
            allTransactions: [...this.state.allTransactions, doc.data()]
          });
        });
      });
  } else if (enteredText[0] === "S") {
    db.collection("transactions")
      .where("student_id", "==", text)
      .get()
      .then(snapshot => {
        snapshot.docs.map(doc => {
          this.setState({
            allTransactions: [...this.state.allTransactions, doc.data()]
          });
        });
      });
  }
};
```


6. Now, call the **handleSearch()** function when the search button is clicked. Run the code to test it.

```
return (
  <View style={styles.container}>
    <View style={styles.upperContainer}>
      <View style={styles.textinputContainer}>
        <TextInput
          style={styles.textinput}
          onChangeText={text => this.setState({ searchText: text })}
          placeholder={"Type here"}
          placeholderTextColor={"#FFFFFF"}
        />
        <TouchableOpacity
          style={styles.scanbutton}
          onPress={() => this.handleSearch(searchText)}
        >
          <Text style={styles.scanbuttonText}>Search</Text>
        </TouchableOpacity>
      </View>
    </View>
```

Output:



7. Next, create a state called **lastVisibleTransaction** and set its value to **null** as its initial value, and in the **handleTransaction()** function add the limit using the **limit()** function to get **10** transactions.

```
export default class SearchScreen extends Component {
  constructor(props) {
    super(props);
    this.state = {
      allTransactions: [],
      lastVisibleTransaction: null,
      searchText: ""
    };
  }
```

```
  getTransactions = () => {
    db.collection("transactions")
      .limit(10)
      .get()
      .then(snapshot => {
        snapshot.docs.map(doc => {
          this.setState({
            allTransactions: [...this.state.allTransactions, doc.data()],
            lastVisibleTransaction: doc
          });
        });
      });
  };
```


FlatList has two more important props as follows:

- **onEndReached** can call a function to get more transaction documents after the last transaction document you fetched.
- **onEndThreshold** defines when you want to call the function inside the **onEndReached** prop. If **onEndThreshold** is **1**, the function will be called when the user has completely scrolled through the list. If **onEndThreshold** is **0.5**, the function will be called when the user is mid-way while scrolling the items.

8. Now set the **onEndReachedThreshold** as **0.7** and write a function to fetch more transaction documents after the last transaction document you fetched.

```
return (
  <FlatList
    data={this.state.allTransactions}
    renderItem={({item})=>(
      <View style={{borderBottomWidth: 2}}>
        <Text>{"Book Id: " + item.bookId}</Text>
        <Text>{"Student id: " + item.studentId}</Text>
        <Text>{"Transaction Type: " + item.transactionType}</Text>
        <Text>{"Date: " + item.date.toDate()}</Text>
      </View>
    )}
    keyExtractor= {(item, index)=> index.toString()}
    onEndReached ={this.fetchMoreTransactions}
    onEndReachedThreshold={0.7}
  />
);
}
```

```
fetchMoreTransactions = async ()=>{
  const query = await db.collection("transactions").startAfter(this.state.lastVisibleTransaction).limit(10).get()
  query.docs.map((doc)=>{
    this.setState({
      allTransactions: [...this.state.allTransactions, doc.data()],
      lastVisibleTransaction: doc
    })
  })
}
```

Output:

You can now see how lazy loading works. Every time you scroll down, you see new items getting added to the list.

CS-PRO-C74(V3)

WhiteHatJr
Live Online Coding for Kids

What's NEXT?

In the next class, you will learn to authenticate a user in their app by asking them to enter their username and password. You will also learn to change security rules for their database so that only authenticated users can access - read and write values - to the database.

EXTEND YOUR KNOWLEDGE

You can read this blog, lazy loading, to know more about it.
To learn more about the split() function refer to this documentation
To learn more about the splice() function refer to this documentation
To learn more about the join() function refer to this documentation