CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

This project is implemented using Javascript (React Native). In this chapter, we will discuss about how the program works.

a. Fetch Data

```
1. _getData = () => {
2.   let mToken  = this.props.user.token
3.   let mUrl    = ApiConstants.GetPedagang
4.   ApiMiddleware(mUrl, null, 'GET', mToken)
5.   .then((json) => {
6.     if( json.status == 200 ) {
7.       let mResult = json.res.result == null ? [] :
   json.res.result
8.       this.setState({
9.         dataPedagang: [...mResult]
10.      })
11.    }
12.    else {
13.      alert(JSON.stringify(json))
14.    }
15.  })
16.  }
```

The code above shows the process of fetch the data from API. When json response is 200 then the data is successfully fetched. In line 6, when json status is 200, the response json.status.result will store in variable called mResult that have a conditional, when the response is not empty the data will be json.status.result and when the data is empty, mResult will contain empty array. After the data successfully fetched, mResult will be assign to local state called dataPedagang in line 8.

b. Processing data using ScrollView with Array.Map()

```
1.  <ScrollView>
2.    { dataPedagang.map((o) =>(
3.      <TouchableOpacity
4.       onPress={()=>this._getTagihan(o.idpedagangkios) }
5.      >
6.        <View style={{flexDirection: 'row',
          justifyContent:'space-between' }}>
7.          <Text text={ o.noreg } />
8.          <Text text={ o.noseri } />
9.       </View>
10.      <View style={{ flexDirection: 'row',
         justifyContent:'space-between' }}>
11.        <Text text={ o.namapedagang } />
12.        <Text text={ o.namacorporate } />
13.      </View>
14.    </TouchableOpacity>
15.    ))
16.    }
17.    </ScrollView>
```

The code above shows how to process the data by using ScrollView. We call state called dataPedagang and use map() to looping the data. As we can see, the data that we will display are noreg, noseri, namapedagang, namacorporate. Since it's using map() which result an array, we should call the object using parameter 'o' from what we have declared in line 2. In line 4, onPress is a prop that contains a function if there is an action from the user. In the onPress above, if there is an action from the user, it will run the _getBill(o.idpedagangkios) function by bringing the idpedagankios data obtained from the mapped data.

c. Processing data using ScrollView with Lodash

```
1.  <ScrollView>
2.  { _.map(dataPedagang, (o) =>(
3.  <TouchableOpacity
4.  style={{
5.  width: metrics.screenWidth,
6.  borderTopWidth: 1,
7.  borderColor: '#EEEEEE',
8.  paddingHorizontal: 10,
9.  paddingVertical: 20
10. }}
11. onPress={ () => this._getTagihan(o.idpedagangkios) }
12. >
13. <View style={{ flexDirection: 'row', justifyContent:
    'space-between' }}>
14. <Text text={ o.noreg } />
15. <Text text={ o.noseri } />
16. </View>

17. <View style={{ flexDirection: 'row', justifyContent:
    'space-between' }}>
18. <Text text={ o.namapedagang } />
19. <Text text={ o.namacorporate } />
20. </View>
21. </TouchableOpacity>
22. ))
23. }
24. </ScrollView>
```

Structurally it is similar to ScrollView with Array.Map(). However, when it comes to processing data, Lodash has an easier way, namely by directly calling the data to be processed. As can be seen in line 2 where Lodash uses the '_' command then followed by '.map' which is functionally the same as Array.Map(). Just like Array.Map(), to loop through data requires a variable. The variable used is the same as Array.Map(), namely the 'o' variable which can be replaced according to taste such as 'id' or 'index'.

d. Processing data using FlatList

```
1. <FlatList
2.   data={ dataPedagang ? dataPedagang : [] }
3.   keyExtractor={ this._keyExtractor }
4.   renderItem={ ({ item }) =>
5.   <ListData noreg={ item.noreg }
6.      noseri={ item.noseri }
7.      namapedagang={ item.namapedagang }
8.      namacorporate={ item.namacorporate }
9.      noregFunction={()=>
10.     this._getTagihan(item.idpedagangkios) }/>
11.   }
12.   initialNumToRender={ 10 }
13.   />
14.
15.   _keyExtractor = (item, index) => index.toString()
16.
17.   const ListData = (props) => {
18.     return(
19.       <TouchableOpacity
20.         onPress={ props.noregFunction }
21.       >
22.
23.       <View>
24.         <Text text={ props.noreg } />
25.         <Text text={ props.noseri } />
26.      </View>
27.
28.      <View>
29.        <Text text={ props.namapedagang } />
30.        <Text text={ props.namacorporate } />
31.      </View>
32.    </TouchableOpacity>
33.   )
34.   }
```

This is how FlatList implemented. In line 2, we must fill the data props to display the data, so we fill with dataPedagang. RenderItem in line 4 is the item that we want to render and show to user. The renderItem works similar with map() which using parameter to call the object. On line12, InitialNumToRender is a useful prop to limit the data that appears when the FlatList first appears. In the above code,

InitialNumToRender is set to 10, and also the default InitialNumToRender on FlatList is 10 items.

e. Processing data using FlatList Pagination

We can optimize & customize FlatList for optimal & better performance, so we will use pagination with FlatList. But, for pagination we need sent something to API to provide the data that we need. For this case, we will send traders name keyword to API and the result that API provide is all traders with contain the keyword. The pagination system in FlatList is if there is 1 limit that contains 10 data and it reaches the end, then the offset will be added by 1, where the number of limits becomes 20.

```
1. _getData = () => {
2. let offset  = this.state.offset //1
3. let limit   = this.state.limit //10
4. let search  = this.state.search //''
5. let mToken  = this.props.user.token
6. let mUrl    = ApiConstants.GetPedagangLimit + '/' + ((offset-
   1)*limit) + '/' + limit + '/' + 0 + '/' + search


7. ApiMiddleware(mUrl, null, 'GET', mToken)
8. .then((json) => {
9.   if( json.status == 200 ) {
10.      let mResult = json.res.result == null ? [] :
        json.res.result
11.      this.setState({
12.        dataPedagang: [...this.state.dataPeda
          gang, ...mResult]
13.      })
14.      }
15.      else {
16.        if(offset == 1){
17.          Alert.alert('Peringatan!',   'Terjadi   kesalahan.
   mohon diulang kembali')
18.      }
19.      else{
20.      this.setState({
21.      searchMore: false
22.      })
23.      }
```

```
24.        }
25.        })
26.        }
```

This is the function code for fetch data, for the API, we need to send offset, limit, and search which is keyword. In Line 12, we need to reassign dataPedagang state so that it is not replaced by upcoming data. Then we need to create a function, where when the data reached the end it will add more data. On line 16 it will be checked whether the offset is 0 or something else. If the first time the function is executed and the offset is 1, then the data cannot be obtained, or an error occurs "Peringatan! Terjadi kesalahan. Mohon diulang kembali".

```
1. _searchFilter = (search) => {
2. this.setState({
3. search: search
4. }, () => {
5. if (this.state.search.length > 3)
6. this._getData()
7. this._checkSession()
8. })
9. }
```

The function above is a function to send keywords to the server. In line 1, the search parameter is obtained through user input which will be passed to this function. In line 5, it will check whether the number of keyword characters is more than 3 characters or not. If the number of characters is more than 3, the _getData() and _checkSession() functions will be executed.

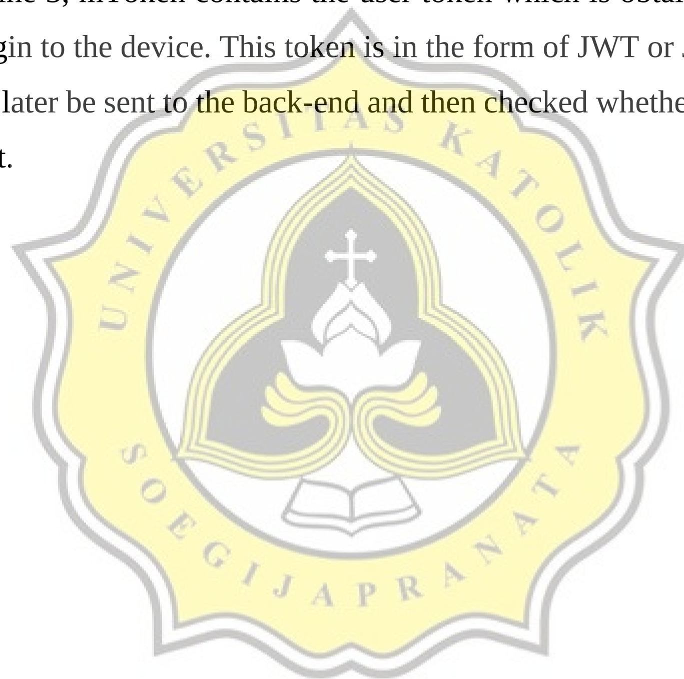
```
1. _checkSession = () => {
2. let mUrl = ApiConstants.SessionCheck
3. let mToken = this.props.user.token
4. ApiMiddleware(mUrl, null, 'GET', mToken)
5. .then((json) => {
6. if( json.status == 401 ){
7. Alert.alert('Peringatan!', 'Sesi anda habis. Silahkan
   masuk kembali')
8. this.props.onSetAlertLogout(true)
9. }else{
10.    }
11.    }).catch((err) => {
12.    Alert.alert('Peringatan!', 'Sesi anda habis. Silahkan
   masuk kembali')
13.    this.props.onSetAlertLogout(true)
14.    })
15.    }
```

This function is used to check the session of the user before fetching data from the API. If json.status is 401 as in line 6, a warning will appear “Warning! Your session has expired. Please log back in” and the user is forced to log out and

log back in. This is done so that the security of the application is maintained by other parties.

In line 2, mUrl contains the API url of the back-end which is called through the ApiConstants file with the variable name SessionCheck. So the way to call it is ApiConstans.SessionCheck.

In line 3, mToken contains the user token which is obtained from the time the user login to the device. This token is in the form of JWT or JSON Web Token which will later be sent to the back-end and then checked whether this token is still valid or not.



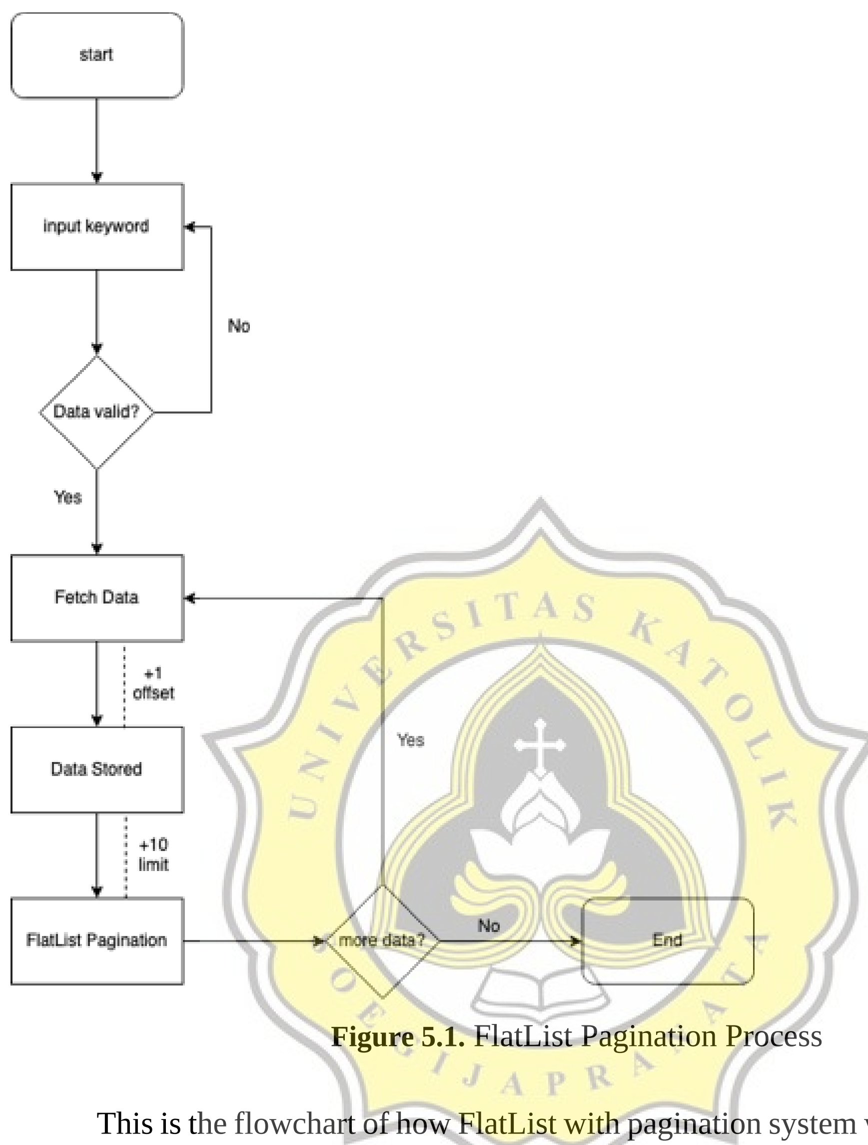


Figure 5.1. FlatList Pagination Process

This is the flowchart of how FlatList with pagination system will be work. After inputting a keyword for example, 'Ketut'. The keyword will be sent to API to check-in database, whether the data is valid or not. If valid, then the data will be retrieved and then stored in the local state of React Native. The stored data before will be processed with FlatList pagination, and when reach end of data FlatList pagination will call a function to check is there still data available for processing. If there is still data available, the data retrieval process will be carried out again and then added to the previously stored data with the limits and offsets that have been added.

```
1. _loadMore = () => {
2. this.setState(
3. (prevState) => ({
4. offset: prevState.offset + 1,
5. }),
6. () => {
7. if (this.state.searchMore == true) this._getData()
8. }
9. )
10.}
```

This is the function code when FlatList item reach the end, and this function will be called to fetch more data if available. In line 3, data is added where prevState is a pre-existing data variable whose offset is added by 1, and if the searchMore variable is true, the _getData() function will be performed which is a function to check and retrieve data from the API.

```
1. <FlatList
2. data={ dataPedagang ? dataPedagang : [] }
3. keyExtractor={ this._keyExtractor }
4. renderItem={ ({ item }) => <ListData noreg={ item.noreg }
   noseri={ item.noseri } namapedagang={ item.namapedagang }
   namacorporate={ item.namacorporate } noregFunction={ () =>
   this._getTagihan(item.idpedagangkios) }/> }
5. onEndReached={ this._loadMore }
6. />
7. _keyExtractor = (item, index) => index.toString()
8. const ListData = (props) => {
9. return(
10.   <TouchableOpacity
11.   style={{
12.   width: metrics.screenWidth,
13.   borderTopWidth: 1,
14.   borderColor: '#EEEEEE',
15.   paddingHorizontal: 10,
16.   paddingVertical: 20
17.   }}
18.   onPress={ props.noregFunction }
19.   >
20.   <View style={{ flexDirection: 'row', justifyContent:
   'space-between' }}>
21.   <Text text={ props.noreg } />
22.   <Text text={ props.noseri } />
23.   </View>
24.   <View style={{ flexDirection: 'row', justifyContent:
   'space-between' }}>
```

```
25.    <Text text={ props.namapedagang } />
26.    <Text text={ props.namacorporate } />
27.    </View>
28.    </TouchableOpacity>
29.    )
30.    }
```

This is the FlatList Pagination code. In Line 2, the data will be checked whether it exists or not, if the dataPedagang is empty then it will be filled by an empty array and if the data is not empty it will be filled with dataPedagang. In Line 5, onEndReached is props that call loadMore function to load more data. In Line 4, renderItem is used by calling the created custom component named ListData. Components in ListData that will display the data on the screen, and how to pass data from ListData to FlatList using props and on FlatList will be called with the example props.noseri={item.noseri}. This is done to make the code look neater, reusable, and can optimize the performance of FlatList because not everything is processed in FlatList but through custom components and functions that have been created previously.

5.2 Testing

For the test, hardware that will be used is Sunmi POS (Point of Sale) P1 4G with the following specification:

- RAM: 1GB
- Storage: 8GB
- Android version: 6.0

The programming language to be used is Javascript (React Native), for testing tool is using Google Chrome Debugger. All tests were carried out with 100% battery condition. Sunmi P1 has the ability up to 60 fps.

1. ScrollView with Array.Map() Test

Table 5.1 : ScrollView Array Map Test

Load Time	**UI FPS**	**JS FPS**	**Frame Drop**	**GPU Process**	**Network Process**
83 second	5 FPS	4 FPS	2786 Frame	HIGH	357.42 ms

Due to the long load time, the test is carried out for 2 minutes. From the data test above, we know that ScrollView load time is bad, less UI & JS FPS, more frame drops, and high GPU process. For the network process, ScrollView with Array.Map() can complete in 357.42 ms. In this test, ScrollView will fetch the data first from API and store it into local store in React Native. Data in ScrollView takes up to 83 seconds to display all data at once on the screen. And if users scroll down, there will be a decrease in fps, an increasing frame drop, the display will look broken and cause lag. The drop frame generated in this test will increase as the test progresses and it is possible to arrive at the last data scroll. Frame drop on Scroll View with Array.Map() is very bad for user experience in using this application or program. UI FPS (Main Thread) and JS FPS (Javascript Thread) on the ScrollView test with Array.Map() are also fairly bad where UI FPS gets 5 FPS while in JS FPS it gets 4 FPS. Because React Native is a native mobile development framework, so many processes use GPU processes instead of CPUs. Therefore, the GPU usage on a ScrollView with Array.Map() is fairly high and has an impact on the performance of the application.

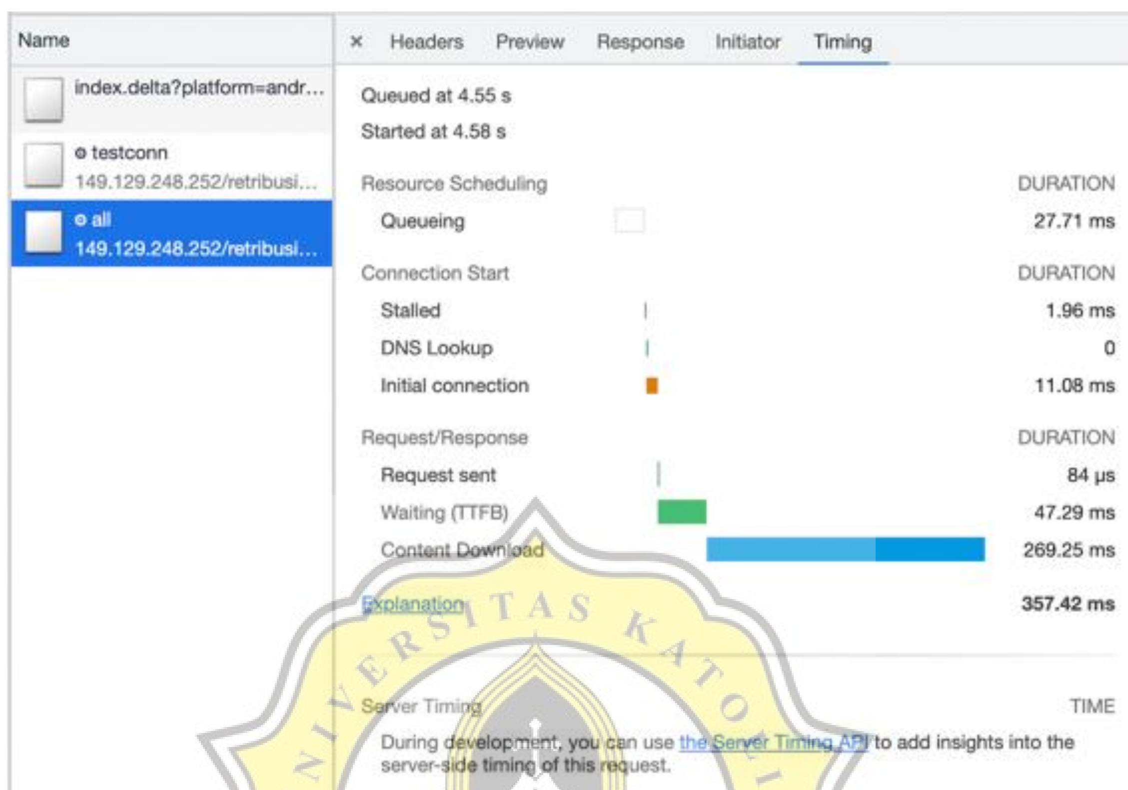


Figure 5.2. ScrollView Array.Map() test Network Process

ScrollView with Array.Map() requires a total time of up to 357.42 ms, where data can be obtained in 269.25 ms which is quite fast. The Waiting section or TTFB stands for Time To First Byte is the time it takes the server to prepare a response to be sent. At this point, Array.Map() takes about 47.29 ms and is pretty good. In the connection section starting a ScrollView with Array.Map() also gets good results where it is stalled or the time the request spends waiting before it can be sent. This time includes time spent negotiating network proxies. In the initial connection section, you get 11.08 ms where the initial connection is the time it takes to connect to the internet and the resulting time is fairly fast. The downloaded content is in the form of JSON which is about 900 KB in size from the API which contains 900-2000 data.

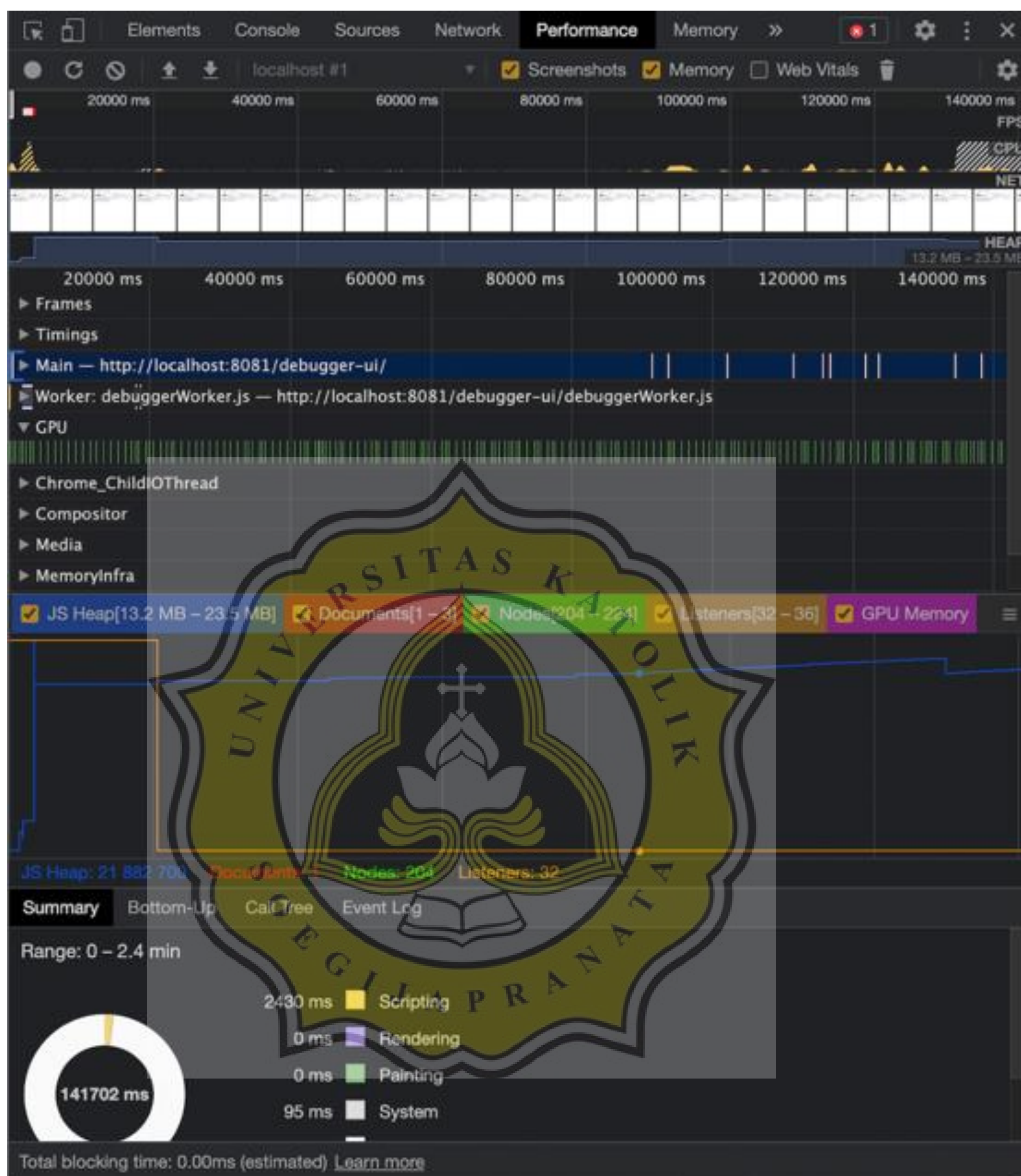


Figure 5.3. ScrollView Array.Map() test snapshot

We can see from the snapshot above, that ScrollView Array Map uses more GPU power, and the JS Heap is high. From 2 minutes test, the heap still high and makes FPS is quite bad & more frame drops. ScrollView uses the GPU process and minimally uses the CPU process, therefore from the above results it can be seen that

the GPU usage is high while the CPU process graph is low. The CPU usage process can be seen in the orange graph and on the summary tab it can be seen that it was a scripting process that took 2430ms or 2.43 seconds. The event listener is high in the beginning because it calls the function to fetch data only once and the data will be stored and then displayed all directly. The heap at first looks small and as time goes on, the heap increases and this indicates that there is a fps drop and abnormal GPU usage in the running application or program. In testing ScrollView using Array.Map() for 2 minutes, the heap has reached 13.2 MB -23.5 MB which is quite large for a memory leak.

2. ScrollView with Lodash test

Table 5.2 : ScrollView with Lodash test

Load Time	UI FPS	JS FPS	Frame Drop	GPU Process	Network Process
80 Second	5 FPS	8 FPS	1924	HIGH	376.37 ms

Same with using Array.Map(), ScrollView with Lodash load time still slow with 80 second, and the UI FPS & JS FPS are not much different from before, the frame drops are less than using Array Map but GPU Process still high. ScrollView with Lodash network process can complete in 376.37 ms which slower than ScrollView Array.Map() test. The UI FPS and JS FPS sections are still pretty bad though slightly higher than the results on the ScrollView test with Array.Map(). The resulting Drop Frame also decreases which results in the number 1924, but it is possible that the number will increase, the same as in a ScrollView with Array.Map(). The process of using GPU in testing here is still the same as before which is still high and will have a bad impact on users of this application later. However, the total network process lost compared to the network process results generated by ScrollView with Array.Map() which got a total time of 357.42 ms.

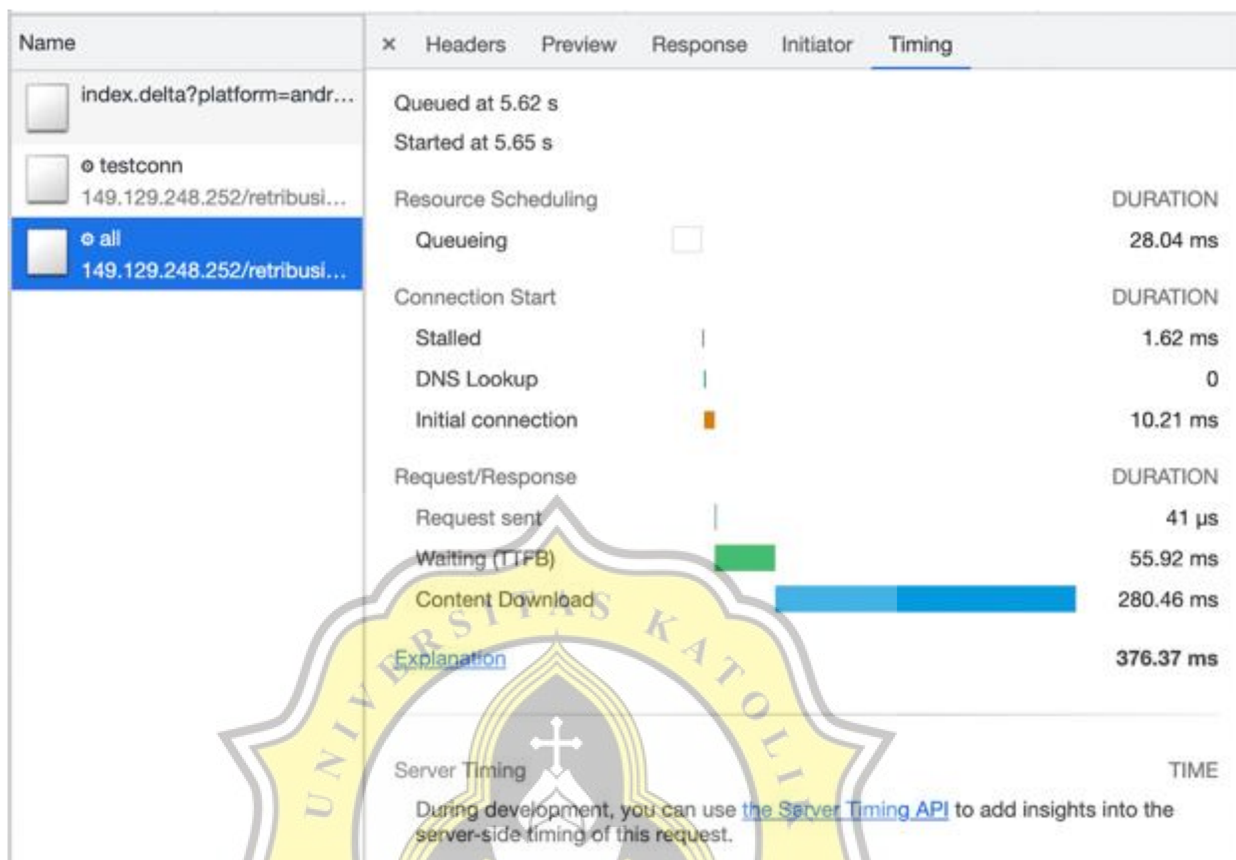


Figure 5.4. ScrollView with Lodash Network Process

In the results of the network process above, ScrollView with Lodash takes a total of 376.37 ms or about 37 seconds to complete all processes. The server prepares the response for 55.92 ms and the data is successfully retrieved within 280.46 ms or for 28 seconds, and the data can be saved and then displayed on the ScrollView. Compared to ScrollView Array.Map(), ScrollView with Lodash gives longer total network results. Initial Connection generated is quite good when compared to ScrollView with Array.Map() with a time of 10.21 ms.

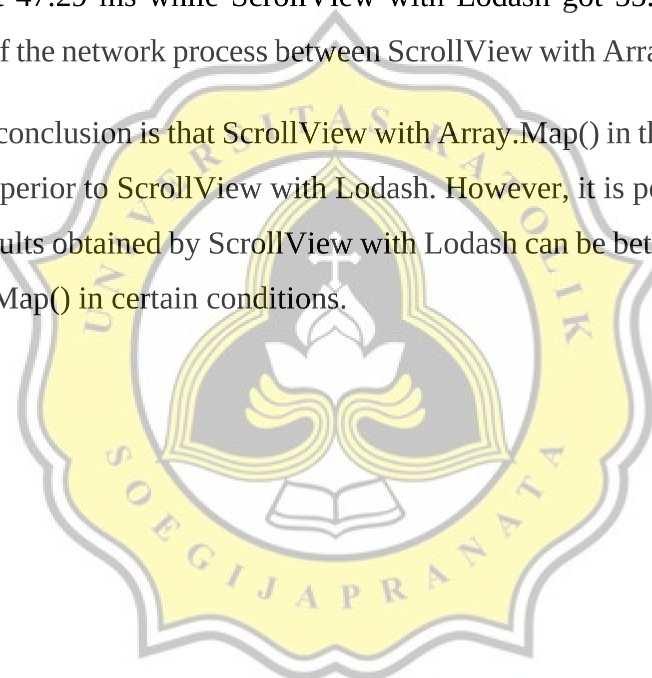
The resulting waiting time is also longer than the results obtained by ScrollView with Array.Map() with 47.29 ms. However, the Request Sent time on a ScrollView with Lodash is better where it gets a time of 41 nanoseconds.

Table 5.3 : Total Network Process time comparison between ScrollView with Array.Map() and ScrollView with Lodash

ScrollView with Array.Map()	ScrollView with Lodash
357.42 ms	376.37 ms

In the table above, it can be seen that ScrollView with Array.Map() gets a better total time than ScrollView with Lodash. The difference between the two is 18.95 ms. In the Waiting section, ScrollView with Array.Map() also got a better total time at 47.29 ms while ScrollView with Lodash got 55.92 ms. Above is a discussion of the network process between ScrollView with Array.Map() and Scroll

The conclusion is that ScrollView with Array.Map() in the Network Process section is superior to ScrollView with Lodash. However, it is possible that in other tests, the results obtained by ScrollView with Lodash can be better than ScrollView with Array.Map() in certain conditions.



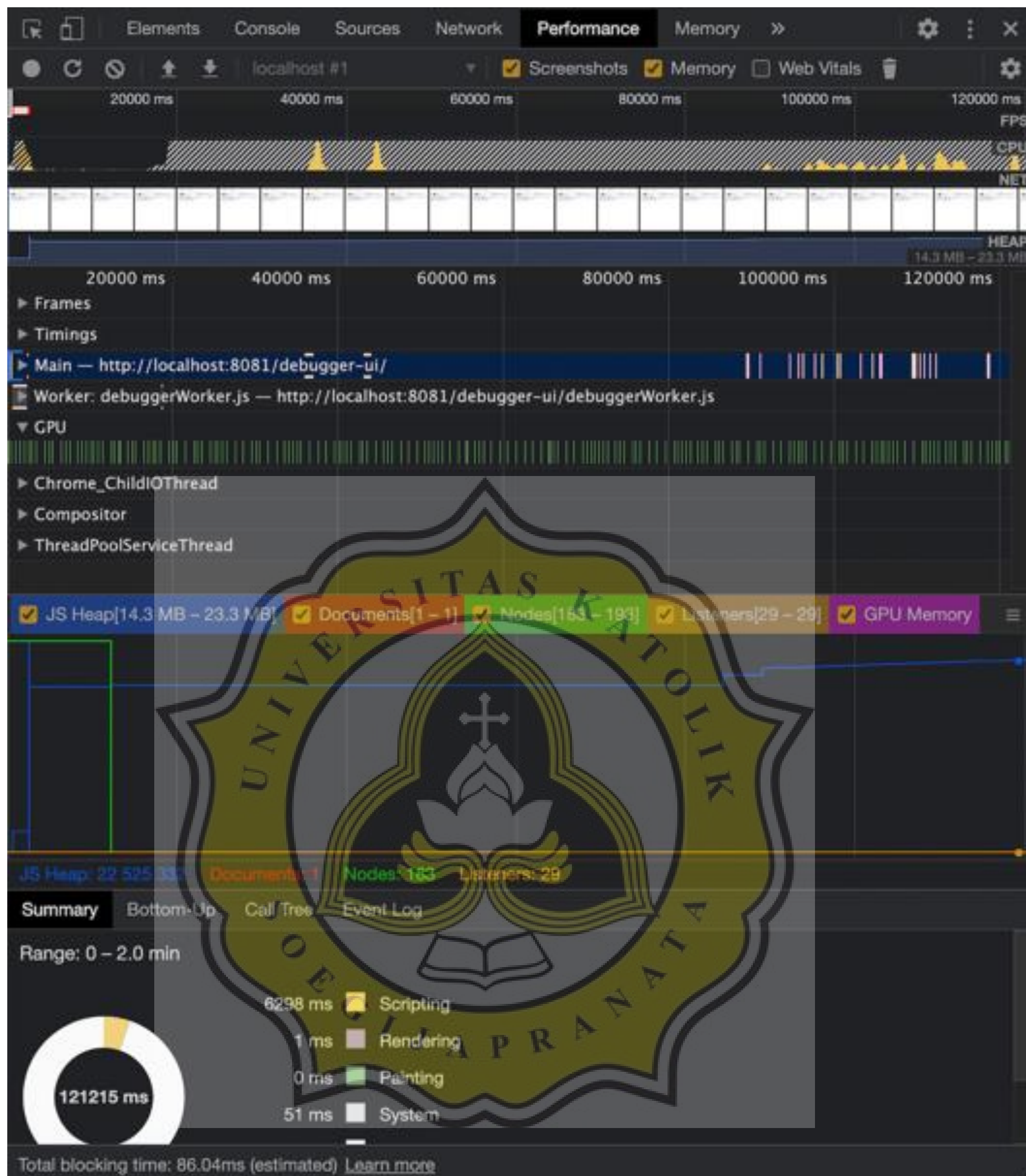


Figure 5.5. ScrollView Lodash test snapshot

From the snapshot above, we notice that ScrollView with Lodash displays a slightly better heap than Array Map. GPU process still high in this test. This test was carried out for 2 minutes, because ScrollView has a load time of 80 seconds. In the Summary section, it can be seen that the scripting process was carried out for

6298 ms or about 6 seconds and 1ms was used for rendering. The results on the snapshot can be seen that the GPU usage is still high similar to ScrollView with Array.Map(). The heap generated by ScrollView with Lodash is a bit smaller but still, it affects the user experience in using the application. Applications with high heap and GPU usage will feel heavy and broken when there is interaction from the user and the bad thing can be a force close on the application due to drastically decreased performance and no more available resources on the system.

3. FlatList test

Now we will test by using FlatList without any customize and we can see the results below.

Table 5.4 : FlatList test

Load Time	UI FPS	JS FPS	Frame Drop	GPU Process	Network Process
2 Second	58 FPS	56 FPS	229	NORMAL	361.00 ms

Load time on the FlatList is very fast, with 2 seconds and the data can be displayed. UI FPS & JS FPS result are good, which is almost 60 FPS. GPU process by using FlatList is normal and not too high like a ScrollView which means FlatList is indeed good in performance in loading data on a large scale. Load time yang cepat pada FlatList sangat membantu dalam mengoptimalkan kinerja React Native dan pengalaman user dalam menggunakan aplikasi tersebut juga merasa nyaman. Bagian UI FPS dan JS FPS mendapatkan angka 58 FPS pada UI FPS dan 56 FPS pada JS FPS dan tidak menutup kemungkinan dapat mendapatkan 60 FPS baik pada UI FPS maupun JS FPS. Frame Drop pada FlatList terbilang rendah, dan bisa dilihat pada hasil pengetesan didapatkan bahwa FlatList mendapatkan 229 Frame Drop. Hasil yang diberikan bisa terbilang bagus dan tidak terlalu mengganggu pengguna

saat menggunakan aplikasi tersebut. Penggunaan GPU pada FlatList juga terbilang normal sehingga performa aplikasi tetap stabil dan terjaga.

In network process section, FlatList need 361.00 ms to complete the request and displayed the data.

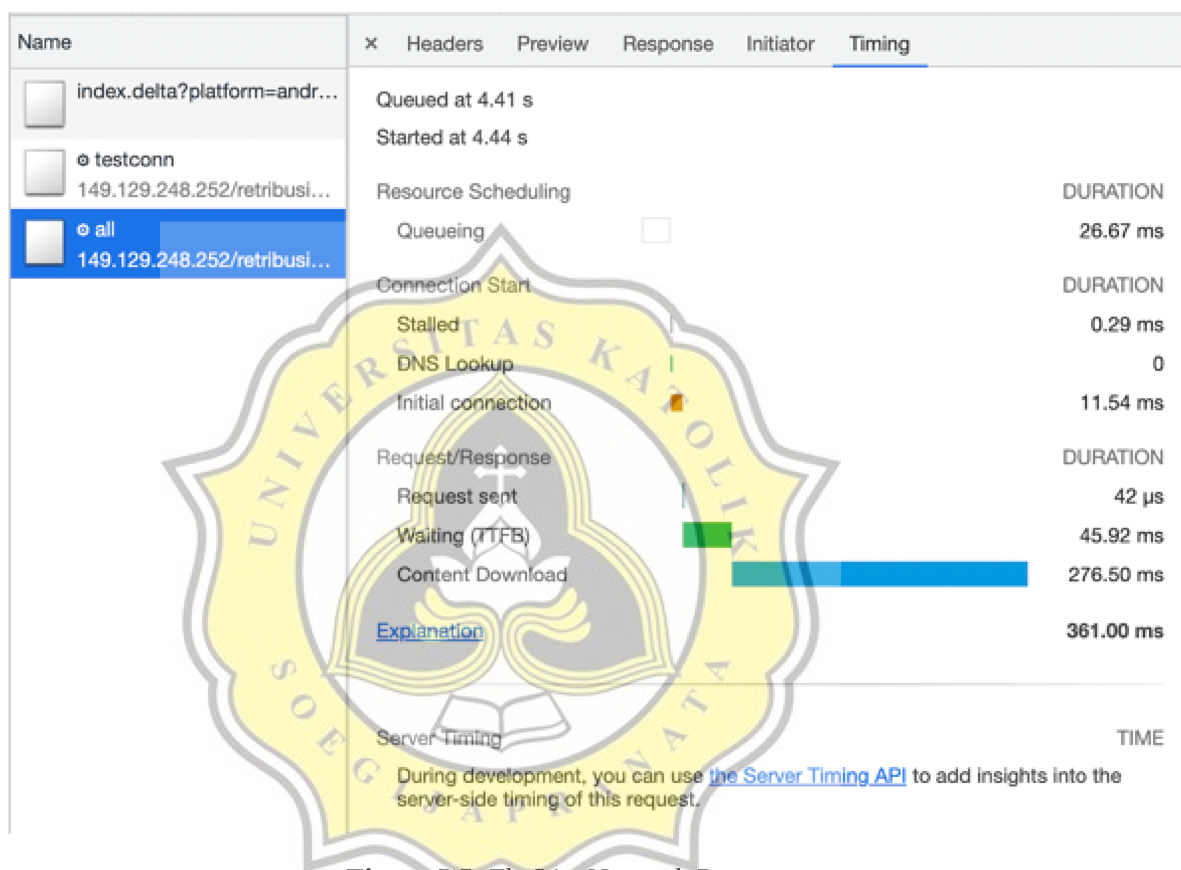


Figure 5.5: FlatList Network Process

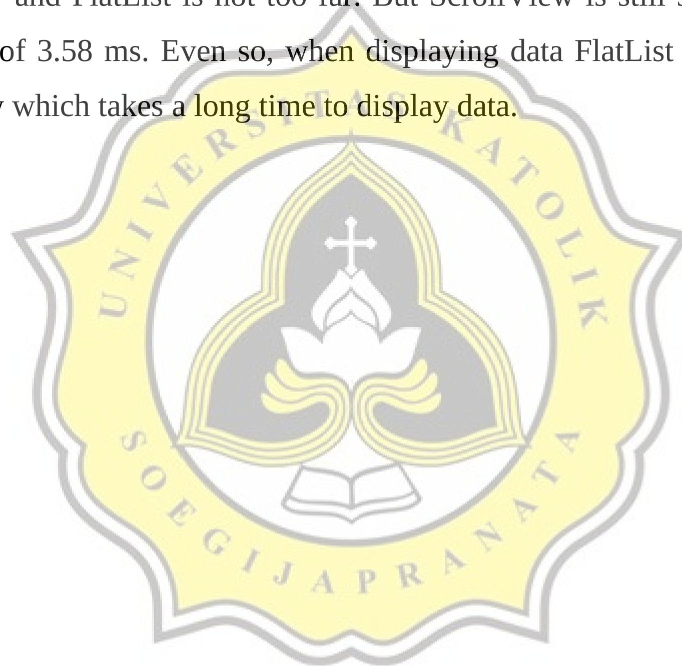
The FlatList in this section takes up to 361.00 ms or 0.361 seconds to complete the process. In the Waiting section, it can be seen that the time it takes for the server to prepare a response is 45.92 ms and the data is successfully retrieved within 276.50 ms or 0.2765 seconds. The total time that FlatList takes is quite a bit longer compared to the total time it takes ScrollView with Array.Map(). On the

Network Process, FlatList gets an Initial Connection time of 11.54 ms. Requests are sent in a short time, with 42 nanoseconds.

Table 5.5 : Total Network Process time comparison between ScrollView and FlatList

ScrollView Array.Map()	FlatList
357.42 ms	361.00 ms

It can be seen in the table above if the difference in total time between ScrollView and FlatList is not too far. But ScrollView is still superior here by a difference of 3.58 ms. Even so, when displaying data FlatList is still better than ScrollView which takes a long time to display data.



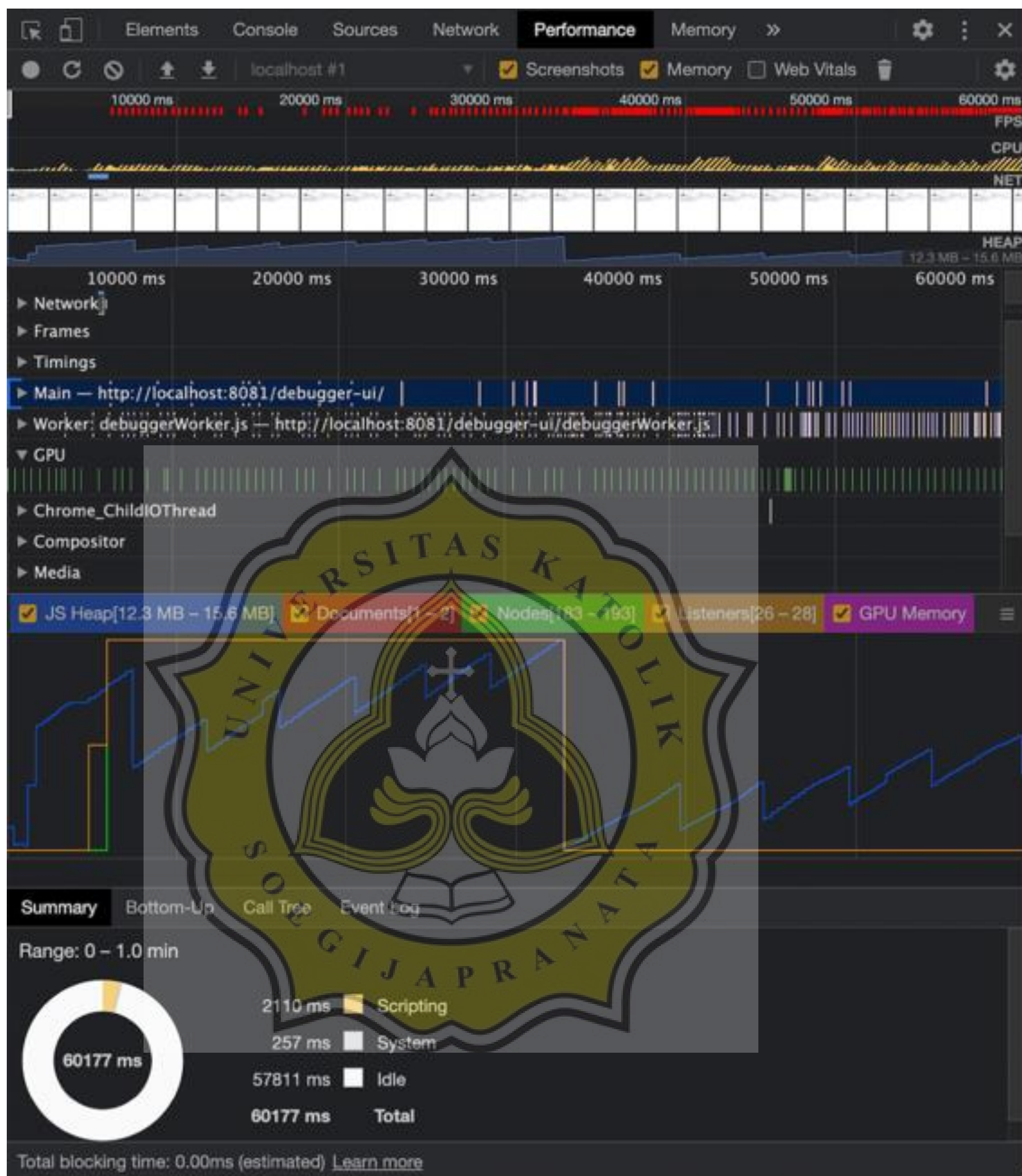


Figure 5.6. FlatList test snapshot

From the snapshot above, we can see that FlatList have a better performance than ScrollView. GPU process in FlatList is normal and not high as ScrollView. FlatList load time is fast with 2 second and data can be displayed to user. The heap

high at first, and low after a few moments. The CPU process graph in the FlatList test is fairly good and not high, so FlatList can run well when the data is displayed and there is interaction with the user when scrolling the list. In the Summary section below, it can also be seen that with a total time of 60177 ms or 1 minute, the scripting process (Javascript process) only takes 2110 ms or 2.11 seconds and the data is already displayed on the Sunmi P1 screen and is ready to be used.

4. FlatList Pagination Test

In this test, we will use 'KETUT' as the keyword, that we will send to API to fetch the data of Banjar market traders with contain name Ketut. And here is the result below.

Table 5.6 : FlatList Pagination Test

Load Time	UI FPS	JS FPS	Frame Drop	GPU Process	Network Process
1 second	60	56	77	NORMAL	68.21 ms

The test was carried out for 1 minute and it was found that the FlatList pagination can load data in 1 second and the data can be displayed. UI FPS & JS FPS results in this test are very good, where FlatList can display data with good fps and few frame drops. GPU process is normal, same as FlatList in standard mode because FlatList can optimize performance & memory usage. FlatList with pagination can complete in 68.21 ms to display all data in page 1.

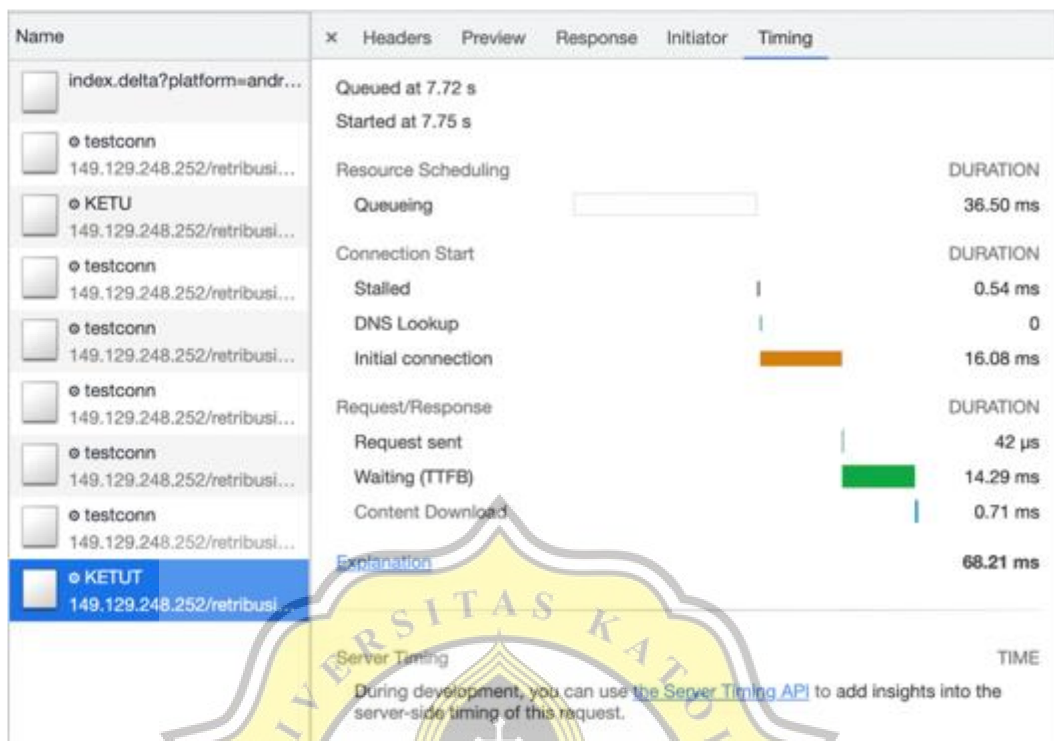


Figure 5.7. FlatList Pagination Network Process

FlatList with pagination system only takes a total of 68.21 ms which is very fast. In the Waiting section, where the server prepares a response, it takes 14.29 ms due to sending parameters and checking data on the server. However, the data was obtained in a short time of 0.71 ms. Compared to regular FlatList, FlatList with pagination system gets much better results.

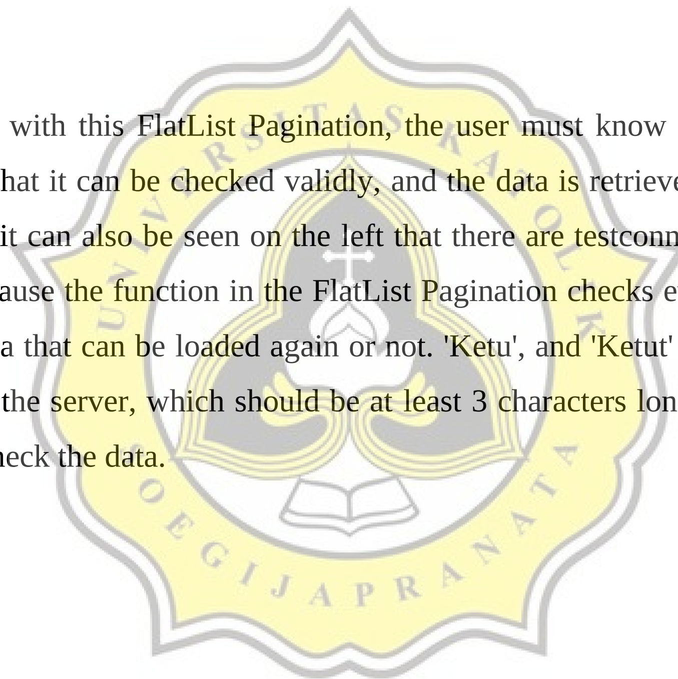
Table 5.7 : Total Network Process time comparison between FlatList and FlatList Pagination

FlatList	FlatList Pagination
361.00 ms	68.21 ms

It can be seen that the result of FlatList Pagination is much better than normal FlatList. This is very influential on the application's network traffic, which

can save network bandwidth. The results of FlatList Pagination can be better because FlatList Pagination sends keywords to the server which will later be checked by the server, and that's why the Waiting part of FlatList Pagination takes 14.29 ms. After the keywords are sent and checked, the data will be sent by the server according to the amount of valid data with the keywords that have been sent. For example, the keyword sent is 'Ketut' then the server will check in the database whether a trader with a name that includes 'Ketut' exists or not. If there is, then all data containing the keyword 'Ketut' is 356, then 356 data will be sent back to React Native.

But with this FlatList Pagination, the user must know what keywords to submit so that it can be checked validly, and the data is retrieved successfully. In Figure 5.7 it can also be seen on the left that there are testconn, Ketu, and Ketut. This is because the function in the FlatList Pagination checks every time whether there is data that can be loaded again or not. 'Ketu', and 'Ketut' are keywords that are sent to the server, which should be at least 3 characters long to be sent to the server to check the data.


UNIVERSITAS KATOLIK SOEGIJAPRANATA

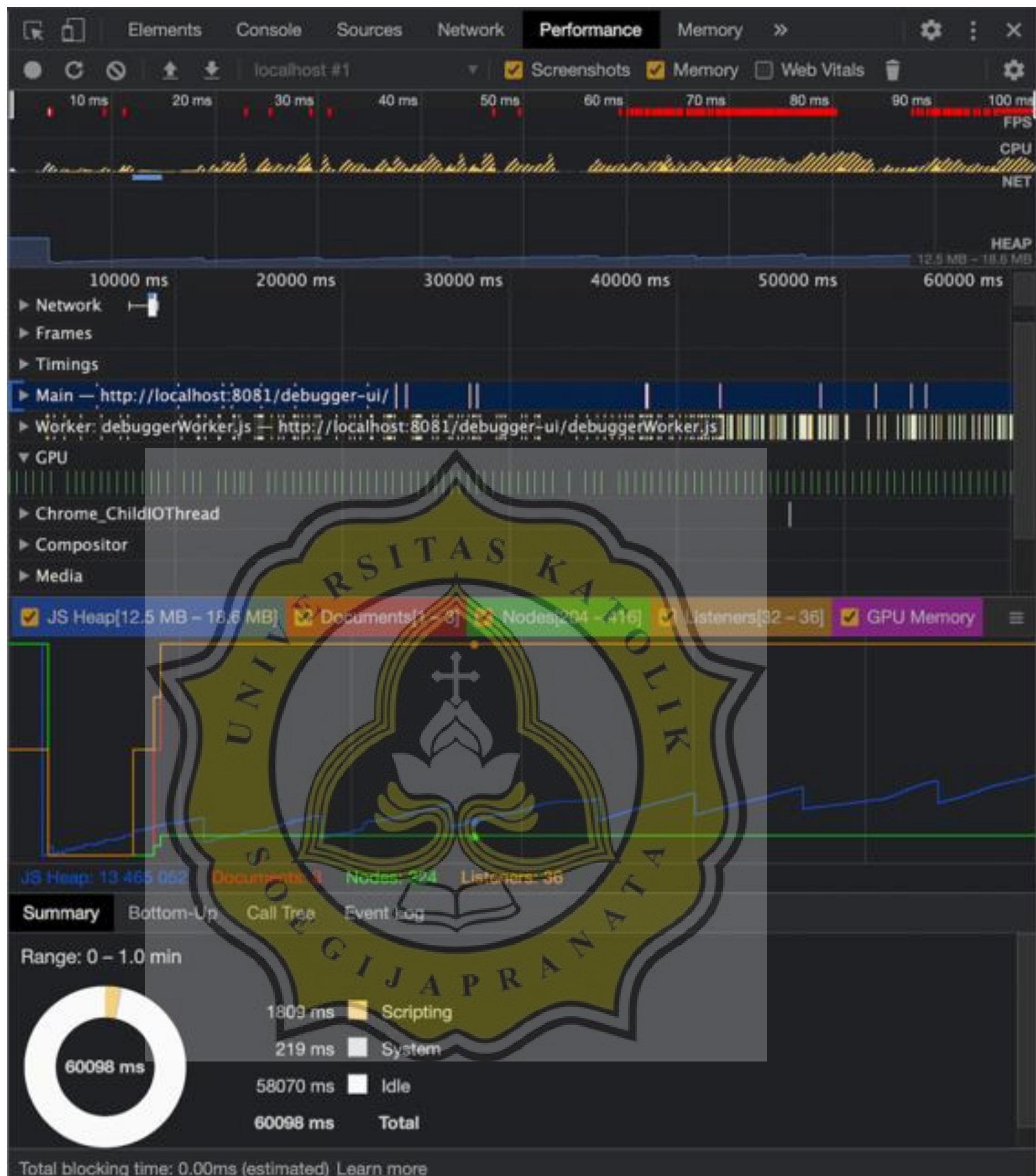


Figure 5.8. FlatList Pagination snapshot

The snapshot result is very good. Heap results on this test are very good, compared to previous tests, this heap in this test is low. For the GPU process is slightly better than normal FlatList, which mean it's good for better performance. The CPU process graph in the snapshot above can be said to be good because the

graph is not too high. This good performance can occur because the FlatList pagination limits the number of items displayed, so that the process can be carried out quickly and optimally by the system which can reduce CPU & GPU performance, keep fps high, reduce fps drop, and maximize load time speed.

The following is a summary of all the tests that have been carried out

Table 5.8: Summary Table

	Load Time	**UI FPS**	**JS FPS**	**Frame Drop**	**GPU Process**	**Network Process**
ScrollView Array Map	83 second	5 FPS	4 FPS	2786	HIGH	357.42 ms
ScrollView Lodash	80 second	5 FPS	8 FPS	1924	HIGH	376.37 ms
FlatList	2 second	58 FPS	56 FPS	229	NORMAL	361.00 ms
FlatList Pagination	1 second	60 FPS	56 FPS	77	NORMAL	68.21 ms

Based on the testing and analysis above, FlatList is better to use than ScrollView because the FlatList in the above test gets fast load times, good UI FPS & JS FPS numbers, and normal GPU usage. ScrollView is not recommended for processing data on a large scale because it can result in high frame drops, large GPU usage, and low fps.

Figure 5.9. Network Process Summary Table

	Waiting (TTFB)	**Content Download**	**Total Time**
ScrollView Array.Map()	47.29 ms	269.25 ms	357.42 ms
ScrollView Lodash	55.92 ms	280.46 ms	376.37 ms
FlatList	45.92 ms	276.50 ms	361.00 ms

FlatList Pagination	14.29 ms	0.71 ms	68.21 ms

From the table above, it can be seen that FlatList Pagination got very good results, due to the short Waiting (TTFB) result with 14.29 ms time, short content download time with 280.46 ms, and the total time of 68.21 ms. However, in this part of the Network Process, when compared between normal Flatlist on ScrollView, ScrollView Array.Map() gets quite good results than normal FlatList.