UMEÅ UNIVERSITY

PERFORMANCE COMPARISON BETWEEN REACT NATIVE AND FLUTTER

Jakub Jagiełło

Bachelor Thesis, 15 credits
Computing science

2019

Abstract

The global mobile OS market share is mainly dominated by two operative systems. One of them is Android, and the other one called iOS. These two together had more than 99% of the market share in 2018, and it does not look like this trend is going to change anytime soon. Because of these two completely different platforms, there are many frameworks that provide solutions for developing applications that are platform independent. This study contains performance comparison between two of the more popular frameworks called Flutter and React Native. The performance is measured in terms of the number of dropped frames under a certain time for two different applications. The results that were acquired during tests reveal that React Native dropped less frames in total, although the difference was not significant.

Contents

1 Introduction

The global mobile OS market share is mainly dominated by two operative systems. One of them is Android made by Google, and the other one called iOS made by Apple. These two together had more than 99% of the market share in 2018, according to statista[11]. Because of the differences between these two operative systems, developing native applications have to be done separately. That makes development and maintenance of the same application potentially more expensive and time-consuming. For that exact reason, cross-platform solutions like React Native were developed. React Native offers the tools to develop one single application that can be installed and ran on either Android or iOS without any hassle. Recently a new framework called Flutter was released to bring fresh ideas and tools to create cross-platform applications. Both React Native and Flutter promise to be equally fast as a native application, but since the latter framework is relatively new to the game, there is not enough performance comparison between these two.

1.1 Purpose and research questions

The purpose of this research is to examine if there is any significant difference between these two frameworks in terms of performance. Performance is one of the crucial attributes that have to be considered when implementing new applications. That is why it should be preferable to choose a framework that offers the best performance.

I hypothesize that Flutter will have better performance in terms of dropping less total frames under a certain time. My hypothesis is based on the fact that Flutter is a newer framework, and they had the ability to chose any programming language that suited best their needs, while React Native were stuck with using JavaScript. Also, Flutter could possibly learn of any mistakes that React Native have done.

1.2 Delimitation

Both Flutter and React Native are cross-platform frameworks, which means that the comparison could be done on Android and iOS-based smartphone. In this study, the tests are done only on the more common operative system Android.

The frameworks run their respective applications in complete different way. There are major differnces in compilations and communication with the underlying system. The focus for this comparison will be to look into how effectively the frameworks is sustaining the desired 60 frames per second in different scenarios.

2 Related work

There exist previously published works that have done similar comparisons. One of them is a paper that besides its case study of the modularity, routing and the visual appealing of syntax, also covers brief performance comparison. The comparison was done first by implementing identical lists by the respective frameworks and then measuring dropped frames while scrolling through the list[12]. For that specific test, Flutter had a slight advantage in relation to less dropped frames. The second test was about measuring the I/O speed for each

application, and this time, React Native had an advantage.

Niclas Hannson and Tomas Vidhall analyzed the performance between applications written in React Native and a pure native Android implementation [10]. The authors implemented the same applications in React Native and natively for Android, then measured performance and analyzed it. In terms of number of dropped frames, the React Native application performed equally good, and sometimes even better than the native version of it. Although the memory footprint and application size was much greater.

3 Background

This chapter provides brief explanation of different aspects to then be able to understand the results presented in this study.

3.1 What is Flutter and React Native

Flutter [2] and React Native [9] are names for two different open-source frameworks that share the same purpose. The first one is made by Google and the second by Facebook. The two offer a way for creating cross-platform mobile applications by giving the developer tools for writing a single application in a predetermined way and language and then enable to run it on either Android or iOS.

The construction of an application in either of these frameworks is done by structuring data and logic in components that are allowed to have some sort of inner state. The components form a hierarchy based on composition. Each component is responsible for drawing itself on the screen. One simple example to better understand it is to imagine a list with clickable buttons. The list is now a parent component with child components in the form of clickable buttons. The parent does not have to know what the children look like, only where they will be located on the screen. That can be done by passing down constraints to the buttons so that they can draw themselves within the inherited constraints.

Flutter and React Native also share other similarities like supporting the reactive programming paradigm. Its main purpose is to solve the common struggle with handling data updates, which makes it easier to track the updates of the applications inner state.

3.2 React Native

React Native is written in a mixture of JavaScript and JSX. It started as an extension to the already existing library React for the web.

JavaScript thread

A JavaScript thread executes the code that was written for your application in JSX and JavaScript. This thread feeds the visual updates to the UI thread, and any long executing tasks will turn into noticeble dropped frames on the UI side.

UI thread

UI thread acts as the main thread for the framework. It has access to the native UI equivalents and handles the communication between framework and operative system. If this thread is slowed by heavy computation, it will most likely result in a dropped frame [5].

3.3 Flutter

Flutter is written in a programming language called Dart. The choice of that language is mainly due to the ahead-of-time compilation and a few preferences like having interfaces and abstract classes. The main principle behind Flutter is that the UI is built only out of widgets

which describe what their view should look like given their current state.

UI thread

Although this thread is called the same as the React Natives thread, it should not be treated the same because they do completely different things. The main purpose of the UI thread is to execute Flutter's framework and the application code written in Dart. All of this work is done in Dart VM. When a Flutter application creates a scene to display, the UI thread creates a layer tree and sends it to the GPU on the device to render it. This thread should not be used for any heavy computations because it will yield in lower FPS [4].

GPU thread

The GPU thread cannot be accessed directly by the code that you write for you application in Dart. It is ran by the framework itself and its main purpose is to recieve the layer tree sent by UI thread and display it by talking to the GPU. If this thread is slow, it is caused by something written in the Dart code.

3.4 How are Flutter and React Native translated into native applications

When an application written in Flutter or React Native is run on either iOS or Android, it has to somehow handle the communication with the operative system. In React Native's case for the Android, it is done through a bridge. The UI elements declared in the React Native application are compiled into their native equivalents, meaning that all the available components have a direct native correspondent. The application is then run on a separate JavaScript thread. When interacting with the application, the communication between React Native and the native components is done through that bridge.

Flutter has taken a different approach for their communication with the underlying native system. Instead of having its UI components correspond to native equivalents, they use their own rendering engine. The Flutter engine is compiled with Android's NDK, and the Dart code is compiled into native code.

3.5 Performance measure

The smoothness of an application is defined by how much frames per second it can sustain. If an application running at 60 FPS suddenly drops down to 30 FPS during an animation, it will be noticed. To prevent that from happening, we do not want to block the thread that runs our application. The thread can be slowed down by either the application specific code or the code that has to be executed by the framework in order to make our application work. To be able to deliver 60 frames per second, the thread executing your application needs to fit the execution of your application in approximately 16 milliseconds. If your code included a heavy computation that took for example 160 milliseconds, that would result in 10 frames dropped each second, making your application appear to stutter. Besides our own code, we do not have too much control over how much work is done in the background by the framework. In case of simple short task where we would expect the application to run at full speed, the framework can still suffer from dropped frames if it is poorly optimized.

3.6 State and rendering

Each Component in React Native and Widget in Flutter can have a single state. That state can contain anything that we want to be able to store. For example, an input field could store the written input saved dynamically whenever a character was added or removed. Each time a state is changed in either framework, the element that holds the state marks itself as dirty. Marking itself dirty tells the framework that the specific element has changed and needs to be re-rendered. When no changes in states have occurred, the framework is basically idle. With that in mind, we can have total control of how often the framework has to work and re-render itself.

4 Method

To carry through the performance comparison, I have taken a similar approach to the one in Wu's work [12]. Applications with similar functionality were implemented using Flutter and React Native. The purpose of these applications is to stress test the frameworks by doing many re-render calls. The measurement will focus on counting the total amount of frames that was dropped while running a testing application under 30 seconds. The tests will be done with the different frequency of the re-render calls which will be set with the help of a time internal. The two applications that the tests will be based on are described below. This method was chosen because it allows to have control over exacly how often re-render calls are made.

4.1 Application 1

The first application consist of a single component named counter, see figure 1. Counter holds a single number that is initiated upon creation to 0 and then within the component, an interval is set for changing its internal state in form of increasing the number by one. The component manifests itself on the screen as a text on white background that shows its current number. The interval initiation is presented for respective framework in listing 1 and 2. When a setState function is called, the framework knows that a change has occurred within that component and calls for a re-render[7][8]. The purpose of this application is to test the simplest form of implementation and measure at which rate of re-rendering frames begin to drop.

```
1 Timer.periodic (Duration(milliseconds : widget.milliseconds), (Timer t){
2     setState (() {
3         counter += 1;
4     });
5 });
6 }
```

Listing 1: Interval initiation for Flutter

```
1 setInterval (() =>
2     this.setState(previous => (
3         {counter : previous.counter + 1}
4 )), this.state.milliseconds);
```

Listing 2: Interval initiation for React Native

4.2 Application 2

The second application is an extension of the previous one. The main difference is that the main component consists of two container components. Each component holds three counters as shown in figure 2. The purpose of this application is to give the frameworks a chance to optimize some computations. First of all, there is a total of 6 exactly the same components, so there is a potential to reuse the same objects. Also, the right side of the component tree is being updated two times as much. It means that if the framework does not have to recompute all of its components when only the left side of the component tree is being updated. This can potentially yield great performance loss if any of the frameworks do not do it efficiently.

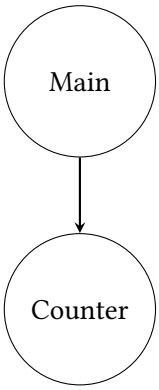

Figure 1: Component tree visualization for application 1

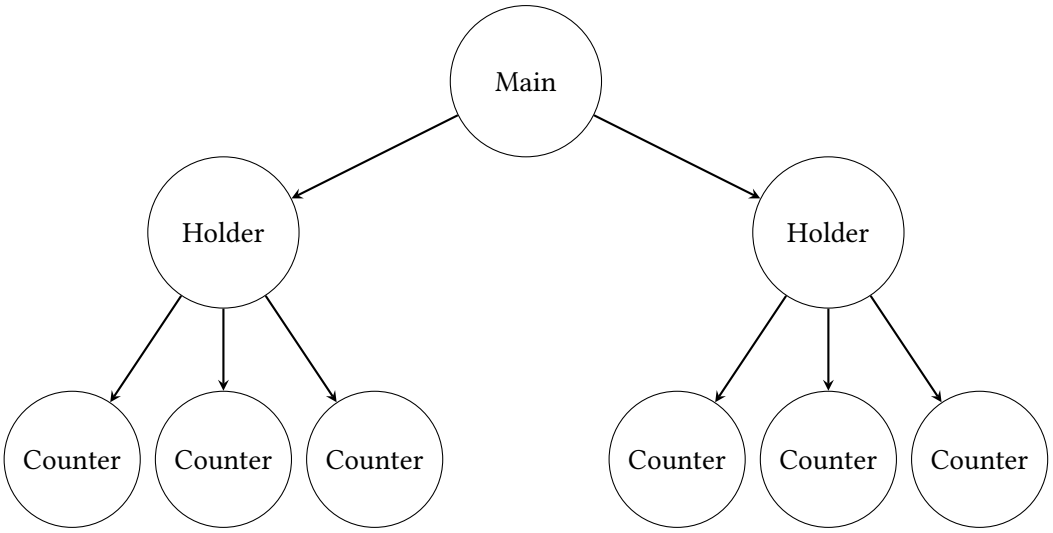

Figure 2: Component tree visualization for application 2

4.3 Data collection

Application 1 and 2 are run on a real physical device, which is recommended by both thre creators of React Native and Flutter[4][5]. The reasoning behind not using an emulator is that they do not use the same hardware, which yields in some operations being either unrealistically fast or slow.

The applications written in React Native are run through expo[1] and measured with its performance monitor. The performance monitor only allows measuring dropped frames with debug mode on. The performance of Flutter application is measured with by their own profiler, which is included in the framework as a class called PerfromanceOverlay[3]. The overlay is divided into two time series, upper layer shows how much time was required to produce each frame and the lower layer shows how much time was required on the GPU thread to produce each frame. Unlike expo, this performance monitor is not limited to debug mode only, but it also allows to measure it while being in production mode.

4.4 Data analysis

The performance monitors for Flutter and React Native allow to look into how long every frame took to render. If a frame took longer than 16 milliseconds to be rendered, then we can be certain that it got dropped. Additionally an average time for rendering frames can be taken so that the difference between a dropped frame and a succeeded frame can be compared and analyzed.

4.5 Test environment

The tests of the applications were run on a real device [6]. All the background applications were suspended from running to prevent unnecessary workload on the CPU. The applications itself were restricted to use only the frameworks own functions and modules, and not contain any third-party libraries.

5 Results

Results presented in this chapter are produced by running application 1 and 2. The results in each table below are divided into columns of different time intervals. Each time interval in form of milliseconds stands for how often re-rendering happened in the application. The tables also contain information about how many frames were dropped while running and what was the average render time for each frame. Each application with specific time interval was run 10 times with the length of 30 seconds each. The device normally produces 1800 frames under 30 seconds.

5.1 Results for React Native

Results for running application 1 and 2 by React Native can be seen in respective table 1 and 2.

Table 1 Results for application 1

milliseconds	128	64	32	16
dropped frames	13	8	6	5
render time	10	10	8	8

Table 2 Results for application 2

milliseconds	128	64	32	16
dropped frames	14	9	6	5
render time	10	10	8	8

5.2 Results for Flutter

Tables 3 to 6 contain the results for running application 1 and 2 in both debug and profile mode.

Tables 3 and 5 contain the results for application 1 and 2 that were run in debug mode. Tables 4 and 6 contain additional results for application 1 and 2 that were run in profile mode. The purpose of additional runs in profile mode was to investigate the impact that the debug mode has on the results.

Table 3 Results for application 1 in debug mode

milliseconds	128	64	32	16
dropped frames	14	10	9	6
render time	10	10	8	8

Table 4 Results for application 1 in profile mode

milliseconds	128	64	32	16
dropped frames	13	10	8	5
render time	10	10	8	8

Table 5 Results for application 2 in debug mode				
milliseconds	128	64	32	16
dropped frames	14	11	9	7
render time	10	10	8	8

Table 6 Results for application 2 in profile mode				
milliseconds	128	64	32	16
dropped frames	13	9	8	5
render time	10	10	8	8

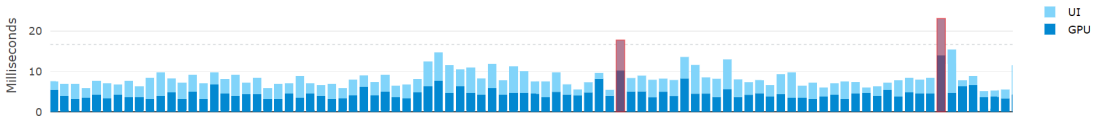

Figure 3: Frame timeline for Flutter

5.3 Closer look into a single frame

Figure 4 shows a time comparison between two frames that were rendered. These two frames are taken from the same timeline in Figure 3 ,one of them is considered as dropped frame and the other one fits under the timelimit of 16 milliseconds. The interesting thing to observe here is the difference between these two frames which is approximately 20 milliseconds. The average render time for this particular test case for frames were 10 milliseconds, meaning that the dropped frame took more than double the time.

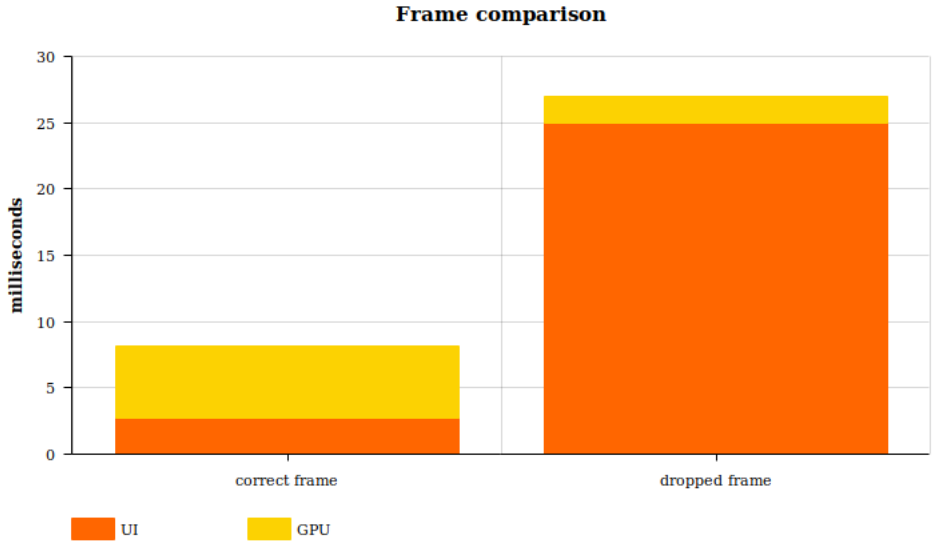

Figure 4: Time difference between correctly rendered frame and a frame that was dropped.

6 Discussion

This chapter discusses the results that were acquired during test runs with application 1 and 2. The total amount of dropped frames for application 1 for both frameworks in tables 1 and 3 do not differ by that much. Judging by the results, we could conclude that for this particular test React Native framework performed better. Although in the column with time interval set to 16 milliseconds, when put in the context of the total 1800 frames render under the testing time, the difference becomes less significant. The same goes for tables 2 and 5 where once again React Native has an advantage however, taking the number of frames produced under the testing time into account, the advantage becomes less impressive.

My hypothesis prior running my tests was that the Flutter would perform significantly better than React Native. That guess was based on the fact which is mentioned in the background section, Flutter does not have to communicate with the underlying system through a bridge. The results that I acquired disproved that, but I think that the results could be closer to my guess if the testing applications were more complex. As Wu found out in his study [12] when both frameworks were tested with large lists where Flutter could sustain much higher frame rate.

6.1 Additional test run for Flutter

Each framework was tested in debug mode but only Flutter was additionally tested in production mode. The debug mode adds more overhead to the application and is not the preferable way when measuring the performance of an application. The reason behind running in debug mode anyway is that React Native application was compiled and run through expo. That made it impossible to measure the total number of dropped frames in any other mode. THe results in tables 3 and 5 show worse performance in terms of dropped frames when compared to tables 4 and 6. However, the difference is minimal when compared to the total number of frames produced. Applications 1 and 2 are most likely too small and simple to be impacted by the debug mode in a negative way. That is why we can also assume that React Native production mode of application 1 and 2 would not show any significant advantage in terms of less dropped frames.

6.2 Fewer frames dropped with heavier workload

There is a trend that follows through all the result presented in tables 1–6. It would be natural to think that the number of dropped frames would increase with increasing computational load in the form of more render calls, but as the tables show, that is not the case. Instead, the lowest amount of dropped frames occurred when there was a re-render call every 16 milliseconds. One of the possible explanation for that is the CPU in my testing device has eight cores; four cores clocked at 2.1 GHz, and the other four at 1.5GHz. The cores have different clock speed for the purpose of running less demanding computations on the lower clocked cores and more demanding computations on higher clocked cores. When applications 1 and 2 were running with least re-render calls, they might have fallen into the category of being less computational demanding, and received the lower clocked core because of that.

6.3 What caused dropped frames

The results in tables 1–6 contain a small number of dropped frames in all the tested time intervals. The fact that there are any dropped frames at all does not necessarely mean that the fault lies in the framework. For each re-render that happened on the device, the framework had to do exactly the same computation. That is why one might wonder why there are occasionally frames that take drastically longer time, like in timeline in Figure 3. The difference between a frame that is considered as dropped and a frame that was rendered under the average time in Figure 4 is approximately the magnitued of 3. It is strange that the UI thread that executes Dart code for application takes sometimes 10 times longer time even though the code to execute is exactly the same.

There can be a few reasons for this occurrence to happen. One of them is a random noise that happens in the background of the testing devices. Even though the testing device had its background applications completely turned off, there is nothing that can be done to have control over how the Android operative system decides how to schedule the workload. That is why the few dropped frames that were detected under tests could be caused by the Android OS stalling the execution of the application and do its own work, which would explain the long sporadic rendering times.

6.4 Lack of a standardized way of testing

Because of the lack of a standardized way of testing the dropped frames by applications, the results that were acquired in this study were done by two different profiling tools. By testing each framework individually with their own profiling tool makes the reliability of the results highly dependent on how objective their tools are.

6.5 Conclusion and future work

The results that I have produced in this study are the exact opposite to what Wu [12] got in his study. According to his results, Flutter framework performed better than React Native in terms of number of frames dropped, when benchmarked with applications that contained very large lists.

As discussed above the fact that the testing applications dropped frames at all could be caused by the random work by the underlying operative system. If there is any interest in further investigation, then the same applications that were used in this study could also be used in future work. The results acquired could be used for comparison if any standardized way of testing dropped frames that will treat the applications equally becomes available.

The future for cross platform mobile frameworks looks bright. The study that did a performance comparison between React Native and a native implementation for Android demonstrated almost equally good performance between them [10]. Considering that, it becomes less beneficial to have separate native implementations for different mobile operative systems. Even though this and [12] study showed completely opposite results, the overall differences in terms of performance are not that significant, meaning that the choice between such cross platform frameworks is also good.

References

[1] Expo for react native. https://docs.expo.io/versions/v32.0.0/ (visited 2019-05-26).

[2] Flutter framework. https://flutter.dev/ (visited 2019-05-26).

[3] Performanceoverlay for flutter. https://api.flutter.dev/flutter/widgets/PerformanceOverlay-class.html (visited 2019-05-26).

[4] Recommendations for profiling flutter applications. https://flutter.dev/docs/testing/ui-performance (visited 2019-05-26).

[5] Recommendations for profiling react native applications. https://facebook.github.io/react-native/docs/performance (visited 2019-05-26).

[6] Samsung s6. https://www.samsung.com/global/galaxy/galaxys6/galaxy-s6/ (visited 2019-05-26).

[7] Setstate function for flutter. https://api.flutter.dev/flutter/widgets/State/setState.html (visited 2019-05-26).

[8] Setstate function for react native. https://facebook.github.io/react-native/docs/state (visited 2019-05-26).

[9] React Native. React native framework. https://facebook.github.io/react-native/ (visited 2019-05-26).

[10] Tomas Vidhall Niclas Hansson. Effects on performance and usability for cross-platform applicationdevelopment using react native. 2016-06-16.

[11] Statista. Mobile os market share 2018. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/ (visited 2019-05-26).

[12] Wenhao Wu. React native vs flutter, cross-platforms mobile application frameworks. 2018-03-01.

A Appendix

B Implementation of application 1

```
import 'package: flutter / material . dart ';
import 'dart : async';

void main() => runApp(App());

class App extends StatelessWidget {
  final int milliseconds = 16;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
     home: Scaffold (
         body : Center(
           child : Counter( milliseconds : this . milliseconds ) ,
         ) ,
       ) ,
    ) ;
  }
}

class Counter extends StatefulWidget {
  final int milliseconds ;

  Counter({Key key, this . milliseconds }) : super(key : key);

  @override
  CounterState createState () => CounterState() ;
}

class CounterState extends State<Counter> {
  int counter = 0;

  @override
  Widget build(BuildContext context) {
    return Text("$counter") ;
  }

  @override
  void initState (){
    super. initState () ;
    Timer. periodic (Duration( milliseconds : widget. milliseconds ) , (Timer t) {
      setState (() {
       counter += 1;
      }) ;
    }) ;
  }
}
```

Listing 3: components

```
import React from ' react ' ;
import { View } from ' react -native' ;
import Counter from ' ./ Counter. js '
```

```
4
5  export default class App extends React.Component {
6
7    constructor(props){
8      super(props);
9      this.milliseconds = 16
10   }
11
12   render() {
13     return (
14       <View style={{flex:1 , alignItems:'center', justifyContent:'center'}}>
15         <Counter milliseconds={this.milliseconds} />
16       </View>
17     );
18   }
19 }
```

Listing 4: Hejsan

```
1  import React from 'react';
2  import { Text } from 'react-native'
3
4  export default class Counter extends React.Component {
5
6      constructor(props) {
7          super(props);
8          this.state = {
9              milliseconds : this.props.milliseconds,
10             counter : 1
11         }
12
13         setInterval(() =>
14             this.setState(prev => (
15                 {counter : prev.counter + 1}
16             )), this.state.milliseconds);
17     }
18
19     render() {
20         return(
21             <Text>{this.state.counter}</Text>
22         );
23     };
24 }
```

Listing 5: Hejsan

C Implementation of application 2

```
1  import 'package:flutter/material.dart';
2  import 'dart:async';
3
4  void main() => runApp(App());
5
6  class App extends StatelessWidget {
```

```
  final int milliseconds = 1;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(
          child: Column(
            children: <Widget>[
              Holder(milliseconds: this.milliseconds),
              Holder(milliseconds: this.milliseconds * 2),
            ],
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          ),
        ),
      )
    );
  }
}

class Holder extends StatelessWidget {
  final int milliseconds;

  const Holder({Key key, this.milliseconds}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
        child: Column(
        children: <Widget>[
          Counter(milliseconds: milliseconds),
          Counter(milliseconds: milliseconds),
          Counter(milliseconds: milliseconds),
        ],
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      )
    );
  }
}

class Counter extends StatefulWidget {
  final int milliseconds;

  Counter({Key key, this.milliseconds}) : super(key: key);

  @override
  CounterState createState() => CounterState();
}

class CounterState extends State<Counter> {
  int counter = 0;

  @override
  Widget build(BuildContext context) {
    return Text("$counter");
  }
```

```
63
64    @override
65    void initState (){
66      super. initState () ;
67      Timer. periodic (Duration( milliseconds : widget. milliseconds ) , (Timer t) {
68        setState (() {
69          counter += 1;
70        }) ;
71      }) ;
72    }
73  }
```

Listing 6: Hejsan

```
1   import React from ' react ' ;
2   import { View } from ' react -native' ;
3   import Holder from ' ./ Holder. js '
4
5   export default class App extends React.Component {
6
7     constructor (props){
8       super(props);
9       this . milliseconds = 1;
10
11    }
12
13    render () {
14      return (
15        <View style={{flex :1 , alignItems : ' center ' , justifyContent : ' center ' }}>
16          <Holder milliseconds ={this . milliseconds }/>
17          <Holder milliseconds ={this . milliseconds *2}/>
18        </View>
19      ) ;
20    }
21  }
```

Listing 7: Hejsan

```
1   import React from ' react ' ;
2   import { View } from ' react -native' ;
3   import Counter from ' ./ Counter. js '
4
5   export default class Holder extends React.Component {
6
7     render () {
8       return (
9         <View style={{flex :1 , alignItems : ' center ' , justifyContent : ' center ' }}>
10          <Counter milliseconds ={this .props. milliseconds } />
11          <Counter milliseconds ={this .props. milliseconds }/>
12          <Counter milliseconds ={this .props. milliseconds }/>
13        </View>
14      ) ;
15    }
16  }
```

```
1   import React from ' react ' ;
2   import { Text } from ' react -native'
```

```
3
4  export default class Counter extends React.Component {
5      constructor(props) {
6          super(props);
7          this.state = {
8              milliseconds : this.props.milliseconds,
9              counter : 1
10         }
11
12         setInterval (() =>
13             this.setState(prev => (
14                 {counter : prev.counter + 1}
15             )), this.state.milliseconds);
16     }
17
18     render() {
19         return(
20             <Text>{this.state.counter}</Text>
21         );
22     };
23 }
```

Listing 8: Counter component

UMEÅ UNIVERSITY