Opinnäytetyö (AMK)

Tietojenkäsittelyn koulutusohjelma

2020

Klaus Jokisuo

FLUTTER-SOVELLUKSEN TILANHALLINTA

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittelyn koulutusohjelma

2020 | 40 sivua, 5 liitesivua

Klaus Jokisuo

FLUTTER-SOVELLUKSEN TILANHALLINTA

Googlen uusi ohjelmistokehys Flutter on saavuttanut suuren suosion nopean kehityksen ja erittäin kovan suorituskykynsä ansiosta. Flutter eroaa muista kilpailevista ohjelmistokehyksistä tuottamalla alustakohtaisia käännöksiä yhdestä koodikannasta.

Flutter on deklaratiivinen ohjelmistokehys. Deklaratiivisten ohjelmistokehyksien suurin ero imperatiivisiin ohjelmistokehyksiin on sovelluksien mahdollinen kehittäminen ilman sovelluksen käyttöliittymän ja sovelluksen logiikan eriyttämistä.

Sovelluksien tilahallinta on yksinkertaisuudessaan logiikan ja käyttöliittymän yhteen saattamista ja niiden välisen datan viemistä eteenpäin. Deklaratiiviset ohjelmistokehykset voivat aiheuttaa loppukehittäjille vaikeuksia luoda hyvää ja hallittavaa koodia. Työssä käytettiin Flutter, Provider, Flutter BloC virallisia dokumentaatioita.

Tilanhallintatekniikoita vertailtiin teoriaosuudessa tarkastelemalla niiden toimintaa ja ominaisuuksia. Tarkastelun perusteella luotiin kolme käyttöliittymältään identtistä esimerkkisovellusta ja luotiin testi, jolla pystytään vertaamaan esimerkkisovelluksien suorituskykyeroja.

Opinnäytetyö saavutti tarkoituksensa tarjota yksityiskohtaisen vertailun kolmen suositellumman tilanhallinta menetelmän välillä. Tulokset osoittavat, että Flutter BloC tilanhallintatekniikkaa kannattaa käyttää suuremmissa sovelluksissa. Provider on selkeä voittaja helppokäyttöisyyden suhteen. SetStatea tulisi käyttää täydentämään käytettyä tilanhallintatekniikkaa, ylimääräisen koodin vähentämiseksi.

ASIASANAT:

Flutter, Dart, State management, setState, Provider, BloC, Business logic component

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology

2020 | 40 pages, 5 in appendices

Klaus Jokisuo

FLUTTER APPLICATION STATE MANAGEMENT

Google’s new cross-platform framework Flutter has gained great popularity due to its rapid development speed and high performance. Flutter differs from other competing software frameworks because it produces platform-specific binaries from a single codebase.

Flutter is a declarative software framework. The greatest difference between declarative software frameworks and imperative software frameworks is the possible development of applications without differentiating the application interface and application logic.

At its core, application state management is bringing application logic and application user interface together and creating communication between them. Declarative software frameworks can make it difficult for the developer to create good and maintainable code. The official documentation of Flutter, Provider, Flutter BloC was used in the study.

State management techniques were compared in the theoretical part of this thesis by examining their properties and features. Based on the review, three sample applications with identical user interfaces were created and a test was created to compare the performance differences of the sample applications.

The thesis achieved its purpose to provide a detailed comparison of the current top three recommended state management approaches. The results indicate that Flutter BloC is the leader when it comes to handling larger applications. When it comes to ease of use the Provider is the clear winner. SetState should be used to complement the used state management technology to reduce boilerplate code.

KEYWORDS:

Flutter, Dart, State management, setState, Provider, BloC, Business logic component

SISÄLTÖ

SANASTO **7**

1 JOHDANTO **8**

2 TEKNOLOGIAKATSAUS **9**
2.1 Flutter 9
2.2 Dart 10

3 FLUTTER-SOVELLUKSEN TILANHALLINTA **12**
3.1 Tilanhallinta 12
3.1.1 Ohjelman tila 12
3.1.2 Hetkellinen tila 12
3.1.3 Sovelluksen tila 13
3.2 Deklaratiivinen vs. Imperatiivinen 13
3.3 Widgetit 14
3.4 Build Context 16

4 SETSTATE **17**
4.1 Esittely 17
4.2 Rajoitukset 18
4.3 Uudelleenrakennuksen rajaus 19

5 PROVIDER **20**
5.1 Esittely 20
5.2 Widgetit 20
5.3 Uudelleenrakennuksen rajaus 21

6 FLUTTER BLOC **23**
6.1 BloC 23
6.2 Esittely 23
6.3 Widgetit 24
6.4 Uudelleenrakennuksen rajaus 25

7 TOTEUTUS **26**
7.1 Käyttöliittymä 26

7.2 setState 27
7.2.1 setState käyttö 27
7.3 Provider 28
7.3.1 Provider käyttö 28
7.4 Flutter BloC 31
7.4.1 Flutter BloC käyttö 31
8 TESTAUS 34
8.1 Käynnistysaika 34
8.2 Ajonaikainen suorituskyky 35
9 YHTEENVETO 37
LÄHTEET 39

LIITTEET

Liite 1. Esimerkkisovelluksen käyttöliittymän lähdekoodi.
Liite 2. Esimerkkisovellus - setState.
Liite 3. Esimerkkisovellus - Provider.
Liite 4. Esimerkkisovellus – Flutter Bloc.
Liite 5. Esimerkkisovellus – Testi.

KOODI

Koodi 1. Esimerkki navigointipalkki. 13
Koodi 2. Mounted-funktion käyttö 17
Koodi 3. setState-funktion käyttö. 17
Koodi 4. Yksinkertainen Provider-widget. 20
Koodi 5. Yksinkertainen ChangeNotifierProvider-widget. 21
Koodi 6. Yksinkertainen FutureProvider-widget. 21
Koodi 7. Yksinkertainen StreamProvider-widget. 21
Koodi 8. Yksinkertainen BlocProvider-widget. 24
Koodi 9. Yksinkertainen BlocBuilder-widget. 24
Koodi 10. Yksinkertainen BlocListener-widget. 25
Koodi 11. Tilallisen widgetin luokan implementointi. 27
Koodi 12. pubspec.yaml tiedosto. 28
Koodi 13. Riippuvaisuuksien asennuskomento. 28
Koodi 14. ChangeNotifierProvider-widget. 28
Koodi 15. ChangeNotifier-luokka. 29
Koodi 16. pubspec.yaml tiedosto. 31
Koodi 17. BlocProvider-widget. 31

Koodi 18. mapEventToState-metodi. 32
Koodi 19. Logiikkataso-luokan rakentaja. 32
Koodi 20. Tapahtuma tyyppi. 32
Koodi 21. Sovelluksen käynnistysajan mittaus. 34
Koodi 22. Trace-startup komennon esimerkkituloste. 34
Koodi 23. Suorituskykytestin esimerkkituloste. 35

KUVAT

Kuva 1. Flutter-järjestelmän yleiskatsaus (GitHub 2020). 10
Kuva 2. Dart-ohjelmointikielen tukemat alustat (Dart 2020). 11
Kuva 3. Tilanhallintakaava (Flutter 2020). 12
Kuva 4. Flutter-sovelluksen esimerkki widget rakenne (Flutter 2020). 15
Kuva 5. Periytyvän widgetin toiminta (Google Developers 2020a). 16
Kuva 6. Monimutkainen widget-puu (Flutter 2020). 18
Kuva 7. setState-funktio kutsun jälkeinen widgettien uudelleenrakennus määrä. 19
Kuva 8. notifyListeners-metodi kutsun jälkeinen widgettien uudelleenrakennus määrä. 22
Kuva 9. BloC-arkkitehtuuri (Bloc 2020). 24
Kuva 10. BloC-tilapäivityksen jälkeinen widgettien uudelleenrakennusmäärä. 25
Kuva 11. Esimerkkisovelluksen käyttöliittymä. 26
Kuva 12. setState-funktion implementointi onPressed callback -funktioon 27
Kuva 13. Decrement- ja Increment-metodien kutsu. 29
Kuva 14. Consumer-widgetin käyttö. 30
Kuva 15. CounterEvent.decrement ja CounterEvent.increment tapahtumien lähetys. 33
Kuva 16. BlocBuilder-widgetin käyttö. 33
Kuva 17. Esimerkkisovelluksen käynnistysajan testin tulokset. 35
Kuva 18. Esimerkkisovelluksen suorituskykytestauksen tulokset. 36

SANASTO

Widget	Widgetit ovat Flutter-sovelluksen käyttöliittymän perusrakenteita, jolla luodaan Flutter-sovelluksen käyttöliittymä (Flutter 2020e)
Tilaton widget	Tilaton widget on staattinen widgetti, jossa itsessään ei laisinkaan tilaa (Flutter 2020e)
Tilallinen widget	Tilallinen widget on dynaaminen widgetti, joka pystyy hallinnoimaan oma tilaansa (Flutter 2020e)
Periytyvä widget	periytyvä widget on widgetti, joka pystyy jakamaan tietoa alla oleville widgeteillä (Flutter 2020f)
Build Context	Build context hallitsee widgetin sijaintia widget-puussa (Flutter 2020i)

1 JOHDANTO

Flutter on Googlen luoma ohjelmistokehityspaketti, jolla voi luoda järjestelmäriippumattomia sovelluksia yhdestä koodikannasta (Flutter 2020a). Flutter on deklaratiivinen ohjelmistokehys, joka mahdollistaa sovelluksien kehittämisen ilman sovelluksen käyttöliittymän ja sovelluksen logiikan eriyttämistä. On tärkeää valita sellainen tilanhallintatekniikka, joka tukee ja pakottaa kehittäjää kirjoittamaan parempaa ja hallittavampaa koodia. (Flutter 2020b.)

Opinnäytetyössä tehdään teknologiakatsaus Flutter-ohjelmistokehityspakettiin ja Dart-ohjelmointikieleen. Opinnäytetyössä myös vertaillaan ja tutkitaan kokeellisesti kolmea erilaista Flutter-kehitysryhmän ja Flutter yhteisön suosittelemia Flutter tilanhallintatekniikoita. Tutkittavina tekniikoina ovat setState, Provider ja BloC. Kyseisten tilanhallintatekniikoiden eroavaisuuksia analysoidaan koodin, suorituskyvyn ja käytettävyyden näkökulmasta ja tarkastellaan niiden tarpeellisuutta Flutter-sovelluksien kehittämisessä.

Tilanhallintatekniikoiden vertailu aloitetaan luomalla yleiskatsaus Flutter-sovelluksen tilanhallintaan ja analysoidaan sen vaikutuksia koodin ylläpitoon, kehitykseen ja suorituskykyyn. Tilanhallintatekniikoita tutkitaan käytännössä luomalla kolme käyttöliittymään nähden identtistä esimerkkisovellusta, joissa jokaisessa käytetään eri tilanhallintatekniikkaa. Vertailun jälkeen tehdään suositus, mitä tilanhallintatekniikkaa kannattaa käyttää riippuen Flutter-sovelluksen ominaisuuksien määrästä ja koodikannan koosta.

2 TEKNOLOGIAKATSAUS

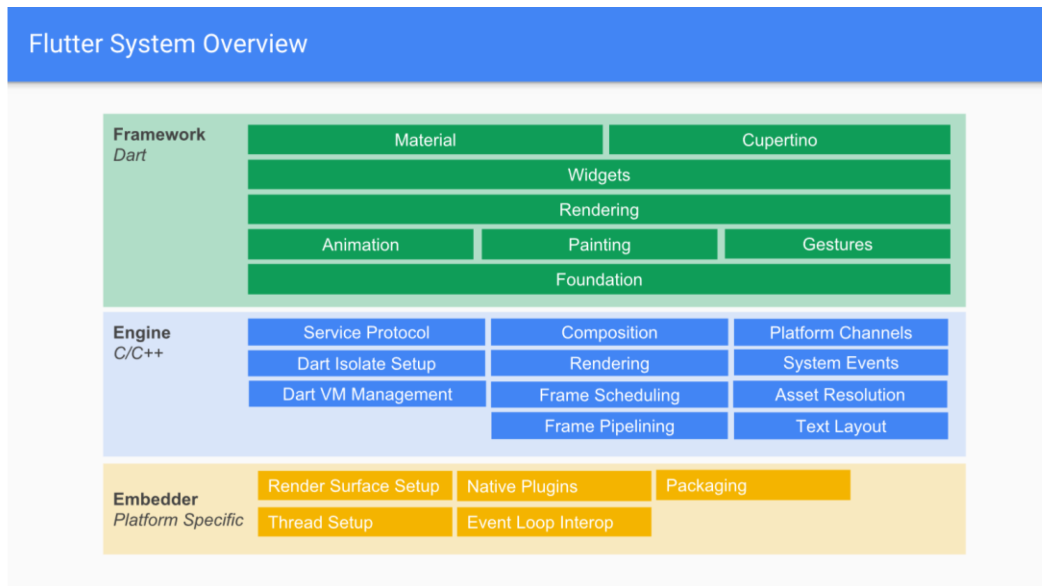
2.1 Flutter

Flutter on Googlen luoma ohjelmistokehityspaketti, jonka ensimmäinen vakaa versio julkaistiin vuonna 2018 (Flutter 2020c). Flutterin tarkoituksena on mahdollistaa kehittäjiä luomaan kauniita ja natiiveja sovelluksia erilaisille alustoille, kuten mobiili, web ja työpöytä (Flutter 2020a). Tällä hetkellä tuotanto valmiit alustat ovat Android ja iOS (Flutter 2020c).

Flutter lähestyy alustojen välistä sovelluskehitystä radikaalimmalla tavalla kuin kanssakilpailijansa. Muihin järjestelmäriippumattomiin viitekehyksiin verrattuna Flutter ei käytä laitevalmistajien tarjoamia widgettejä. Sen sijaan Flutter käyttää Skia-renderöintimoottoria luodakseen widgetit, joka mahdollistaa Flutter-sovelluksien korkean ja tasaisen laadun. (Flutter 2020c.)

Flutter tarjoaa valmiiksi luotuja korkean tason widgettejä kehittäjille, jotka on luotu yhdistämällä alemman tason widgettejä yhteen, joita kehittäjät ovat tottuneet käyttämään Android ja iOS alustoilla. Flutter jakaa Android- ja iOS alustojen widgetit kahteen ryhmään: Material Design (Android) ja Cupertino (iOS). Flutter on suunniteltu siten, että loppukehittäjät voivat widgettejä yhdistämällä luoda uusia korkean tason widgettejä. (Flutter 2020c.)

Flutter on avoimen lähdekoodin ohjelmistokehityspaketti, joka mahdollistaa sovelluksen täydellisen kustomoinnin, verrattuna natiiviin Android- ja iOS kehitykseen, jossa kehittäjä työskentelee suljetun lähdekoodin parissa. Flutter on luotu käyttämällä C-, C++- ja Dart-ohjelmointikieliä (Kuva 1). Suurin osa lähdekoodista on kirjoitettu Dart-ohjelmointikielellä, joka tekee Flutterista helposti lähestyttävän ohjelmistoviitekehyksen kehittäjille. (Flutter 2020c.)



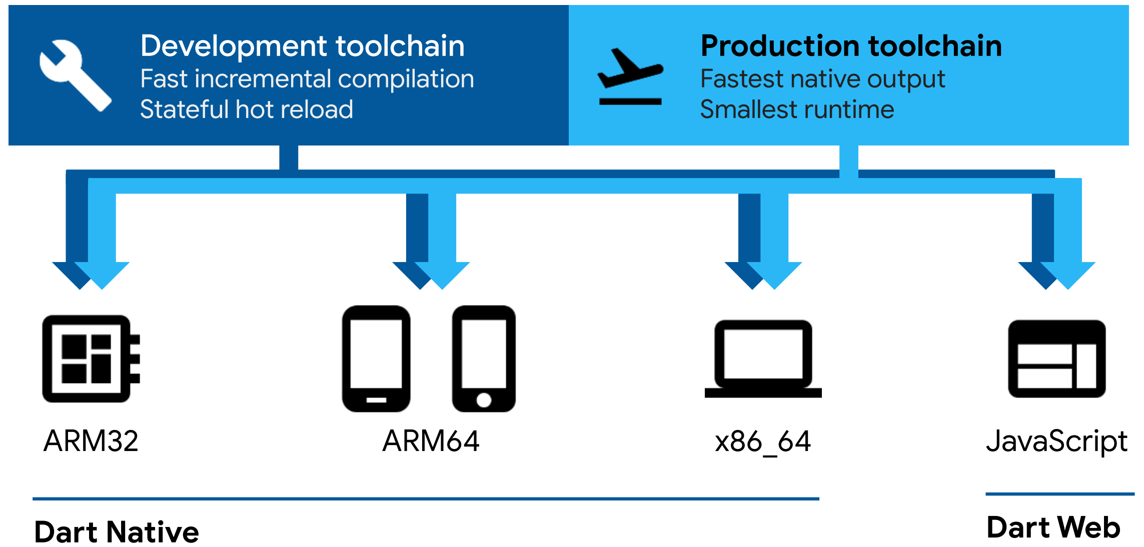
Kuva 1. Flutter-järjestelmän yleiskatsaus (GitHub 2020).

2.2 Dart

Dart on Googlen kehittämä ohjelmointikieli, jota käytetään Flutter-sovelluksen kehityksessä kuin myös itse Flutter-sovelluskehyksessä. Dart-ohjelmointikieli käyttää samankaltaista syntaksia kuin Java, jota on yksinkertaistettu kehittäjän näkökulmasta. Dart-ohjelmointikielen tarkoitus ei koskaan ollut olla mullistava, vaan helposti omaksuttava ja lähestyttävä ohjelmointikieli alkaville kuin myös kehittyneemmille ohjelmoijille. (Dart 2020a.)

Dart-ohjelmointikieli on optimoitu käyttöliittymien luomiseen, antamalla kehittäjille tarpeelliset työkalut nopeaan sovelluskehitykseen. Flutter sisältää Hot-reload-toiminnon, joka kääntää käynnissä olevan sovelluksen koodin sen tilaa muuttamatta. Hot-reload toiminnon mahdollistaa Dart-ohjelmointikielen JIT (Just in Time) -kääntäjä (Kuva 2). (Dart 2020b.)

Flutter-sovelluksien yksi suurimpana nopeuteen liittyvistä tekijöistä ovat alustakohtaiset käännökset, jonka vuoksi Flutter-sovellukset ovat suorituskykyisiä. Alustakohtaiset käännökset ovat mahdollisia Dart-ohjelmointikielen AOT (Ahead of Time) -kääntäjän avulla (Kuva 2). (Dart 2020b.)



Kuva 2. Dart-ohjelmointikielen tukemat alustat (Dart 2020).

3 FLUTTER-SOVELLUKSEN TILANHALLINTA

3.1 Tilanhallinta

Tilanhallinta yksinkertaisuudessaan tarkoittaa tapaa, miten sovelluksen tilaa voi muuttaa ja lukea. Deklaratiivista viitekehystä käyttäessä sovelluksen tilanhallinta on kehittäjän vastuulla, kehittäjän pitää itse laukaista käyttöliittymän uudelleenrakentaminen, jotta käyttöliittymä pysyisi ajan tasalla sovelluksen tilan kanssa (Kuva 3).

Kuva 3. Tilanhallintakaava (Flutter 2020).

3.1.1 Ohjelman tila

Laajassa mielessä ohjelman tila tarkoittaa kaikkea ohjelman muistissa olevaa tietoa, kun ohjelma on käynnissä. Ohjelman tiedoilla tarkoitetaan ohjelman muistiin ladattuja tiedostoja, muuttujia, tekstuureja ja kaikkea sellaista, jota ohjelma pystyy hyödyntämään ohjelman käynnissä olon aikana. (Flutter 2020l.)

3.1.2 Hetkellinen tila

Hetkellinen tila, jota myöskin kutsutaan nimellä käyttöliittymän, tila on tila, jota yksi widgetti pystyy hallitsemaan. Esimerkiksi navigointipalkin nykyistä valintaa voidaan kutsua hetkelliseksi tilaksi. Navigointipalkin tila on yksinkertainen ja muut widgetit eivät ole riippuvaisia kyseisestä tilasta, jolloin widgetin ei tarvitse ottaa huomioon mitään muita muuttujia kuin mitä itse widgetin tilaan on asetettu. Esimerkki navigointipalkin tilaa muutetaan

käyttämällä setState tilanhallinta menetelmää, jolloin tilan päivitys tapahtuu vain widgetin sisällä (Koodi 1). (Flutter 2020l.)

```
class MyHomepage extends StatefulWidget {
  @override
  _MyHomepageState createState() => _MyHomepageState();
}

class _MyHomepageState extends State<MyHomepage> {
  int _index = 0;

  @override
  Widget build(BuildContext context) {
    return BottomNavigationBar(
      currentIndex: _index,
      onTap: (newIndex) {
        setState(() {
          _index = newIndex;
        });
      }, items: [],
    );
  }
}
```

Koodi 1. Esimerkki navigointipalkki.

3.1.3 Sovelluksen tila

Sovelluksen tila, jota myöskin kutsutaan jaetuksi tilaksi, on tila, jota jaetaan sovelluksen widgeteille, jolloin monet widgetit pystyvät muuttamaan sitä. Sovelluksen tila säilyy myöskin istuntojen vaihtuessa. Esimerkkejä sovelluksen tilasta on käyttäjäntiedot ja kirjautumistiedot. (Flutter 2020e.) Sovelluksen tila määritellään siinä kohtaa widget-puuta, josta alapuolella olevien widgettien halutaan pystyä lukemaan ja hallitsemaan tilaa.

3.2 Deklaratiivinen vs. Imperatiivinen

Ohjelmistokehykset, joilla yleensä luodaan Win32-, web-, Android- ja iOS-sovelluksia, käyttävät yleisesti imperatiivista tapaa käyttöliittymän ohjelmointiin. Imperatiivisessa ohjelmistokehyksessä kehittäjä luo käyttöliittymän kokonaisuudessaan, jonka jälkeen luodaan erilaisia kontrollereita, jotka liitetään käyttöliittymään. Kontrollereiden avulla käyttöliittymää asetetaan uusia arvoja tarpeen mukaan. (Flutter 2020m.)

Flutter on deklaratiivinen ohjelmistoviitekehys, joka eroaa imperatiivisista ohjelmistoviitekehyksistä siten, että Flutter ei hajauta käyttöliittymää, kontrollereita ja asetteluita erikseen vaan käyttää widgettejä yhtenäisenä ja johdonmukaisena mallina hallita Flutter-sovellusta. (Flutter 2020e, Flutter 2020b)

3.3 Widgetit

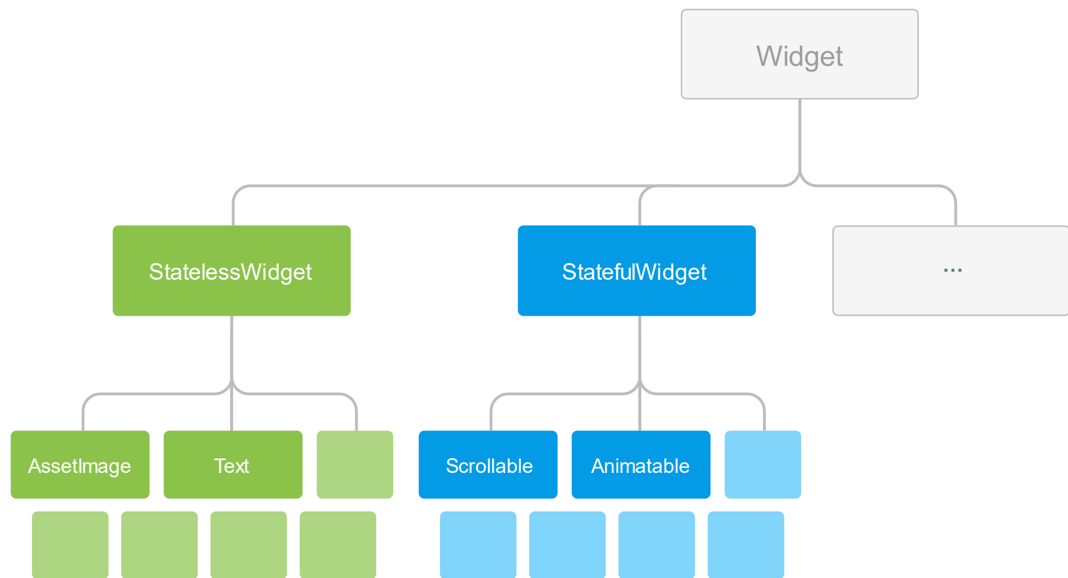
Pystyäkseen ymmärtämään Flutter-sovelluksen tilanhallintaa ja erilaisia tilanhallinta menetelmiä on tärkeää ymmärtää, mitä widgetit ovat, minkä tyyppisiä widgettejä Flutter tarjoaa ja miten ne eroavat toisistaan.

Widgetit ovat Flutter-sovelluksen käyttöliittymän perusrakenteita. Widgetit koostuvat usein yhdistämällä alemman tason widgettejä yhteen, jolloin pystytään luomaan erilaisia kokonaisuuksia. Flutter tarjoaa kehittäjälle käytettäväksi kolmea erilaista widgettiä. Käyttöliittymää luodessa käytetään tilattomia widgettejä ja tilallisia widgettejä (Kuva 4). (Flutter 2020e.) Widgettien väliseen informaation jakamiseen voidaan käyttää periytyviä widgettejä.

Tilaton widget on staattinen widgetti, jossa itsessään ei laisinkaan tilaa, vaan widgetti saa tarvittavat argumentit suoraan parent widgetistä ja argumentteja käytetään luomaan uusi tilaton widgetti. Tilattomia widgettejä käytetään silloin kun widgetin tilaa ei tarvitse hallita widgetin sisältä käsin, vaan se hoidetaan luomalla widgetti kokonaan uusiksi tai käyttämällä erilaisia tilanhallinta menetelmiä. (Flutter 2020e.)

Tilallinen widget on dynaaminen widgetti, joka pystyy hallinnoimaan omaa sisältöään muuttamalla omaa tilaansa. Tilallisia widgettejä käytetään silloin kun on tarvetta pystyä muuttamaan widgetin tilaa muualtakin kuin ulkoisesti. (Flutter 2020e.)

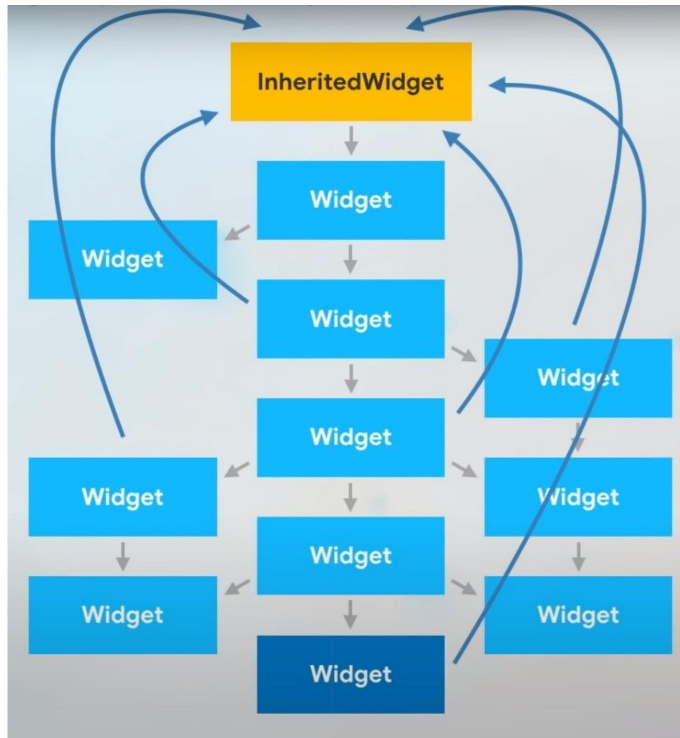
Tilallisia widgettiä käytetään myös, kun halutaan ylläpitää erilaisten kontrollereiden tilaa kuten vierittämisen määrää nykyisessä sivussa ja animaation tilaa erilaisissa animaatio widgeteissä (Kuva 4).



Kuva 4. Flutter-sovelluksen esimerkki widget rakenne (Flutter 2020).

Periytyvä widget eroaa muista widgeteistä siten, että sitä ei pysty käyttämään käyttöliittymän luomiseen. Periytyvän widgetin tarkoitus on helpottaa informaation jakamista eri widgettien kesken ja sitä voi myöskin käyttää tilanhallinta menetelmänä. (Flutter 2020f.)

Periytyvää widgettiä käytettäessä käytetään myöskin build context luokan dependOnInheritedWidgetOfExactType-metodia. dependOnInheritedWidgetOfExactType-metodi mahdollistaa InheritedWidgetin muutoksien kuuntelun alempana widget-puussa. Widgetit, jotka kuuntelevat periytyvän widgetin muutoksia, tullaan rakentamaan uudelleen muutoksien tapahtuessa (Kuva 5). (Flutter 2020f.)



Kuva 5. Periytyvän widgetin toiminta (Google Developers 2020a).

3.4 Build Context

Build contextin tarkoitus on hallita widgetin sijaintia widget-puussa. Build context välitetään WidgetBuilder-funktioissa, kuten tilattoman widgetin build-funktiossa. Koska build context pitää sisällään widgetin informaatiota, sitä alemmat widgetit voivat käyttää erilaisia funktiota päästäkseen käsiksi widgetin informaatioon. (Flutter 2020k.)

Monet tilanhallinta kirjastot rakentavat koodin periytyvän widgetin ympärille koska se sallii getElementForInheritedWidgetOfExactType-metodin käytön. getElementForInheritedWidgetOfExactType on build context luokan alainen metodi ja build context välitetään aina uuden widgetin luomistilanteessa, jolloin sitä helppo käyttää informaation välittämiseen ylemmistä widgeteistä koko widget-puuhun (Flutter 2020i).

getElementForInheritedWidgetOfExactType ja dependOnInheritedWidgetOfExactType eroavat toisistaan siten, että getElementForInheritedWidgetOfExactType ei luo suhdetta haettuun periytyvään widgettiin, jolloin metodin kutsuja ei myöskään rakennu uudelleen periytyvän widgetin tilan muuttuessa. (Flutter 2020k.)

4 SETSTATE

4.1 Esittely

SetState-funktio on tilanhallintatekniikoista yksinkertaisin.

Kutsumalla setState-funktiota, funktio ilmoittaa Flutter-ohjelmistokehykselle, että objektin sisäinen tila on muuttunut sillä tavalla, että se voi vaikuttaa sovelluksen käyttöliittymään kyseisessä widget-puussa. Funktiokutsun jälkeen ohjelmistokehys aikatauluttaa kyseisen objektin tilan uudelleen rakentamisen. (Flutter 2020g.)

setState-funktiota voidaan käyttää ainoastaan tilallisissa widgetissä eikä sitä saa kutsua dispose-metodin jälkeen. Dispose-metodi poistaa widgetin widget puusta, jolloin setState funktio yrittäisi uudelleen rakentaa sellaista widgettiä, jota ei ole enää widget-puussa. Yksinkertainen tapa tarkastaa onko widget vielä widget-puussa on käyttää mounted-funktiota (Koodi 2). (Flutter 2020g.)

```
onPressed: () {
  if (mounted) {
    setState(
      () => _counter++,
    );
  }
},
```

Koodi 2. Mounted-funktion käyttö

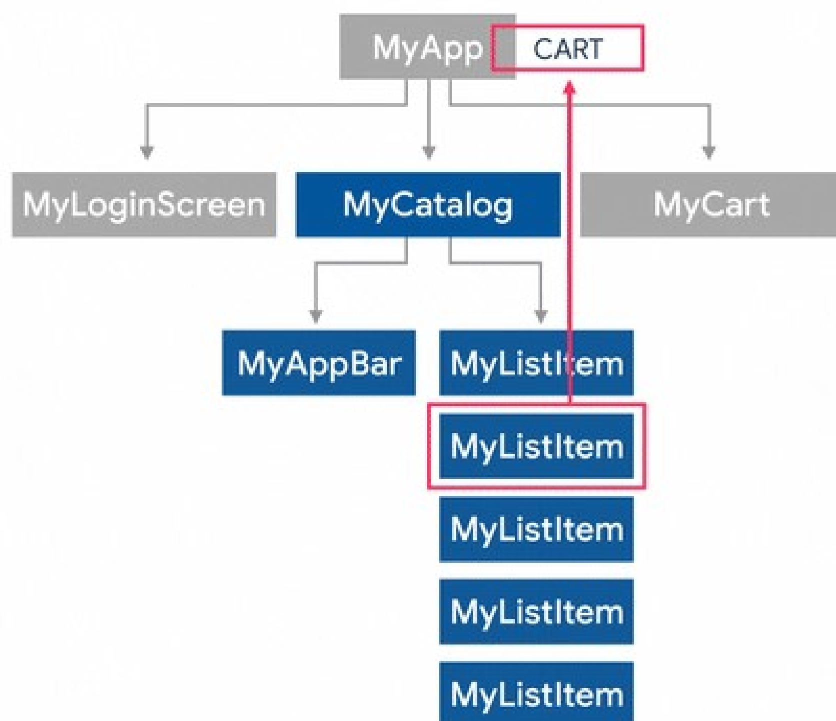
setState-funktio kutsun rakenne on erittäin yksinkertainen. Funktion runkoon ei ole pakko laittaa muuttuvan arvon muuttujaa ja uutta arvoa mutta sitä pidetään koodin luettavuuden kannalta suositeltavana (Koodi 3). (Flutter 2020g.)

```
setState(() {oldValue = newValue;})
```

Koodi 3. setState-funktion käyttö.

4.2 Rajoitukset

Useimmiten sovellus koostuu useammasta kuin yhdestä sivusta tai korkeamman tason widgetistä, jolloin widgettien välinen informaation vaihtaminen tulee tarpeen. Esimerkiksi verkkokauppa sovelluksessa käyttäjä avaa tuotteen lisätiedot ja sitä kautta lisää sen ostoskoriin, pelkän setState-funktion käyttäminen tässä tilanteessa luo enemmän ongelmia kuin ratkaisuja. Suurimmat ongelmat tässä tapauksessa ovat tilanhallinta ja koodin ylläpito. Kun käytetään pelkästään setState-funktiota, emme pysty helposti hallinnoimaan informaation välitystä muille widgeteille, jolloin lopputulos on usein se, että eri widgeteillä on väärä tila, koska muilla widgeteillä ei ole tietoa siitä onko tila muuttunut vai ei (Kuva 6).



Kuva 6. Monimutkainen widget-puu (Flutter 2020).

4.3 Uudelleenrakennuksen rajaus

Käyttämällä setState-funktiota ohjelmistokehys joutuu uudelleenrakentamaan kaikki widgetit, jotka ovat kyseisen widgetin alapuolella. Uudelleenrakentaminen loppuu, kun ohjelmistokehys havaitsee samantyyppisen widgetin arvon samanlaiseksi kuin ennen uudelleenrakentamisen aloittamista. Widgettien, joiden arvo ei muutu uudelleenrakentamisen jälkeen on turhaa ja kallista suorituskyvyn kannalta, koska ei tuota mitään uutta käyttöliittymään (Kuva 7). (Flutter 2020h.)

Widget	Location	Last Frame	Current Screen
	main.dart:6	1	2
Text	setState_example.dart:21	1	2
AppBar	setState_example.dart:20	1	2
Icon	setState_example.dart:56	1	2
FloatingActionButton	setState_example.dart:49	1	2
Text	setState_example.dart:44	1	2
Icon	setState_example.dart:40	1	2
FloatingActionButton	setState_example.dart:33	1	2
Text	setState_example.dart:27	1	2
Scaffold	setState_example.dart:19	1	2
MaterialApp	setState_example.dart:13	1	2

Kuva 7. setState-funktio kutsun jälkeinen widgettien uudelleenrakennus määrä.

5 PROVIDER

5.1 Esittely

Provider on Flutter kehitystiimin suosittelema tilanhallintakirjasto. Providerin luonut Rémi Rousselet kuvaa Provideria ei niinkään tilanhallinta menetelmäksi vaan osaksi sitä, joka auttaa kehittäjää luomaan parempaa ja hallittavampaa koodia. (Skills matter 2020.)

Provider on luotu periytyvän widgetin ympärille, luodakseen helpomman tavan käyttää periytyvän widgetin ominaisuuksia. Provider tarjoaa lukuisia widgettejä, joiden tarkoitus on altistaa ja kuunnella informaatio muutoksia erilaisista data-luokista. (GitHub 2020.)

5.2 Widgetit

Kaikkia Provider-widgettejä yhdistää yksi tekijä, altistaa informaatio sen alapuolella oleviin widgetteihin. Yleisimmät Provider-widgetit ovat Provider, ChangeNotifierProvider, FutureProvider ja StreamProvider. (GitHub 2020.)

Provider-widget altistaa kokonaisen luokan informaation, mukaan lukien muuttujat ja metodit, alapuolella oleville widgeteille. Create callback -funktioon annetaan luokka, jonka informaatio halutaan altistaa alemmille widgeteille (Koodi 4). (GitHub 2020.)

```
Provider<Counter>(
create: (context) => Counter(),
child: MaterialApp()
```

Koodi 4. Yksinkertainen Provider-widget.

ChangeNotifierProvider-widget tarjoaa samat ominaisuudet kuin Provider, mutta pystyy myöskin ilmoittamaan alapuolella oleville widgeteille koska ChangeNotifierProvider-widgetin tarjoaman informaatio muuttuu. Informaation muutos ilmoitetaan käyttämällä notifyListeners-funktiota, jonka ilmoituksen vastaanottaa Consumer-widget (Koodi 5). (GitHub 2020.)

```
ChangeNotifierProvider<Counter>(
 create: (context) => Counter(),
 child: Consumer<Counter>(
   builder: (context, counter, child) => Text('${counter.value}')
 ));
```

Koodi 5. Yksinkertainen ChangeNotifierProvider-widget.

FutureProvider eroaa muista Provider-widgeteistä kuuntelemalla asynkronisia muutoksia. FutureProvider odottaa widgetin rakentamista niin kauan kunnes saa jonkun arvon create callback-funktiosta. Create callback-funktioon annetaan asynkroninen-metodi tai funktio, joka palauttaa määritellyn data-luokan (Koodi 6). (GitHub 2020.)

```
FutureProvider<Counter>(
 create: (context) => asyncFunctionToGetCounterModel(),
 child: Consumer<Counter>(
   builder: (context, counter, child) => Text('${counter.value}')
 ));
```

Koodi 6. Yksinkertainen FutureProvider-widget.

StreamProvider eroaa FutureProvider-widgetistä kuuntelemalla viimeisintä arvoa, jonka Dart-ohjelmointikielen tukema Stream-luokka antaa. Saadessaan uuden arvon StreamProvider rakentaa widgetin uudelleen ja palauttaa viimeisimmän arvon kutsuja-metodille (Koodi 7). (GitHub 2020.)

```
StreamProvider<int>.value(
  value: Stream<int>.periodic(Duration(milliseconds: 100),
          (count) => count + 1).take(100),
 child: Consumer<int>(
     builder: (context, value, child) => Text('$value')
 ));
```

Koodi 7. Yksinkertainen StreamProvider-widget.

5.3 Uudelleenrakennuksen rajaus

Käyttämällä Provider-widgettejä kehittäjä pystyy rajaamaan mitä widgettejä tullaan rakentamaan uudelleen (Kuva 8). Uudelleen rakennettavat widgetit rajataan joko

Consumer-widgetillä tai erilaisilla Provider-widgeteillä. Provider- ja Consumer-widgetit tarjoavat kehittäjälle builder callback -funktion, jotka mahdollistavat uudelleenrakennettavien widgettien rajaamisen. (Flutter 2020j.)

Widget	Location	Last Frame	Current Screen
Text	provider_example.dart:56	1	2
Consumer	provider_example.dart:55	1	2

Kuva 8. notifyListeners-metodi kutsun jälkeinen widgettien uudelleenrakennus määrä.

6 FLUTTER BLOC

6.1 BloC

BloC eli Business Logic Controller. BloC-kaavan on kehittänyt Google ja se esiteltiin vuonna 2018 Google I/O tapahtumassa. Google kehitti BloC-kaavan helpottaakseen Flutter-sovelluksen tilanhallintaa. (Google Developers 2020b.)

Kun puhutaan tilanhallinta menetelmästä, joka perustuu BloC-kaavaan, tulee kehittäjän eriyttää kaikki logiikkaan liittyvä koodi omiin luokkiinsa. Logiikkaa sisältävien luokkien perusperiaate on ainoastaan kuunnella ja lähettää data asynkronisesti käyttämällä streamejä. (Google Developers 2020b.)

Dart Stream-luokka tarjoaa tavan vastaanottaa tapahtumasarjoja. Jokainen tapahtuma jaetaan kahteen eri tyyppiin: data tapahtumaan tai virhetapahtumaan. Data tapahtuma, joita myöskin kutsutaan elementeiksi voi sisältää minkä tyyppistä dataa tahansa. Virhe tapahtuma tulee silloin kun tapahtuu jokin virhe. (Dart 2020c.)

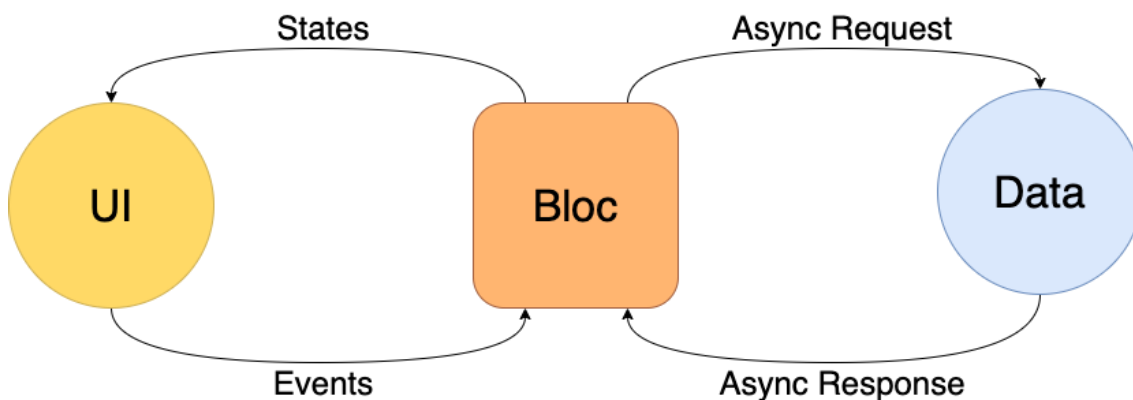
6.2 Esittely

Flutter BloC tilanhallintakirjaston on luonut Felix Angelov. Flutter BloC hyödyntää Googlen luomaa BloC-kaavaa. Flutter BloC on kehitetty helpottamaan ja pakottamaan kehittäjiä luomaan selkeämpää ja hallittavampaa koodia. Flutter BloC koostuu kolmesta tasosta: Informaatio, logiikka ja käyttöliittymä (Kuva 9). (Bloc 2020.)

Informaatiotaso on alin kaikista tasoista. Informaatiotason tehtävänä on keskustella tietokantojen kanssa tai muiden asynkronisten informaatiolähteiden kanssa ja luoda verkkopyyntöjä. (Bloc 2020.)

Logiikkatason tehtävänä on hallinnoida ja vastaanottaa uusia tapahtumia käyttöliittymätasosta ja informaatiotasosta. Logiikkataso vastaanottaa annetut tapahtumat ja keskustelee informaatiotason kanssa rakentaakseen uuden tilan käyttöliittymälle. (Bloc 2020.)

Käyttöliittymätason tehtävänä on vastaanottaa uusia tapahtumia logiikkatasolta ja renderöidä itsensä uudelleen tapahtumien perusteella. Käyttöliittymätason muihin tehtäviin kuuluu myöskin käsitellä käyttäjän ja sovelluksen elinkaari tapahtumat. (Bloc 2020.)



Kuva 9. BloC-arkkitehtuuri (Bloc 2020).

6.3 Widgetit

Flutter BloC tarjoaa widgettejä, joiden tarkoitus on tarjota informaatiota tai vastaanottaa saapuvia tapahtumia logiikkatasolta (Pub.dev 2020).

BlocProvider-widget altistaa määritellyn logiikkatason alapuolella oleville widgeteille. BlocProvideria käytetään riippuvuusinjektiona, jotta vain yksittäinen instanssi logiikkatasosta voidaan altistaa alapuolella oleville widget-puille. BlocProvider-widgetissä on create callback -funktio, jota suositellaan logiikkatason luomiseen koska silloin BlocProvider voi myös automaattisesti sulkea sen (Koodi 8). (Pub.dev 2020)

```
BlocProvider(
  create: (context) => CounterBloc(), child: Container());
```

Koodi 8. Yksinkertainen BlocProvider-widget.

BlocBuilder-widget vaatii olemassa olevan logiikkatason pystyäkseen vastaanottamaan tapahtumia. Logiikkatasosta vastaanotetut tapahtumat hallinnoivat koska widget pitää rakentaa uudelleen (Koodi 9). (Pub.dev 2020)

```
BlocBuilder<CounterBloc, int>(
  builder: (context, state) => Text('$state')),
```

Koodi 9. Yksinkertainen BlocBuilder-widget.

BlocListener-widget tarjoaa samat ominaisuudet kuin BlocBuilder paitsi ei koskaan rakenna widgettiä uudelleen. BlocListenerin tehtävä on kuunnella ja reagoida vastaanotettuihin tapahtumiin sellaisella tavalla, joka ei vaadi widgetin uudelleenrakentamista (Koodi 10). (Pub.dev 2020)

```
BlocListener(
  listener: (context, state) {
    if(state != null){
      print('State is not null');
    }
  },
));
```

Koodi 10. Yksinkertainen BlocListener-widget.

6.4 Uudelleenrakennuksen rajaus

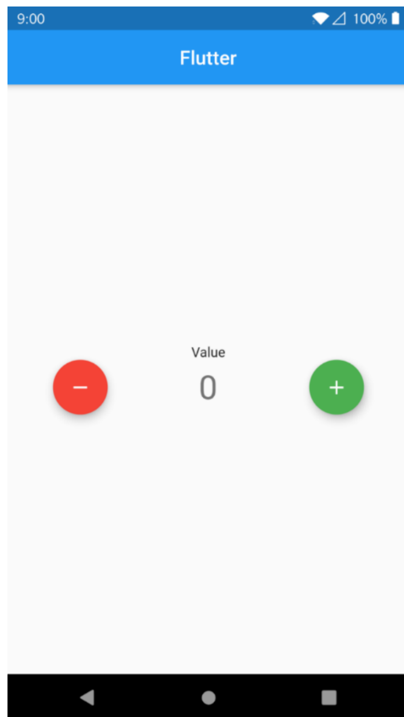
Käyttämällä BlocBuilder-widgettiä, kehittäjä pystyy rajaamaan mitä widgettejä tullaan rakentamaan uudelleen. Uudelleen rakennettavat widgetit rajataan BlocBuilder-widgetillä. BlockProvider-widgetti tarjoaa kehittäjälle builder callback -funktion, joka mahdollistaa uudelleenrakennettavien widgettien rajaamisen (Kuva 10). (Pub.dev 2020.)

Widget	Location	Last Frame	Current Screen
Text	bloc_example.dart:62	1	2
BlocBuilder	bloc_example.dart:61	1	2

Kuva 10. BloC-tilapäivityksen jälkeinen widgettien uudelleenrakennusmäärä.

7 TOTEUTUS

Teoriaosuudessa tarkasteltiin erilaisia Flutter-sovelluksen tilanhallintatekniikoita ja teoriaosuuden perusteella luodaan esimerkkisovellus, jossa hyödynnetään kyseisiä tekniikoita. Esimerkkisovelluksen käyttöliittymä tullaan pitämään mahdollisimman yksinkertaisena, jotta tilanhallintatekniikoita on helpompi verrata toisiinsa (Kuva 11).



Kuva 11. Esimerkkisovelluksen käyttöliittymä.

7.1 Käyttöliittymä

Esimerkkisovelluksen lähdekoodi koostuu Scaffold-widgetistä, joka implementoi Material-tyyliin perustuvat peruselementit. Tilamuutoksien laukaisimeen tullaan käyttämään FloatingActionButton-widgettiä. FloatingActionButton-widgetti on Material-tyyliin pohjautuva nappula, josta löytyy onPressed callback -funktio. onPressed callback -funktioon asetetaan erilaisia funktiota tai metodeja riippuen mitä tilanhallintatekniikkaa käytetään. Sovelluksen tilan näyttämisessä hyödynnetään yksinkertaista Text-widgettiä, joka mahdollistaa erilaisten arvojen näyttämisen sovelluksessa. Text-widget sidottu muuttujaan,

joka vastaa sovelluksen nykyistä tilaa. Esimerkkisovelluksen tila tulee olemaan tyyppiä Integer. (Liite 1.)

7.2 setState

SetState-funktio ei tarvitse erillistä asennusta, koska se on valmiina käytettäväksi Flutter ohjelmistokehyksessä. SetState-funktio ei toimi tilattomissa widgeteissä koska tilattomat widgetit eivät pysty hallinnoimaan omaa tilaansa. Tilallinen widget luodaan implementoimalla StatefulWidget abstrakti luokka (Koodi 11).

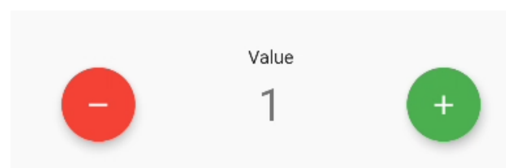
```
class ExampleApplication extends StatelessWidget
```

Koodi 11. Tilallisen widgetin luokan implementointi.

7.2.1 setState käyttö

Pystyäksemme käyttämään setState-funktiota, pitää esimerkkisovellus muuttaa käyttämällä tilallista widgettiä. Widgetin muuttamisen jälkeen setState-funktion runkoon asetetaan muutettava muuttuja ja muuttujan haluttu uusi arvo, jotka ovat tässä tapauksessa _counter ja _counter muuttujan nykyinen arvo lisättynä yhdellä. Asettamalla setState-funktion FloatingActionButton-widgettien onPressed callback -funktioihin pystymme pyytämään ohjelmistokehystä aikatauluttamaan widgetin uudelleenrakentamisen nappia painamalla (Kuva 12). (Liite 2.)

```
onPressed: () => setState(() {
  _counter++;
}),
```



Kuva 12. setState-funktion implementointi onPressed callback -funktioon

7.3 Provider

Provider paketti asennetaan käyttämällä Pub-paketinhallintaa. Provider paketin lisääminen projektiin onnistuu asettamalla pubspec.yaml tiedostoon uuden Provider riippuvaisuuden (Koodi 12). Riippuvaisuuden lisäämisen jälkeen suoritetaan flutter pub get asennuskomento komentorivillä (Koodi 13).

```
dependencies:
  flutter:
    sdk: flutter

  provider: ^4.1.3
```

Koodi 12. pubspec.yaml tiedosto.

```
flutter pub get
```

Koodi 13. Riippuvaisuuksien asennuskomento.

7.3.1 Provider käyttö

Sovelluksen yksinkertaisuuden vuoksi tulemme käyttämään ChangeNotifierProvideria, joka altistaa informaation alapuolella oleville widgeteille ja antaa mahdollisuuden informaatio muutoksien kuunteluun (Koodi 14).

```
ChangeNotifierProvider<Counter>(
create: (context) => Counter());
```

Koodi 14. ChangeNotifierProvider-widget.

ChangeNotifierProvider-widgetissä on parametrina create, joka vaatii luokan, joka implementoi ChangeNotifier mixin luokan. ChangeNotifier-luokka mahdollistaa informaatio muutoksien kuuntelun. Luokan nimeksi asetamme Counter ja luomme value-muuttujan, jota tulemme muokkaamaan increment ja decrement -metodeilla. Metodien toinen

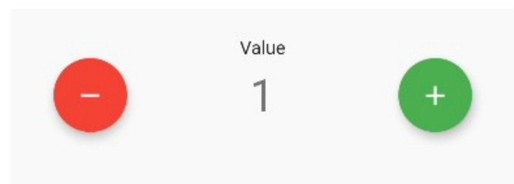
tehtävä muokkaamisen lisäksi on ilmoittaa value-muuttujan muutoksista käyttämällä notifylisteners-metodia (Koodi 15).

```
class Counter with ChangeNotifier {
  int value = 0;

  void increment() {
    value += 1;
    notifyListeners();
  }

  decrement() {
    value -= 1;
    notifyListeners();
  }
}
```

Koodi 15. ChangeNotifier-luokka.

FloatingActionButton-widgetteihin onPressed callback -funktioon asetetaan Provider.of-funktio, joka pyytää parent-widgetiltä lähintä Counter-objektia. Pyynnön jälkeen pystymme kutsumaan increment- ja decrement-metodeja Counter-luokasta (Kuva 13).

```
FloatingActionButton(
  key: Key('decrement'),
  backgroundColor: Colors.red,
  onPressed: () =>
      Provider.of<Counter>(context).decrement(),
  tooltip: 'Decrement',
  child: Icon(
    Icons.remove,
  ),
),
FloatingActionButton(
  key: Key('increment'),
  backgroundColor: Colors.green,
  onPressed: () =>
      Provider.of<Counter>(context).increment(),
  tooltip: 'Increment',
  child: Icon(Icons.add),
),
```



Kuva 13. Decrement- ja Increment-metodien kutsu.

Counter-luokan value muuttujaa käytetään Text-widgetissä, jota ympäröi Consumer-widget. Consumer widgetin tarkoituksena on kuunnella tulevia muutoksia Counter-luokasta. Consumer-widget tullaan rakentamaan uudelleen, kun Counter-luokan increment tai decrement -metodia kutsutaan (Kuva 14).

```
Consumer<Counter>(
  builder: (context, counter, child) => Text(
    '${counter.value}',
    key: Key('counter'),
    style: Theme.of(context).textTheme.display1,
  ),
),
```

Kuva 14. Consumer-widgetin käyttö.

Provider tilanhallintateknologian implementoinnin lähdekoodi esimerkkisovellukseen kokonaisuudessaan löytyy liitteistä (Liite 3).

7.4 Flutter BloC

Flutter BloC paketti asennetaan samalla tavalla kuin Provider. Flutter BloC riippuvaisuus lisätään pubspec.yaml tiedostoon, jonka jälkeen suoritetaan flutter pub get asennuskomento komentorivillä (Koodit 13 ja 16).

```
dependencies:
  flutter:
    sdk: flutter

  flutter_bloc: ^6.0.1
```

Koodi 16. pubspec.yaml tiedosto.

7.4.1 Flutter BloC käyttö

Sovelluksessa käytetään BlocProvider-widgettiä, joka altistaa BloC-logiikkatason alapuolella oleville widgeteille ja antaa mahdollisuuden BloC-logiikkatason muutoksien kuunteluun ja kutsumiseen (Koodi 17).

```
BlocProvider(
create: (BuildContext context) => CounterBloc(),
```

Koodi 17. BlocProvider-widget.

BlocProvider-widgetissä on parametrina create, joka vaatii luokan, joka jatkaa abstraktia BloC-luokkaa. BloC-luokan jatkaminen vaatii mapEventToState-metodin yliajamisen. MapEventToState-metodin tehtävänä on vastaanottaa tulevia tapahtumia ja lähettää tarvittaessa uusia tapahtumia käyttöliittymätasolle (Koodi 18). BloC-luokan jatkaminen vaatii myöskin super-luokan rakentajan initialState parametrin yliajamisen, joka asettaa BloC-logiikkatason oletus tilan (Koodi 19).

```
class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.decrement:
        yield state - 1;
        break;
      case CounterEvent.increment:
        yield state + 1;
        break;
      default:
        throw Exception('Incorrect event!');
    }
  }
}
```

Koodi 18. mapEventToState-metodi.

```
class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);
```

Koodi 19. Logiikkataso-luokan rakentaja.

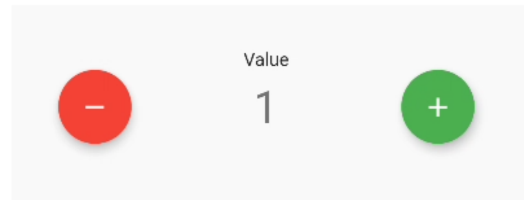
Flutter BloC tarvitsee tila-luokan ja logiikkatason lisäksi myös tapahtumaluokan. Tapahtumaluokkana sovelluksen yksinkertaisuuden vuoksi tullaan käyttämään lueteltua tyyppiä, jossa on tunnukset increment ja decrement (Koodi 20).

```
enum CounterEvent { increment, decrement }
```

Koodi 20. Tapahtuma tyyppi.

FloatingActionButton-widgetteihin onPressed callback -funktioon asetetaan BlocProvider.of-funktio, joka pyytää parent-widgetiltä lähintä CounterBloc-objektia. Pyynnön jälkeen pystymme lähettämään CounterEvent.increment ja CounterEvent.decrement tapahtumia BloC-logiikkatasolle add-metodia hyödyntäen (Kuva 15).

```
FloatingActionButton(
  key: Key('decrement'),
  backgroundColor: Colors.red,
  onPressed: () => BlocProvider.of<CounterBloc>(context)
      .add(CounterEvent.decrement),
  tooltip: 'Decrement',
  child: Icon(
    Icons.remove,
  ),
),
FloatingActionButton(
  key: Key('increment'),
  backgroundColor: Colors.green,
  onPressed: () => BlocProvider.of<CounterBloc>(context)
      .add(CounterEvent.increment),
  tooltip: 'Increment',
  child: Icon(Icons.add),
),
```



Kuva 15. CounterEvent.decrement ja CounterEvent.increment tapahtumien lähetys.

Text-widgetissä käytetään state-arvoa, joka vastaanotetaan BlocBuilder-widgetiltä. BlocBuilder rakentaa Text-widgetin uudelleen, kun se vastaanottaa uuden tapahtuman BloC-logiikkatasolta (Kuva 16).

```
BlocBuilder<CounterBloc, int>(
  builder: (context, state) => Text(
    '$state',
    key: Key('counter'),
    style: Theme.of(context).textTheme.display1,
  ),
),
```

Kuva 16. BlocBuilder-widgetin käyttö.

Flutter BloC tilanhallintateknologian implementoinnin lähdekoodi esimerkkisovellukseen kokonaisuudessaan löytyy liitteistä (Liite 4).

8 TESTAUS

Käyttämällä Flutter ohjelmistokehyksen ulkopuolisia kirjastoja voi sovelluksen suorituskyky muuttua. Seuraavissa testeissä verrataan miten erilaiset tilanhallintatekniikat vaikuttavat sovelluksen suorituskykyyn.

Testit suoritettiin käyttämällä Flutter v.1.20.4 julkaisua profile-tilassa ja OnePlus3T (Android 9) laitteella.

8.1 Käynnistysaika

Sovelluksen käynnistysajan mittaamiseen käytettiin flutter run komentoa, johon määriteltiin trace-startup –profile parametri (Koodi 21). Komento mittaa sovelluksen mittaa neljää erilaista aikaa: Aikaa kunnes Flutter alustetaan, aikaa kunnes sovelluksen ensimmäinen kuva renderoidaan, aikaa kuinka kauan Flutter ohjelmistokehys alustetaan ja aikaa kunnes Flutter ohjelmistokehyksen alustus on valmis. Komento käyttää aikamääreenä mikrosekuntia, mutta testien yhteneväisyyden vuoksi ajat on muutettu millisekunneiksi.

Hyvä indikaattori sovelluksen käynnistysajasta on, kun ensimmäinen kuva on renderöity.

```
flutter run --trace-startup --profile
```

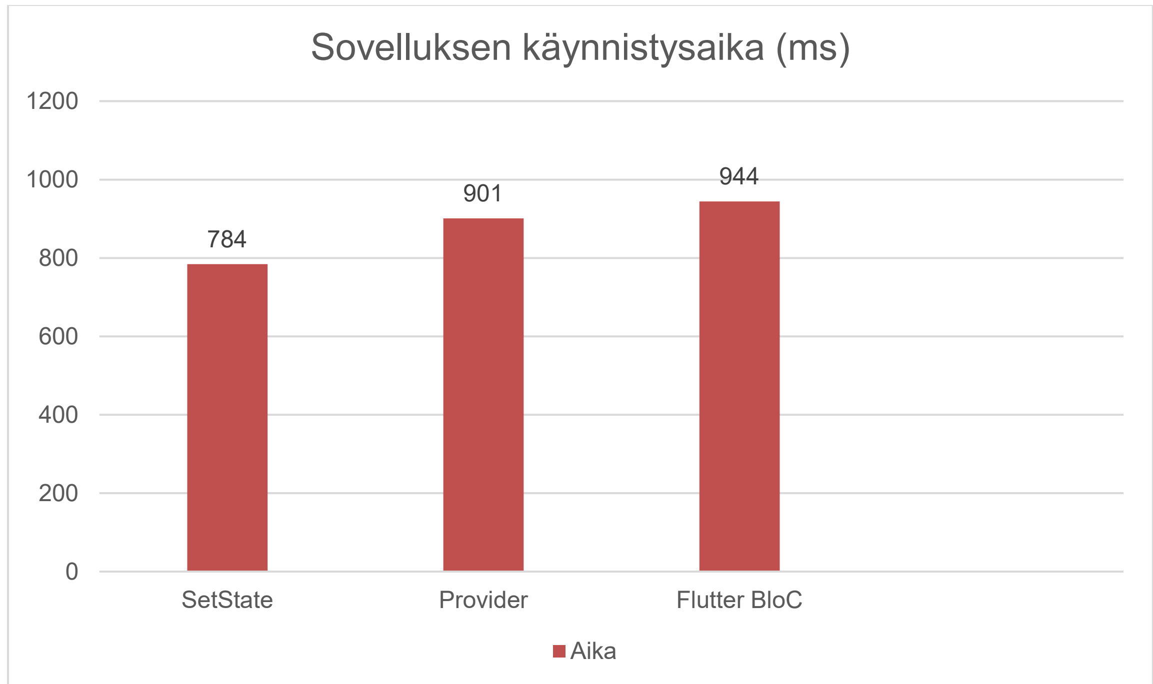
Koodi 21. Sovelluksen käynnistysajan mittaus.

```
{
  "engineEnterTimestampMicros": 326051222501,
  "timeToFrameworkInitMicros": 388254,
  "timeToFirstFrameMicros": 784459,
  "timeAfterFrameworkInitMicros": 396205
}
```

Koodi 22. Trace-startup komennon esimerkkituloste.

Käynnistysajaltaan setState esimerkkisovellus on selkeästi nopeampi kuin Provider ja Flutter BloC esimerkkisovellukset (Kuva 17). Flutter BloC ja Provider -kirjastot joutuvat tekemään alustuksia ennen kuin sovellus on valmis käynnistymään. Suositeltavaa on

sijoittaa Provider- ja BlocProvider-widgetit niin alas widget-puuhun kuin mahdollista jotta alustuksia ei tapahdu ennen kuin Provider tai BlocProviderin tarjoamaa informaatiota tarvitaan. Vaikka setState on nopeampi, ei normaalissa sovelluksen käytössä loppukäyttäjä tule juurikaan huomaamaan eroa.



Kuva 17. Esimerkkisovelluksen käynnistysajan testin tulokset.

8.2 Ajonaikainen suorituskyky

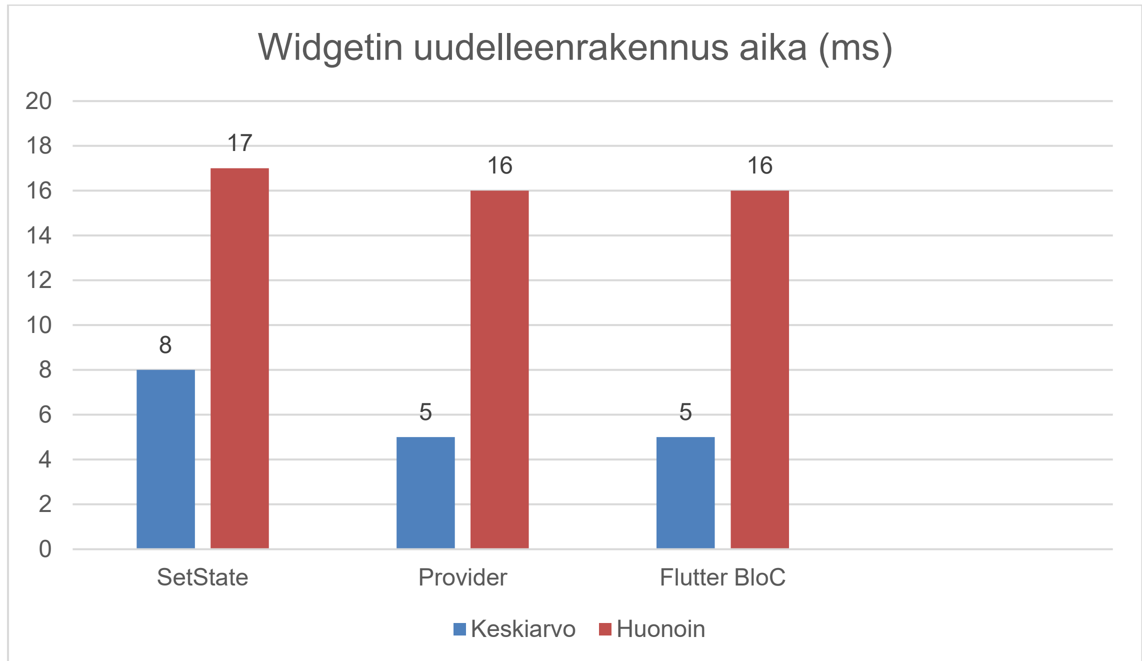
Sovelluksen ajonaikana suorituskyvynasteikkona käytettiin widgetin uudelleenrakennusaikaa. Suorituskyvyn testaamiseen käytettiin esimerkkisovellukseen luotua testiä, joka hyödynsi flutter driver- ja test-kirjastoja (Liite 5). Testi simuloi sovelluksen käyttöä painamalla lisää ja poista nappuloita. Testi käyttää aikamääreenä millisekunteja (Koodi 23).

```
Avg frame build time: 5.9912551020408165ms
Worst frame build time: 17.08ms
```

Koodi 23. Suorituskykytestin esimerkkituloste.

Ajonaikaiselta suorituskyvyltään Provider- ja Flutter BloC -esimerkkisovellukset ovat samankaltaisia, kun taas setState jää jonkun verran jälkeen (Kuva 18). Provider ja Flutter BloC pystyvät rajaamaan widgettien uudelleenrakennusmäärää, kun taas SetState

rakentaa aina koko widgetin uudelleen. Esimerkkisovelluksen yksinkertaisuuden takia suorituskykyerot ovat suhteellisen pienet ja loppukäyttäjän näkökulmasta eroa ei huomaa.



Kuva 18. Esimerkkisovelluksen suorituskykytestauksen tulokset.

9 YHTEENVETO

Opinnäytetyön tavoitteena oli verrata ja testata kolmea erilaista Flutter-ohjelmistokehyksen tilanhallintatekniikkaa ja luoda niistä johtopäätökset jota voidaan hyödyntää tilanhallintatekniikkaa valittaessa. Tilanhallintatekniikoiksi valittiin setState, Provider ja Flutter BloC niiden suosion ja käytettävyyden vuoksi.

Opinnäytetyön teoriaosuuden alussa luotiin teknologiakatsaus Flutter-ohjelmistokehykseen ja Dart-ohjelmointikieleen, jotka luovat perusteet Flutter tilanhallinnan ymmärtämiseen. Tilanhallintatekniikoita tutkittiin niiden hallittavuuden, helppokäyttöisyyden ja suorituskyvyn näkökulmasta.

Opinnäytetyön toteutus osuudessa luotiin kolme käyttöliittymältään samanlaista esimerkkisovellusta, joissa hyödynnettiin valittuja tilanhallintatekniikoita. Esimerkkisovelluksen yksinkertaisuuden vuoksi suorituskyvyn mittaus oli helppoa, mutta koodin hallittavuuden tarkastelu oli vaikeampaa.

Hallittavuuden kannalta Flutter BloC on hallittavampi koodikannan kasvaessa kuin Provider ja setState. Provider ja setState eivät ohjaa kehittäjää mihinkään ohjelmistoarkkitehtuuriin, kun taas Flutter BloC hyödyntää Business Logic Component kaavaa, joka edesauttaa koodikannan hallittavuutta. SetState tilanhallintatekniikka on yksinkertaisin kaikista valituista tilanhallintatekniikoista. Suorituskyvyltään Provider ja Flutter BloC ovat samalla tasolla.

Kaikissa tilanhallintatekniikoissa on hyvät ja huonot puolensa. Kehittäjän kannalta on tärkeää miettiä mitä asioita arvostaa ja mitä asioita tulee projektissaan tarvitsemaan. Jos kehittäjä on luomassa ison mittakaavan sovellusta, jossa koodikannan hallittavuus on tärkeää, suosittelen Flutter BloC tilanhallintakirjastoa. Provider tilanhallintatekniikkaa kannattaa silloin käyttää, jos ei pidä tai halua käyttää tapahtumapohjaista tilanhallintatekniikkaa kuten Flutter BloC. SetState tilanhallintatekniikkaa suosittelen käyttämään Flutter BloC ja Provider tilanhallintatekniikoiden tukena, jolloin kehittäjä pystyy välttämään liiallista ja turhaa koodin kirjoittamista yksinkertaisissa widgeteissä.

Flutter on nuori ohjelmistokehys ja tilanhallintatekniikoita julkaistaan nopealla tahdilla. Kehittäjän on tärkeä ymmärtää valita oikea tilanhallintatekniikka projektin mukaan eikä sen perusteella mikä on tällä hetkellä uutta ja hienoa.

Opinnäytetyö eteni suunnitellusti ja sopivalla tahdilla. Flutter-ohjelmistokehykseen löytyy valtava määrä dokumentaatiota nuoreen ikäänsä nähden, joka helpotti opinnäytetyön suunnittelussa ja tekemisessä. Tartuin tähän koska halusin kasvattaa ymmärrystäni Flutter-ohjelmistokehyksestä ja sen tilanhallintatekniikoista. Opinnäytetyössä parannuksia olisi ollut arkkitehtuurisesti vaativimpien sovelluksien luonti, jolloin olisin pystynyt paremmin vertaamaan erilaisten tilanhallintatekniikoiden hyötyä suurimmissa kokonaisuuksissa.

LÄHTEET

Bloc 2020. Architecture. Viitattu 8.6.2020 https://bloclibrary.dev/#/architecture.

Dart 2020a. FAQ. Viitattu 14.4.2020 https://dart.dev/faq.

Dart 2020b. Dart. Viitattu 14.4.2020 https://dart.dev/faq.

Dart 2020c. Stream<T> class. Viitattu 7.6.2020 https://api.dart.dev/stable/2.8.4/dart-async/Stream-class.html.

Flutter 2020a. Flutter. Viitattu 26.3.2020 https://flutter.dev/.

Flutter 2020b. Start thinking declaratively. Viitattu 26.3.2020 https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative.

Flutter 2020c. FAQ. Viitattu 1.4.2020 https://flutter.dev/docs/resources/faq.

Flutter 2020d. FAQ. Viitattu 1.4.2020 https://flutter.dev/docs/development/tools/sdk/releases.

Flutter 2020e. Technical overview. Viitattu 21.4.2020 https://flutter.dev/docs/resources/technical-overview.

Flutter 2020f. InheritedWidget class. Viitattu 3.5.2020 https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.

Flutter 2020g. setState. Viitattu 3.5.2020 https://api.flutter.dev/flutter/widgets/State/setState.html.

Flutter 2020h. Performance best pracises. Viitattu 3.5.2020 https://flutter.dev/docs/perf/rendering/best-practices.

Flutter 2020i. BuildContext class. Viitattu 10.5.2020 https://api.flutter.dev/flutter/widgets/BuildContext-class.html.

Flutter 2020j. Simple app state management. Viitattu 31.5.2020 https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple.

Flutter 2020k. getElementForInheritedWidgetOfExactType. Viitattu 27.9.2020 https://api.flutter.dev/flutter/widgets/BuildContext/getElementForInheritedWidgetOfExactType.html.

Flutter 2020l. Differentiate between ephemeral state and app state. Viitattu 27.9.2020 https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app.

Flutter 2020m. Introduction to declarative UI. Viitattu 27.9.2020 https://flutter.dev/docs/get-started/flutter-for/declarative.

GitHub 2020. Provider. Viitattu 24.5.2020 https://github.com/rrousselGit/provider.

Google Developers 2020a. Inherited Widgets Explained - Flutter Widgets 101 Ep. 3. Viitattu 30.5.2020 https://www.youtube.com/watch?v=Zbm3hjPjQMk.

Google Developers 2020b. Build reactive mobile apps with Flutter (Google I/O '18). Viitattu 7.6.2020 https://www.youtube.com/watch?v=RS36gBEp8OI.

Pub.dev 2020. flutter_bloc package. Viitattu 9.6.2020 https://pub.dev/documentation/flutter_bloc/latest/.

Skills matter 2020. The real reason why you should care about Provider Viitattu 23.5.2020 https://skillsmatter.com/skillscasts/14025-flutter-london-june-meetup.

Strange Loop 2020. ""Flutter: How we're building a UI framework for tomorrow at Google" by Eric Seidel. Viitattu 14.4.2020 https://www.youtube.com/watch?v=VUiVkDpikDI.

Esimerkkisovelluksen käyttöliittymän lähdekoodi

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

// ignore: must_be_immutable
class ExampleApplication extends StatelessWidget {
  int _counter = 0;
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Flutter',
        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: Scaffold(
          appBar: AppBar(
            title: Text('Flutter'),
            centerTitle: true,
          ),
          body: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Text(
                'Value',
              ),
              Row(
                mainAxisAlignment: MainAxisAlignment.spaceAround,
                children: <Widget>[
                  FloatingActionButton(
                    backgroundColor: Colors.red,
                    onPressed: null,
                    tooltip: 'Decrement',
                    child: Icon(
                      Icons.remove,
                    ),
                  ),
                  Text(
                    '$_counter',
                    style: Theme.of(context).textTheme.display1,
                  ),
                  FloatingActionButton(
                    backgroundColor: Colors.green,
                    onPressed: null,
                    tooltip: 'Increment',
                    child: Icon(Icons.add),
                  ),
                ],
              )
            ],
          ),
        ));
  }
}
```

Esimerkkisovellus - setState

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

class SetStateApplication extends StatefulWidget {
  @override
  _SetStateApplicationState createState() => _SetStateApplicationState();
}

class _SetStateApplicationState extends State<SetStateApplication> {
  int _counter = 0;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Flutter',
        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: Scaffold(
          appBar: AppBar(
            title: Text('Flutter'),
            centerTitle: true,
          ),
          body: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Text(
                'Value',
              ),
              Row(
                mainAxisAlignment: MainAxisAlignment.spaceAround,
                children: <Widget>[
                  FloatingActionButton(
                    key: Key('decrement'),
                    backgroundColor: Colors.red,
                    onPressed: () => setState(() {
                      _counter--;
                    }),
                    tooltip: 'Decrement',
                    child: Icon(
                      Icons.remove,
                    ),
                  ),
                  Text(
                    '$_counter',
                    key: Key('counter'),
                    style: Theme.of(context).textTheme.display1,
                  ),
                  FloatingActionButton(
                    key: Key('increment'),
                    backgroundColor: Colors.green,
                    onPressed: () => setState(() {
                      _counter++;
                    }),
                    tooltip: 'Increment',
                    child: Icon(Icons.add),
                  ),
                ],
              )
            ],
          ),
        ));
  }
}
```

Esimerkkisovellus - Provider

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

class Counter with ChangeNotifier {
  int value = 0;

  void increment() {
    value += 1;
    notifyListeners();
  }

  decrement() {
    value -= 1;
    notifyListeners();
  }
}

class ProviderApplication extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Provider<Counter>(
      create: (context) => Counter(),
      child: MaterialApp(
          debugShowCheckedModeBanner: false,
          title: 'Flutter',
          theme: ThemeData(
            primarySwatch: Colors.blue,
          ),
          home: Scaffold(
            appBar: AppBar(
              title: Text('Flutter'),
              centerTitle: true,
            ),
            body: Builder(
              builder: (context) => Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Text(
                    'Value',
                  ),
                  Row(
                    mainAxisAlignment: MainAxisAlignment.spaceAround,
                    children: <Widget>[
                      FloatingActionButton(
                        key: Key('decrement'),
                        backgroundColor: Colors.red,
                        onPressed: () =>
                            Provider.of<Counter>(context, listen: false)
                                .decrement(),
                        tooltip: 'Decrement',
                        child: Icon(
                          Icons.remove,
                        ),
                      ),
                      Consumer<Counter>(
                        builder: (context, counter, child) => Text(
                          '${counter.value}',
                          key: Key('counter'),
                          style: Theme.of(context).textTheme.display1,
                        ),
                      ),
                      FloatingActionButton(
                        key: Key('increment'),
                        backgroundColor: Colors.green,
                        onPressed: () =>
                            Provider.of<Counter>(context).increment(),
                        tooltip: 'Increment',
                        child: Icon(Icons.add),
                      ),
                    ],
                  )
                ],
              ),
            ),
          )),
    );
  }
}
```

Esimerkkisovellus – BloC

```
import 'dart:async';

import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.decrement:
        yield state - 1;
        break;
      case CounterEvent.increment:
        yield state + 1;
        break;
      default:
        throw Exception('Incorrect event!');
    }
  }
}

class BlocApplication extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (BuildContext context) => CounterBloc(),
      child: MaterialApp(
          debugShowCheckedModeBanner: false,
          title: 'Flutter',
          theme: ThemeData(
            primarySwatch: Colors.blue,
          ),
          home: Scaffold(
            appBar: AppBar(
              title: Text('Flutter'),
              centerTitle: true,
            ),
            body: Builder(
              builder: (context) => Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Text(
                    'Value',
                  ),
                  Row(
                    mainAxisAlignment: MainAxisAlignment.spaceAround,
                    children: <Widget>[
                      FloatingActionButton(
                        key: Key('decrement'),
                        backgroundColor: Colors.red,
                        onPressed: () => BlocProvider.of<CounterBloc>(context)
                            .add(CounterEvent.decrement),
                        tooltip: 'Decrement',
                        child: Icon(
                          Icons.remove,
                        ),
                      ),
                      BlocBuilder<CounterBloc, int>(
                        builder: (context, state) => Text(
                          '$state',
                          key: Key('counter'),
                          style: Theme.of(context).textTheme.display1,
                        ),
                      ),
                      FloatingActionButton(
                        key: Key('increment'),
                        backgroundColor: Colors.green,
                        onPressed: () => BlocProvider.of<CounterBloc>(context)
                            .add(CounterEvent.increment),
                        tooltip: 'Increment',
                        child: Icon(Icons.add),
                      ),
                    ],
                  )
                ],
              ),
            ),
          )),
    );
  }
}
```

Esimerkkisovellus – Testi

```
import 'package:flutter_driver/flutter_driver.dart';
import 'package:test/test.dart';

void main() {
  group('Counter App', () {
    final counterTextFinder = find.byValueKey('counter');
    final incrementButtonFinder = find.byValueKey('increment');
    final decrementButtonFinder = find.byValueKey('decrement');

    FlutterDriver driver;

    setUpAll(() async {
      driver = await FlutterDriver.connect();
    });

    tearDownAll(() async {
      if (driver != null) {
        driver.close();
      }
    });

    test('starts at 0', () async {
      expect(await driver.getText(counterTextFinder), "0");
    });

    test('modify counter value', () async {
      final timeline = await driver.traceAction(() async {
        for (int i = 0; i <= 50; i++) {
          await driver.tap(incrementButtonFinder);
        }
        for (int i = 0; i <= 50; i++) {
          await driver.tap(decrementButtonFinder);
        }

        expect(await driver.getText(counterTextFinder), "0");
      });

      final summary = TimelineSummary.summarize(timeline);

      print('Avg frame build time:'
          ' ${summary.computeAverageFrameBuildTimeMillis()}ms');
      print('Worst frame build time:'
          '  ${summary.computeWorstFrameBuildTimeMillis()}ms');
    });
  });
}
```