State management in Flutter: een vergelijkende studie.

Angela Degryse.

Scriptie voorgedragen tot het bekomen van de graad van
Professionele bachelor in de toegepaste informatica

Promotor: Sofie Lambert
Co-promotor: Tim Alenus
Instelling: The Mobility Factory
Academiejaar: 2022–2023
Eerste examenperiode

Departement IT en Digitale Innovatie .

HO
GENT

Woord vooraf

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Samenvatting

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Inhoudsopgave

Lijst van figuren vii

1 Inleiding 1
- 1.1 Probleemstelling . . . 1
- 1.2 Onderzoeksvraag . . . 1
- 1.3 Onderzoeksdoelstelling . . . 2
- 1.4 Opzet van deze bachelorproef . . . 2

2 Stand van zaken 3
- 2.1 Flutter . . . 3
- 2.2 Flutter Architectuur . . . 3
  - 2.2.1 widget . . . 4
  - 2.2.2 Widget rendering . . . 4
  - 2.2.3 states . . . 5
  - 2.2.4 widget states . . . 5
- 2.3 State management . . . 6
  - 2.3.1 setState . . . 6
  - 2.3.2 Inherited widget . . . 7
  - 2.3.3 Provider . . . 9
  - 2.3.4 RiverPod . . . 11
  - 2.3.5 Redux . . . 12
  - 2.3.6 Bloc . . . 15

3 Methodologie 17

4 Conclusie 19

A Onderzoeksvoorstel 21
- A.1 Introductie . . . 21
- A.2 State-of-the-art . . . 22
  - A.2.1 Application performance, een onderdeel van User Experience . . . 22
  - A.2.2 Technieken om applicatie performantie te verbeteren . . . 22
  - A.2.3 State management binnen Flutter . . . 23
  - A.2.4 Relevante studies . . . 24
- A.3 Methodologie . . . 24
- A.4 Verwachte resultaten . . . 25
- A.5 Discussie, conclusie . . . 25

Bibliografie 26

Lijst van figuren

2.1 Voorbeeld van een widgetboom 4
2.2 De 3 widget rendering bomen van Flutter (Flutter, 2023) 5
2.3 Lifecycle van een stateless and stateful widget (Goel, 2023) 6
2.4 Redux lifecycle (Sharma, 2018) 13
2.5 BLoC Architectuur (Bloc, 2023) 15

1

Inleiding

Cross-platform ontwikkelen wordt steeds populairder omdat er vanuit één code-base de applicatie op meerdere toestellen gedraaid kan worden. Er bestaan veel cross-platform frameworks en Flutter is één van de bekendste die in 2018 door Google ontwikkeld werd.

1.1. Probleemstelling

The Mobility Factory (TMF) is een coöperatie die IT-oplossingen biedt aan bedrijven die elektrische autodelen diensten aanbieden. De applicaties worden door de TMF-ontwikkelaars in Flutter geïmplementeerd. Omdat de applicaties van TMF steeds groter en complexer worden, werkt de applicatie steeds minder performant. Daarom kwam de vraag om de performantie van hun applicatie te optimaliseren. Een mogelijke reden van minder performante applicatie is door het slecht beheren van states. In Flutter zijn er tal van mogelijkheden voor het beheren van states. Afhankelijk van de complexiteit van de applicatie moet er overwogen worden welke state management benadering het best bij de applicatie past en of het de meest performante optie is.

1.2. Onderzoeksvraag

De hoofdonderzoekvraag van dit onderzoek is de volgende:
Op welk manier kan states bijgehouden worden in Flutter en welk past het bij bij de applicatie van The Mobility Factory?

Daarnaast kan de hoofdonderzoekvraag verder verdeeld worden in de volgende deelonderzoeksvragen:

- Welk benadering van state management is het performantste qua cpu-gebruik, opstartsnelheid, geheugengebruik...?

- Hoeveel moeite zal het The Mobility Factory kosten om de verschillende benaderingen van state management te implementeren? Hoe complex is het om te integreren in de bestaande code?

1.3. Onderzoeksdoelstelling

Het hoofddoel van dit onderzoek is om de developers van The Mobility Factory te helpen om een beslissing te maken of het veranderen van state management benadering een invloed kan hebben op de performantie van de applicatie. Ook zal er moeten overwogen worden of complexiteit om het te integreren in de bestaande code de moeite waard is. Dit wordt gedaan door middel van een vergelijkende studie. Eerst wordt er een literatuurstudie gedaan over de benaderingen van state management en daarna worden ze met elkaar vergeleken met behulp van een proof-of-concept.

1.4. Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:
In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.
In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.
In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2

Stand van zaken

In dit onderdeel wordt dieper ingegaan op wat flutter is hoe het werkt. Daarna wordt er onderzocht welke state management benaderingen er zijn en hoe ze werken en worden ze met elkaar vergeleken.

2.1. Flutter

Flutter is een open-source framework ontwikkeld door Google voor het bouwen van multi-platform applicaties vanuit één codebase. Dit betekent dus dat ontwikkelaars in een korte tijd applicaties kunnen ontwikkelen voor mobile, web, desktop en ook voor embedded systemen. Bovendien wordt flutter code vertaald naar native machine code, wat zorgt voor snelle applicaties en annimaties. [1]

Flutter applicaties worden geschreven met Dart, een programmeertaal die ook eigendom is van Google. Dart ondersteunt AOT (ahead of time) compilation en JIT (just in time) compilation. AOT verbetert de opstarttijd en performantie van de application en dankzij JIT compilation wordt hot reload mogelijk. Dit wilt zeggen dat enkel de gewijzigd stukken code opnieuw moet gecompileerd worden in plaats van de volledige applicatie. Het is m.a.w. mogelijk om wijzigingen in bijna real time te testen. [2]

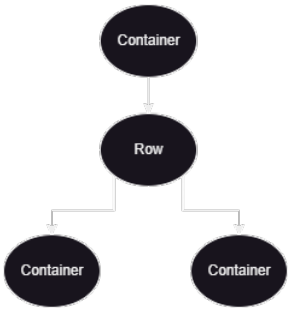
2.2. Flutter Architectuur

Voordat er over state management binnen flutter besproken kan worden, wordt er eerst een paar begrippen uitgelegd binnen Flutter. Ook wordt er in dit deel uitgelegd hoe Flutter werkt intern.

[1] https://flutter.dev
[2] https://docs.dart.dev

2.2.1. Widgets

In Flutter wordt alle componenten in een app als een ”Widget” beschouwd. Wat een widget is, is eigenlijk een Dart-klasse die beschrijft hoe een stuk van de user interface eruitziet. De UI van de app wordt m.a.w. gebouwd door widgets in ouder-kind relatie te combineren. Elk widget wordt in een parent widget genest en kan context doorgeven van ouder naar kind. Als gevolg hiervan zitten alle widgets in een widgetboom, waarbij elk ouderwidget één of meerdere kind widgets kan hebben.



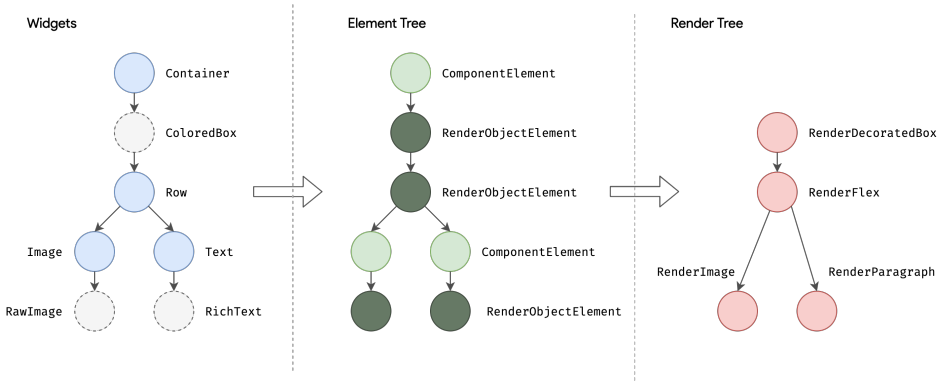
Figuur (2.1)
Voorbeeld van een widgetboom

Elk widget is immutable of onveranderlijk. Wanneer zich een gebeurtenis voordoet, zoals een gebruikersinteractie, veranderen applicaties hun gebruikersinterface door het framework te vertellen om een widget in de hiërarchie te vervangen door een andere. Het framework werkt als gevolg de gebruikersinterface efficiënt bij door de ouderwidgets te vergelijken met de nieuwe. Indien er een verandering is aan de nieuwe widget, wordt de oude dan vervangen met de nieuwe widget.

2.2.2. Widget rendering

In de vorige sectie wordt er uitgelegd wat een widget is en dat Flutter een widgetboom heeft. Maar hoe kan flutter widgets efficiënt widgets kunnen gebouwen? Dit vraag wordt in dit sectie uitgelegd.
Flutter heeft eigenlijk niet enkel 1 boom. Flutter maakt namelijk gebruik van 3 parallel bomen voor het renderen van widgets, die samen bijdragen aan de efficiëntie van een Flutter applicatie. (Adnan, 2023)

- **Widgetboom**
  Een widget in een widgetboom vertegenwoordigt de UI-structuur en configuratie.
- **Elementboom**
  Tijdens de bouwfase vertaalt Flutter de widgets naar een overeenkomstige elementenboom, met één element voor elke widget. De elementboom beheert updates, wijzigingen van de UI en regelt alles.



Figuur (2.2)
De 3 widget rendering bomen van Flutter (Flutter, 2023)

- **Renderboom**
  Een renderobject in een renderboom is verantwoordelijk voor het finale schilderen van widgets op de User Interface. Het definieert hoe elementen visueel worden weergegeven op het scherm. De renderobject zorgt voor de grootte, lay-out en het schilderen op het scherm. Wat je uiteindelijk op het scherm ziet is geen widget, maar een renderobject.

2.2.3. States

Elk widget kan states hebben. Flutter beschrijft een state als informatie die enerzijds synchroon kan worden gelezen wanneer de widget wordt gebouwd en anderzijds kan veranderen tijdens de levensduur van de widget[3]. Flutter is declaratief, dit wilt zeggen dat dat Flutter zijn gebruikersinterface bouwt om de huidige status van je app te weerspiegelen. Indien er een verandering is aan een state, wordt de widget getriggerd en wordt de UI herbouwd.

2.2.4. Widget states

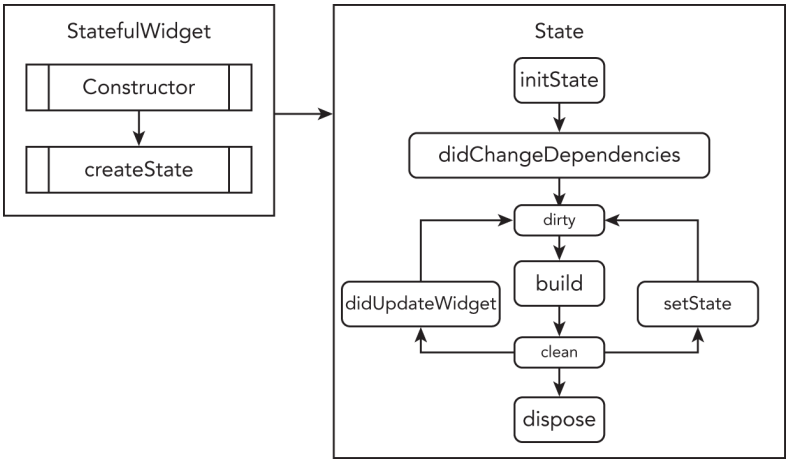
Nu dat er een beeld is over wat states zijn binnen Flutter, kan er over de 2 meest gebruikte soorten widgets besproken worden.

- **stateless widget**
  Stateless widgets, zoals de naam het al zegt, zijn widgets zonder states. Deze widgets worden gedurende de lifecycle van de applicatie niet gewijzigd. Een voorbeeld hiervan is een Icon-widget. Het houdt intern geen states bij en weergeeft altijd een vaste icon.

- **stateful widget**
  Stateful widgets daarentegen kunnen wel state-objecten hebben die kunnen veranderen. Een stateful widget heeft 2 klasse: een klasse van ”StatefulWidget” en een klasse ”State”. De state-klasse bevat de veranderbaar state van de

[3] https://api.flutter.dev/flutter/widgets/State-class.html

widget. Als een widget state wijzijgt door de methode *setState()* op te roepen, wordt de widget getriggerd om zichzelf te herbouwen en worden alle afstammelingen van deze widget binnen de widgetboom vergeleken met hun oude widget met behulp van de *didWidgetUpdate*-methode en indien er een verandering is worden ze ook herbouwd.
Een voorbeeld van een stateful widget is een Checkbox widget. Indien de user op een checkbox klikt, wordt de state *isChecked* nu gelijk aan *true*. Als gevolg van deze state verandering, wordt de widget opniew gebouwd met een vinkje op de checkbox widget.



Figuur (2.3)
Lifecycle van een stateless and stateful widget (Goel, 2023)

2.3. State management

Zoals eerder vermeld wordt Flutter ontwikkeld om performant te zijn. Maar een mogelijke reden die de performantie van de applicatie kan beïnvloeden is het slecht bijhouden van states binnen de applicatie.
State management is een techniek (of meerdere) die gebruikt wordt om veranderingen in de applicatie te beheren. Het gebruik van de juiste state management kan zorgen voor betere leesbaarheid van code en ook voor een performanter applicatie.

2.3.1. setState

setState is het gemakkelijkste manier om states te veranderen binnen een flutter applicatie. Wanneer de *setState*-method binnen een stateful widget opgeroepen wordt, wordt de *build*-method van die widget opgeroepen en wordt het herbouwd met de nieuwe state. Zoals eerder vermeld, als een ouder widget herbouwd wordt, worden er een check gedaan of alle afstammelingen van die ouder herbouwd moeten worden.

```
    class CheckBoxWidget extends StatefulWidget {
  final bool isChecked;
  const CheckBoxWidget({super.key, required this.isChecked});

  @override
  State<CheckBoxWidget> createState() => CheckBoxWidgetState();
}

class CheckBoxWidgetState extends State<CheckBoxWidget> {
  late bool isChecked;
  @override
  void initState() {
    isChecked = widget.isChecked;
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Checkbox(
        value: isChecked,
        onChanged: (value) {
          setState(() {
            isChecked = value!;
          });
        });
  }
}
```

Voor- en nadelen

Volgens Slepnev (2020) is het gebruik van setState heel eenvoudig en gemakkelijk te leren, maar er zijn ook wat nadelen aan het gebruik van setState.

- Indien een kind diep in de widgetboom een state nodig heeft van een parent aan de top van de boom, moet de state aan alle afstammelingen binnen de widgetboom doorgegeven worden. Dit zorgt voor minder leesbare code en wordt het onderhouden van de code moeilijker.
- Er is geen scheiding van views en business logica.

2.3.2. Inherited widget

Inherited widget lost het probleem op van setState waarbij je properties door de afstammelingen in een widgetboom moet meegeven. Met inherited widget krijgt

alle widgets die binnen de inherited widget subboom zitten toegang tot de inherited widget. Als gevolg hiervan hoeft de properties niet meer aan de kinderen doorgegeven worden, omdat de widgets de properties rechtstreeks kunnen krijgen van de inheritedwidget. (Windmill, 2020)
Als een widget binnen de widgetboom informatie nodig heeft van een inherited widget, kan die widget gebruik maken van de *of*-methode van de inherit widget. Deze methode zoekt in de boomstructuur en vindt de dichtstbijzijnde ouder-inheritedwidget van dat type. Een voorbeeld hiervan is als MediaQuery.of(context).size wordt opgeroepen. Flutter zoekt naar de dichstbijzijnde MediaQuery widget, en haalt de size property daarvan. Indien de waarde van de state van de inherited widget wijzigt, worden de consumers van die data dat getroffen is herbouwd.

```
class InheritColor extends InheritedWidget {
  const InheritColor({
    super.key,
    required this.color,
    required this.changeColor
    required super.child,
  });

  final Color color;
  final Function(Color color) changeColor;

  static InheritColor of(BuildContext context) {
    final InheritColor? result =
        context.dependOnInheritedWidgetOfExactType<InheritColor>();
    assert(result != null, 'No InheritColor found in context');
    return result!;
  }

  @override
  bool updateShouldNotify(InheritColor oldWidget) => color != oldWidget.color;
}
```

Dit is een voorbeeld van hoe je een inherit widget in een stateful widget kan zetten om zijn state te veranderen.

```
Color color = Colors.pink;
Widget build(BuildContext context) {
    return InheritColor(
      color: color,
      changeColor: (Color newColor){
```

```
        setState(()⇒color = newColor);
      },
      child: OtherChildWidget(),
    );
  }
```

Wanneer *InheritColor.of(context).changeColor(Colors.green)* wordt opgeroepen in een widget onder de inheritWidget, zal de *InheritColor*-widget herbouwd worden met een nieuwe Color-object en zullen alle kinderen die luisteren naar de inheritedWidget ook herbouwd worden.

Voor- en nadelen

Zoals voordien aangehaald is het grootste pluspunt van inherited widget dat states niet meer door alle constructors moeten indien een state nodig is in een widget op een lage niveau op de widgetboom. Een andere voordeel is dat InheritedWidget deel is van de Flutter API, er hoeft dus geen externe package te installeren.
Een nadeel van inherited widget is dat het geen stand alone state management benadering is. Het is afhankelijk van een stateful widget en zijn setstate methode om states van een inherited widget te wijzigen. Er is ook nog altijd boilerplate code nodig: er moet steeds twee klassen zijn: een inheritedWidget en een stateful/stateless widget.

2.3.3. Provider

Provider is de aanbevolen state management benadering volgens de Flutter team. Provider is een wrapper van InheritedWidget om de implementatie ervan gemakkelijker en meer herbruikbaar te maken. Door de provider te wrappen rond de applicatie, wordt de states van de provider toegankelijk voor alle widgets binnen de applicatie.
Volgens de documentatie van Provider[4] zijn er veel soorten Providers: Provider, ChangeNotifierProvider, ListenableProvider, ProxyProvider, FutureProvider, StreamProvider en veel meer. ChangeNotifierProvider is de provider die het meest wordt gebruikt.
In de code hieronder wordt een CarModel klasse uitgebreid(extended) met de **ChangeNotifier**. Een *ChangeNotifier* biedt change notifications(wijzigingemeldingen) aan de luisteraars. Een ChangeNotifier klasse heeft de methode *notifyListeners*. Dit methode kan opgeroepen worden indien de state wijzigt en een rebuild nodig is in de widget die luistert naar dit state.

```
class CarModel extends ChangeNotifier {
    Color _color = Colors.black;
```

[4] https://pub.dev/packages/provider

```
        Color get color ⇒ _color;

        void changeColor(Color newColor) {
             _count = newColor;
            notifyListeners();
        }
}
```

ChangeNotifierProvider is een widget die een instantie van ChangeNotifier biedt aan de afstammelingen in de widgetboom.

```
    Widget build(BuildContext context) {
    return ChangeNotifierProvider(
        create:  (context) ⇒ CarModel(),
        child: childWidget(),
    );
  }
```

Een **consumer** is een widget die luistert naar een Provider en geeft het mee aan de widget die het consumeert. Het is een best practice om je Consumer widgets zo diep mogelijk in de tree te plaatsen. Dit vermijdt dat grote delen van de UI wordt herbouwd omdat ergens een detail is veranderd.

```
    Widget build(BuildContext context) {
    return Consumer<CarModel>(
        builder: (context, car, child) {
          return Text('The car is ${car.color} in color.');
        },
      );
  }
```

Een alternatief voor het Consumer Widget is door context.read<T>() of context.watch<T>() te gebruiken. De watch methode luistert naar de changes van de Provider en triggert een rebuild indien er wijzigingen zijn aan de state van de provider. De read-methode daarentegen returnt T zonder te luisteren en triggert dus geen rebuilds. Op hetzelfde manier kan er ook gebruikt gemaakt worden van Provider.of<T>(context) om aan de provider te geraken.Er is een optionele parameter listen. Indien de UI niet hoeft te veranderen kan Provider.of<T>(context, listen: false) ook gebruikt worden.

Voor- en nadelen

Volgens Makadiya (2022) is het gebruik van Provider eenvoudiger dan InheritWidget omdat het de boilerplate code vermindert. Bovendien faciliteert het ook de schaalbaarheid van de applicatie.

Een nadeel hiervan is dat provider het onnodig heropbouwen van widgets kan veroorzaken. Als er een verandering is aan een state binnen de Provider, wordt alle widgets die naar dit state luistert verwittigd over de verandering en wordt het herbouwt, maar dit hoeft niet altijd te gebeuren.

2.3.4. RiverPod

Riverpod is een verbetering tegenover de Provider package. Het gebruikt de concepten van Provider en voegt functionaliteiten toe door het verbeteren van flexibiliteit en performantie.(Arshad, 2021) Riverpod kan netwerkverzoeken uitvoeren met ingebouwde foutafhandeling en caching, terwijl het automatisch gegevens opnieuw ophaalt als dat nodig is [5].
Om Riverpod te kunnen gebruiken moet er eerste een ProviderScope widget gewrapd worden rond de wortelwidget(rootwidget) van de widgeboom. Deze provider scope houdt alle providers bij en zorgt ervoor dat er geen lek is van de states ondanks dat de providers globaal worden gedeclareerd.

```
    void main() {
  runApp(
    ProviderScope(
      child: MyApp(),
    ),
  );
}
```

De provider van Riverpod zitten buiten de widgetboom. Om die providers te kunnen lezen is er nood aan een extra referentie-object. De eerste manier om de data van een provider te lezen is met behulp van een *ConsumerWidget*. In plaats van een widget uit te breiden(extend) met een StatelessWidget, wordt een ConsumerWidget gebruikt omdat die een extra Ref object heeft die kan gebruikt worden om naar de providers te luisteren. Voor StatefulWidgets wordt er dan gebruik gemaakt van een *ConsumerStatefulWidget* en een *ConsumerState*.

```
    final CarProvider = Provider<String>((_) => 'This is a car');
    class CarWidget extends ConsumerWidget {
        @override
        Widget build(BuildContext context, WidgetRef ref) {
          final car = ref.watch(CarProvider);
          return Text(car);
        }
      }
```

Een tweede manier is om een widget te wrappen in een *Consumer*-widget zoals

[5] https://riverpod.dev/docs/introduction/why_riverpod

bij de Provider library. Zoals bij de eerste manier is er bij deze ook een Ref-object nodig. In dit geval is de Ref object een argument in de builder.

```
    final CarProvider = Provider<String>((_) => 'This is a car');

class CarWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {

    return Consumer(
      builder: (_, WidgetRef ref, __) {
        return Text(ref.watch(CarProvider));
      },
    );
  }
}
```

Er zijn zoals bij de Provider library ook veel soorten providers bij RiverPod:

- Provider wordt gebruikt voor objecten die niet veranderen of niet mutable zijn.
- StateProvider wordt gebruikt voor simpele state objecten die kunnen veranderen. StateProvider wordt in RiverPod 2.0 niet meer gebruikt en wordt vervangen door Notifier.
- FutureProvider is een provider voor states die mogelijk niet direct beschikbaar is, maar wel op een bepaald moment in de toekomst. Dit is handig voor asynchroon operaties, zoals het ophalen van data van een externe API.
- StreamProvider is handig wanneer er geluisterd moet worden naar een stream van events, zoals de wijzigingen van een databank.

Voor- en nadelen

Er zijn veel voordelen aan het gebruik van Riverpod. Er is meer controle over het herbouwen van widgets. Bovendien is het gemakkelijk te gebruiken met weinig boilerplate code. Het enige nadeel van Riverpod is dat je een externe library moet installeren.

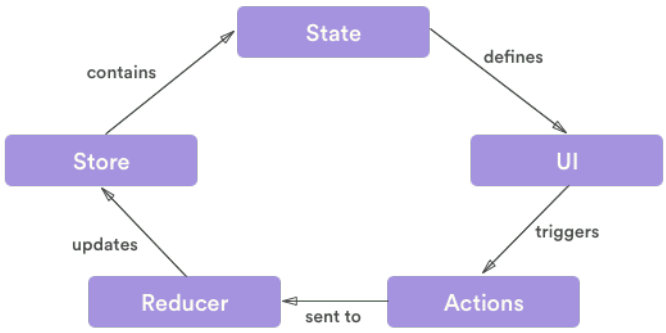
2.3.5. Redux

Redux is een library gekend door Javascript developers voor het ontwikkelen van webapplicaties. Echter is het ook mogelijk om Redux te gebruiken met Flutter. Redux is een architectuur met een unidirectionele dataflow. De states worden verzameld in een centraal geplaatste store. De gegevens in deze centrale opslag zijn

toegankelijk voor elk widget dat de gegevens nodig heeft, zonder dat er door de afstammelingen van de widgets in de boomstructuur hoeft te gaan. (Jolayemi, 2021)
Redux heeft 3 fundamentele principes:

- **Single source of truth**: Er is maar één store van informatie in de hele applicatie
- **Immutability**: De state van de applicatie is onveranderlijk en mag enkel gelezen worden. Dit betekent dus dat indien de waarde van een state wijzigt, dat de state vervangen moet worden door een nieuwe state.
- **Functions should be the state changers**: Veranderingen aan states mogen enkel gebeuren door een functie. Deze functies, genaamd Reducers, zijn de enige entiteit die veranderingen mogen brengen aan de states.

Voordat er meer over Redux beschreven kan worden, moet er eerst een paar kernbegrippen van Redux uitgelegd worden:
Een **Store** is een centrale locatie binnen de applicatie die alle states bijhoudt van de applicatie. Een **Reducer** is een synchrone functie die de store updatet met een nieuwe state. De reducer is ook de enige die de state mag veranderen. Een reducer neemt als argument de vorige state en een Action, en geeft een nieuwe state terug. Een **Action** beschrijft wat er is gebeurt. Een Action is de object die bepaalt wat er met de state moet gebeuren. Met behulp van de dispatch methode wordt een Action meegegeven aan de Store, zodat de state aangepast kan worden volgens de meegegeven Action. Een **Middleware** is een speciale functie die uitgevoerd wordt voordat de dispatched Action de Store bereikt. Ze worden meestal gebruikt om te luisteren naar Actions en asynchrone calls uit te voeren, zoals request naar een externe API. Eens dat er een response is, kan een andere Action dispatched worden naar de Reducer.
Elk state wijziging in Redux gaat door dezelfde cyclus zoals geïllustreerd in figuure



Figuur (2.4)
Redux lifecycle (Sharma, 2018)

2.4. De UI wordt gedefinieerd door de states. De drie primaire componenten van

Redux zijn Store, Reducers en Actions. De User interface triggert de Actions, die naar de Reducers worden gestuurd. De Reducers werken vervolgens Stores bij die de states bevatten.
Met behulp van *StoreProvider*-widget worden alle widgets binnen de widgetboom geinjecteerd met de Redux Store. De StoreBuilder-widget geeft de store door aan zijn kind en luistert continue naar de Store. Indien de Store geüpdatet wordt, zal dit het herbouwen van zijn afstammelingen triggeren. Een alternatief is de *StoreConnector*-widget. De StoreConnector zet de Store om in een ViewModel. Dit is veel performanter omdat het naar de ViewModel luistert. De afstammelingen worden enkel herbouwt als de ViewModel wijzigt.

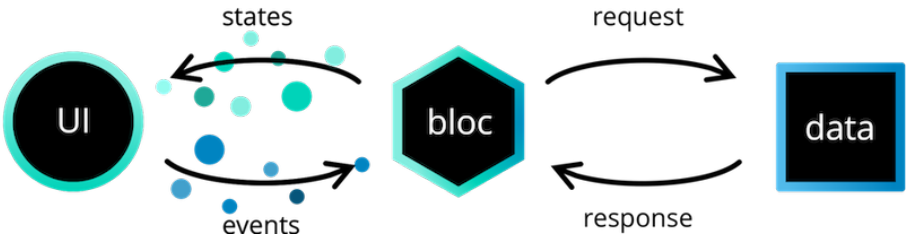
```
class MyApp extends StatelessWidget {
final store = Store(reducer,
    initialState: AppState.initialState(),
    );

@override
Widget build(BuildContext context) {
  return StoreProvider(
    store: store,
    child: MaterialApp(
      title: 'Flutter Redux Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: StoreConnector<int, Model>(
        distinct: true
        converter: (store)⇒ store.state.toString(),
        builder: ( context,  model) {
            return Text(model.value)
   },
    ),
  ));
  }
}
```

Voor- en nadelen
Er zijn veel voordelen aan het gebruik van Redux. Redux is ontworpen om bugs te voorkomen door State onveranderlijk te maken en met de unidirectionele dataflow. Ook worden de vorige 5 states bijgehouden, dus indien er iets misloopt wordt het debuggen gemakkelijker.

De nadelen van Redux is dat er heel veel boilerplate code is. Hoewel het goed is voor synchroon operaties, wordt het wat gecompliceerd bij asynchroon operaties.

2.3.6. Bloc

BLoC staat voor Business Logic Component. Het scheidt de presentatielaag van de business logica, die de code sneller, gemakkelijker om te testen en herbruikbaar maakt. Door gebruik te maken van BLoC wordt de applicatie verdeeld in 3 lagen (:)



Figuur (2.5)
BLoC Architectuur (Bloc, 2023)

- **Presentatielaag (user interface)**
  De verantwoordelijkheid van de presentatielaag is om uit te zoeken hoe hij zichzelf moet renderen op basis van een of meer BLoC-states.
- **Business logica (BLoC)**
  De businesslaag is verantwoordelijk voor het reageren op input van de presentatielaag met nieuwe states. Deze laag kan afhankelijk zijn van een of meer repositories om gegevens op te halen die nodig zijn om de states van de applicatie op te bouwen. De businesslaag wordt op de hoogte gebracht van gebeurtenissen/acties van de presentatielaag en communiceert vervolgens met de repository om een nieuwe state op te bouwen die de presentatielaag kan gebruiken.
- **Datalaag**
  Dit bestaat uit een repository en een data provider. De data provider is een klasse met methoden om CRUD operaties uit te voeren via een (externe) API. De repository is een wrapper rond één of meerder data providers en communiceert met de BLoC laag.

Voor- en nadelen Volgens Slepnev (2020) zijn er veel voordelen voor het gebruik van BLoC:

- Het biedt een goeie schaalbaarheid en testbaarheid
- Er is een goed seperation of concerns omdat de business logica in een aparte klasse zit

- Het is performant voor groot applicaties

De nadelen zijn:

- Streams moet in beide richtingen gebruikt worden. Dit leidt tot meer boiler-plate code
- Het is enkel efficient bij groot applicaties.

3

Methodologie

Etiam pede massa, dapibus vitae, rhoncus in, placerat posuere, odio. Vestibulum luctus commodo lacus. Morbi lacus dui, tempor sed, euismod eget, condimentum at, tortor. Phasellus aliquet odio ac lacus tempor faucibus. Praesent sed sem. Praesent iaculis. Cras rhoncus tellus sed justo ullamcorper sagittis. Donec quis orci. Sed ut tortor quis tellus euismod tincidunt. Suspendisse congue nisl eu elit. Aliquam tortor diam, tempus id, tristique eget, sodales vel, nulla. Praesent tellus mi, condimentum sed, viverra at, consectetuer quis, lectus. In auctor vehicula orci. Sed pede sapien, euismod in, suscipit in, pharetra placerat, metus. Vivamus commodo dui non odio. Donec et felis.

Etiam suscipit aliquam arcu. Aliquam sit amet est ac purus bibendum congue. Sed in eros. Morbi non orci. Pellentesque mattis lacinia elit. Fusce molestie velit in ligula. Nullam et orci vitae nibh vulputate auctor. Aliquam eget purus. Nulla auctor wisi sed ipsum. Morbi porttitor tellus ac enim. Fusce ornare. Proin ipsum enim, tincidunt in, ornare venenatis, molestie a, augue. Donec vel pede in lacus sagittis porta. Sed hendrerit ipsum quis nisl. Suspendisse quis massa ac nibh pretium cursus. Sed sodales. Nam eu neque quis pede dignissim ornare. Maecenas eu purus ac urna tincidunt congue.

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.

Maecenas non massa. Vestibulum pharetra nulla at lorem. Duis quis quam id lacus dapibus interdum. Nulla lorem. Donec ut ante quis dolor bibendum condimentum. Etiam egestas tortor vitae lacus. Praesent cursus. Mauris bibendum pede at elit. Morbi et felis a lectus interdum facilisis. Sed suscipit gravida turpis. Nulla at

lectus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Praesent nonummy luctus nibh. Proin turpis nunc, congue eu, egestas ut, fringilla at, tellus. In hac habitasse platea dictumst.

Vivamus eu tellus sed tellus consequat suscipit. Nam orci orci, malesuada id, gravida nec, ultricies vitae, erat. Donec risus turpis, luctus sit amet, interdum quis, porta sed, ipsum. Suspendisse condimentum, tortor at egestas posuere, neque metus tempor orci, et tincidunt urna nunc a purus. Sed facilisis blandit tellus. Nunc risus sem, suscipit nec, eleifend quis, cursus quis, libero. Curabitur et dolor. Sed vitae sem. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Maecenas ante. Duis ullamcorper enim. Donec tristique enim eu leo. Nullam molestie elit eu dolor. Nullam bibendum, turpis vitae tristique gravida, quam sapien tempor lectus, quis pretium tellus purus ac quam. Nulla facilisi.

4

Conclusie

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit lacus ut lorem. Sed luctus justo sed enim.

Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetuer libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem. Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

A

Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1. Introductie

Smartphones zijn een integraal onderdeel van ons dagelijks leven geworden. Daarom worden er steeds meer vraag om mobiele applicaties te ontwikkelen. Ook wordt cross-platform ontwikkelen steeds aantrekkelijker voor ontwikkelaars omdat ze in een kortere tijd applicaties kunnen ontwikkelen voor meerdere platformen.
The Mobility Factory (TMF) is een coöperatie die IT-oplossingen biedt aan bedrijven die elektrische autodelen diensten aanbieden. De applicaties worden door de TMF-ontwikkelaars in Flutter geïmplementeerd. Omdat de applicaties van TMF steeds groter en complexer worden, werkt de applicatie steeds minder performant. Daarom kwam de vraag om de performantie van hun applicatie te optimaliseren.
In Flutter zijn er tal van mogelijkheden voor het beheren van states. Afhankelijk van de complexiteit van de applicatie moet er overwogen worden welke state management benadering het best bij de applicatie past en of het de meest performante optie is. In dit onderzoek wordt eerst de verschillende soorten state management benaderingen grondig besproken en daarna worden ze vergeleken met elkaar op basis van prestatie en code complexiteit aan de hand van een proof-of-concept. Hieruit wordt de onderzoeksvraag dus gecreëerd: *Op welk manier kan states bijgehouden worden in Flutter en welk past het bij bij de applicatie van The Mobility Factory?*. De hoofdonderzoeksvraag kan verder verdeeld worden in de volgende deelonderzoeksvragen:

- Welk benadering van state management is het performantste qua cpu-gebruik, opstartsnelheid, geheugengebruik...?

- Hoeveel moeite zal het The Mobility Factory kosten om de verschillende benaderingen van state management te implementeren? Hoe complex is het om te integreren in de bestaande code?

A.2. State-of-the-art

Flutter is een open source raamwerk van Google voor het bouwen van multi-platform applicaties vanuit één codebase, of met andere woorden een cross-platform raamwerk. Flutter werd in 2018 uitgebracht en wordt ondertussen door meer dan twee miljoen ontwikkelaars gebruikt om meer dan 350 duizend Flutter applicaties te ontwikkelen. [1]

A.2.1. Application performance, een onderdeel van User Experience

Applicatieperformantie is de werkelijke performantie die wordt ervaren door de eindgebruikers van de applicatie, zoals de gemiddelde responstijd bij normale belasting of piekbelasting (Rouse, 2014).
Door een onderzoek van Google wordt er vastgesteld dat naarmate de laadsnelheid oploopt, hoe meer geneigd de gebruikers zijn om de website te verlaten (An, 2010). Bij een laadsnelheid van meer dan 3 seconden van een webpagina, vertrekt 53% van de gebruikers bij het inladen van de webpagina.
Volgens Nielsen (2010), een UX-expert, is responsiviteit van een applicatie een basisregel voor het ontwerp van gebruikersinterfaces. De gebruikers moeten snel door de applicatie kunnen navigeren. Volgens Nielsen is een laadsnelheid tussen 1 en 10 seconden aanvaardbaar. Een laadsnelheid van langer dan 10 seconden kan ervoor zorgen dat de gebruiker afgeleid wordt waardoor het moeilijker is om geavanceerde taken tot een goed einde te brengen tegen dat de applicatie reageert.

A.2.2. Technieken om applicatie performantie te verbeteren

Flutter werd ontwikkeld om performant te zijn. Maar de prestatie van een Flutter applicatie kan sterk beïnvloed worden door hoe de applicatie ontworpen en geïmplementeerd wordt. [2]
Dit zijn 3 voorbeelden om de prestatie van een Flutter applicatie te verbeteren:

1. **Memory management**
   Het memory management in Flutter wordt afgehandeld door de Dart Virtual Machine (DVM), die een garbage collector heeft die automatisch ongebruikte geheugen terughaalt. Er kunnen echter nog steeds geheugenlekken optreden als objecten niet op de juiste manier uit het geheugen worden vrijgegeven, waardoor ze in de heap blijven hangen en onnodige bronnen verbruiken.

[1] https://flutter.dev/
[2] https://docs.flutter.dev/perf/best-practices

Om geheugenlekken te vermijden moet de Dispose-methode gebruikt worden om onnodige objecten te verwijderen. (Sabbagh, 2023)

2. **Netwerk optimalisatie**
   Om netwerk prestatie te verhogen moet er gebruik gemaakt worden van efficiënte netwerk libraries zoals Dio. Ook door de gegevens te cachen op het apparaat van de gebruiker kan er onnodige netwerkverzoeken vermeden worden en wordt de latentie verminderd. [3]

3. **State management**
   Binnen dit onderzoek wordt er gefocust op dit onderdeel. Dit wordt verder beschreven in het volgende puntje 2.4

A.2.3. State management binnen Flutter

Flutter beschrijft een state als informatie die enerzijds synchroon kan worden gelezen wanneer de widget wordt gebouwd en anderzijds kan veranderen tijdens de levensduur van de widget. [4]
States worden niet van parent widget naar child widget doorgegeven. De state van een widget wordt beheerd door zichzelf of door een parent widget en als de state van de parent widget wijzigt, moet alle widgets onder die parent widget ook herladen worden. Daarom kan het apart beheren van states zorgen voor efficiëntere updates en betere prestaties van de applicatie, aangezien enkel betreffende widgets heropgebouwd moeten worden. Deze functie is vooral gunstig voor complexe applicaties, waar het herbouwen van de volledige widgetstructuur tijdrovend kan zijn en kan leiden tot ondermaatse prestaties. Het is dus van groot belang dat de juiste benadering van state management wordt toegepast, zodat het optimaal presteert en een naadloze gebruikerservaring garandeert. (siddhardha, 2023)
Dit zijn de mogelijke benaderingen van state management in Flutter:

- SetState
- Provider
- GetX
- RiverPod
- Redux
- BLoC / Rx
- MobX

[3] https://www.linkedin.com/pulse/flutter-performance-optimization-techniques-best-practices/
[4] https://api.flutter.dev/flutter/widgets/State-class.html

A.2.4. Relevante studies

State management wordt door het Flutter-team als complex bestempeld en er wordt veel gediscussieerd onder de Flutter-ontwikkelaars over wat de beste aanpak is.
Een vergelijkbaar onderzoek dat reeds werd gedaan over state management is door De Vriest (2019). Hoewel zijn onderzoek maar 3 jaar oud is, is Flutter heel sterk geëvolueerd sindsdien. Zijn onderzoek concludeerde dat Provider het meest performant is voor een simpel CRUD-applicatie. Maar in zijn onderzoek waren er tekortkomingen, zoals het enkel testen op een Android-toestel en ook werden data niet van een API opgehaald, wat niet realistisch is voor de realiteit.
Een andere vergelijkbaar onderzoek werd door Slepnev (2020) geschreven. In zijn onderzoek werd de verschillende benaderingen van state management gegroepeerd en vergeleken. In zijn onderzoek werden er criteria opgesteld om het best passende benadering te kiezen voor de verschillende use cases.

A.3. Methodologie

Eerst zal een vergelijkende studie uitgevoerd worden door de verschillende benaderingen van state management op te sommen en hun voor- en nadelen te vergelijken. Uit de gevonden state management benaderingen zullen er vier uitgekozen worden om toe te passen in het proof-of-concept.
In het methodologiegedeelte zullen verschillende versies van een bepaalde feature van de applicatie van TMF geïmplementeerd worden, waarbij voor elke versie een andere benadering van state management wordt toegepast. De applicatie van TMF die nagemaakt zal worden is een beheerapplicatie om reservaties en gebruikers te beheren.
De prestaties van de applicatie zullen getest worden op basis van CPU-gebruik, laadsnelheid en nodige opslag voor de applicatie. Er wordt ook een afweging gemaakt tussen de prestaties die worden geleverd door state management en de complexiteit van de integratie ervan in de bestaande code.
Het proof-of-concept zal geïmplementeerd worden met Flutter op Visual Studio Code. De applicatie van TMF wordt voornamelijk gebruikt op de web, daarom zal de performantie van de proof-of-concept getest worden als een webapplicatie en ook op een fysieke iOS-toestel. De code complexiteit van de state managements zullen vergeleken worden op basis van aantal nodige lijnen code. Om de performantie te meten zal er automated integration testen geschreven en uitgevoerd worden aan de hand van Flutter Driver. Bij elk uitvoering van de testen zullen de performantie gemeten worden. [5]

[5] https://docs.flutter.dev/cookbook/testing/integration/profiling

A.4. Verwachte resultaten

Uit eerdere onderzoeken blijkt dat Provider het best benadering van state management is op vlak van complexiteit en ook op prestatie vlak. Dit is ook de state management benadering die aanbevolen wordt door Flutter. Maar omdat de applicatie van TMF complexer is, wordt er verwacht dat Bloc een betere benadering van state management zal zijn.

A.5. Discussie, conclusie

Er zijn talloze manieren om de prestaties van een Flutter-applicatie te verhogen. Een van de manieren is om het meest geschikte state management toe te passen, afhankelijk van de complexiteit van de geïmplementeerde applicatie.
Het doel van deze studie is om TMF-ontwikkelaars een idee te geven van welke state management opties beschikbaar zijn in Flutter en wat het meest performant is voor hun applicatie.

Bibliografie

Adnan, M. (2023). Flutter State Management: What to Choose- Provider, BLoC, or Redux? Verkregen oktober 23, 2023, van https://www.linkedin.com/pulse/learn-how-flutter-render-widgets-muhammad-adnan/

An, D. (2010). Find out how you stack up to new industry benchmarks for mobile page speed. Verkregen juli 21, 2023, van https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

Arshad, W. (2021). *Managing State in Flutter Pragmatically*. Packt Publishing.

Bloc. (2023). Architecture. Verkregen november 25, 2023, van https://bloclibrary.dev/#/architecture

De Vriest, J. (2019). *Een vergelijkende studie tussen verschillende benaderingen van State Management in Flutter* [Bachelor's Thesis]. Hogeschool Gent. Verkregen augustus 5, 2023, van https://scriptie.hogent.be/2019-2020/339_201640029_PBA-TIN_scriptie.pdf

Flutter. (2023). Flutter architectural overview. Verkregen november 20, 2023, van https://docs.flutter.dev/resources/architectural-overview

Goel, T. (2023). Lifecycle of Flutter App. Verkregen november 23, 2023, van https://mobikul.com/lifecycle-of-a-flutter-app/

Jolayemi, D. (2021). Flutter Redux: Complete tutorial with examples. Verkregen november 23, 2023, van https://blog.logrocket.com/flutter-redux-complete-tutorial-with-examples/

Makadiya, K. (2022). Flutter State Management: What to Choose- Provider, BLoC, or Redux? Verkregen oktober 23, 2023, van https://medium.com/dhiwise/flutter-state-management-what-to-choose-provider-bloc-or-redux-214160adbae4

Nielsen, J. (2010). Website Response Times. Verkregen juli 21, 2023, van https://www.nngroup.com/articles/website-response-times/

Rouse, M. (2014). Application performance. Verkregen juli 21, 2023, van https://www.techopedia.com/definition/30457/application-performance

Sabbagh, A. (2023). Flutter Performance Optimization Techniques Best Practices. Verkregen augustus 16, 2023, van https://blog.devgenius.io/identifying-and-handling-memory-leaks-in-flutter-apps-1ad5e7d499e7

Sharma, R. (2018). ABC of Redux. Verkregen november 23, 2023, van https://dev.to/radiumsharma06/abc-of-redux-5461

siddhardha. (2023). My Experience with Flutter State Management. Everything You Need to Know About it. Verkregen augustus 16, 2023, van https://levelup.gitconnected.com/everything-you-need-to-know-about-flutter-state-management-from-my-experience-19f56729f3c4#:~:text=State%2C%20in%20Flutter%2C%20refers%20to,itself%20or%20its%20parent%20widget.

Slepnev, D. (2020). *State Management approaches in Flutter* [Bachelor's Thesis]. South-Eastern Finland University of Applied Sciences. Verkregen augustus 15, 2023, van https://www.theseus.fi/bitstream/handle/10024/355086/Dmitrii_Slepnev.pdf

Windmill, E. (2020). *Flutter in action*. Mannning Publications.