УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за електротехнику и рачунарство
Одсјек за рачунарску технику и рачунарске комуникације

ЗАВРШНИ (BACHELOR) РАД

Кандидат: **Милош Бујаковић**
Број индекса: **SW-58/2015**

Тема рада: **ЈЕДНО РЈЕШЕЊЕ МОБИЛНЕ АПЛИКАЦИЈЕ ЗА ПРИКАЗ ЕЛЕТРОНСКОГ ПРОГРАМСКОГ ВОДИЧА**

Ментор рада: **др Илија Башичевић**

Нови Сад, август, 2024

УНИВЕРЗИТЕТ У НОВОМ САДУ ●**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	Монографска документација
Тип Записа, **ТЗ**:	Текстуални штампани материјал
Врста рада, **ВР**:	Завршни (Bachelor) рад
Аутор, **АУ**:	**Милош Бујаковић**
Ментор, **МН**:	**Др Илија Башичевић**
Наслов рада, **НР**:	**Једно рјешење мобилне апликације за приказ електроснког програмског водича**
Језик публикације, **ЈП**:	Српски / ćirilica
Језик извода, **ЈИ**:	Српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	**2024**
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	
Научна област, **НО**:	Електротехника и рачунарство
Научна дисциплина, **НД**:	Рачунарска техника
Предметна одредница/Кључне речи, **ПО**:	
УДК	
Чува се, **ЧУ**:	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	**У раду је описан један од начина израде мобилне апликације за приказ и коришћење електронског програмског водича (EPG) на Android уређајима (првенствено таблетима). Главна сврха израде је истраживање начина рада Jetpack Compose-a, новог окружења за развој GUI-a у Android-у. Поред Jetpack Compose-a, бавићемо се и добављањем података са послужиоца на интернету, слањем HTTP захтјева користећи Android-ово RETROFIT окружење за приступ послужиоцима. Дио добављених података ћемо чувати у засебној Бази Података, како бисмо могли сачувати трајно измјене начињене од корисника за што ћемо користити ROOM окружење. Све поменуто ћемо уклопити заједно помоћу MVVM обрасца грађе и DaggerHilt окружења за управљање радом Android апликације.**
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:	Предсједник:	др Иван Каштелан	
	Члан:	др Небојша Пјевалица	Потпис ментора
	Члан, ментор:	др Илија Башичевић	

UNIVERSITY OF NOVI SAD●**FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	Bachelor Thesis
Author, **AU**:	**Miloš Bujaković**
Mentor, **MN**:	**Ilija Bašičević, phd**
Title, **TI**:	**A solution of a mobile app for managing and displaying EPG for LiveTV users**
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2024
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	
Scientific field, **SF**:	Electrical & Software Engineering
Scientific discipline, **SD**:	Computer Engineering, Engineering of Computer Based Systems
Subject/Key words, **S/KW**:	
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:	
Abstract, **AB**:	**The paper describes one way of creating a mobile application for displaying and using the electronic program guide (EPG) on Android devices (primarily tablets). The main purpose of making it is to explore how the Jetpack Compose, a new Android library for GUI development, works. In addition to Jetpack Compose, we will explore ways of acquiring data from servers on the Internet, by sending HTTP requests using the Android's RETROFIT library for communicating with online servers. We will store some of the acquired data in a local Database, so we can permanently save the changes made by the local user, for which we will use the ROOM environment. All of the above will be fit to work together using the MVVM project design pattern and Android's DaggerHilt Dependency Injection library to manage it.**
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	

Defended Board, **DB**:	President:	Ivan Kaštelan, phd	
	Member:	Nebojša Pjevalica, phd	Menthor's sign
	Member, Mentor:	Ilija Bašičević, phd	

Захвалност

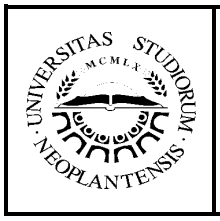
Посебно се захваљујем Борису Тиркајли, Милану Новаковићу, Дејану Нађу, као и ментору проф. др Илији Башичевићу на стручној помоћи, савјетима и утрошеном времену како бих успио завршити рад.

Захваљујем се руководству предузећа и института РТ-РК на указаној прилици да се боље упознам са начином рада у инжењерском окружењу и будем укључен у ток развоја нових програмских рјешења.

Нарочиту и изразиту захвалност указујем проф. Миодрагу Ђукићу, без чијег разумијевања, помоћи и залагања би не само израда овог рада много дуже трајала, него и уопштено ток студија.

Захваљујем се својој родбини и пријатељима на исказаном стрпљењу и повјерењу.

На крају захваљујем се и свима осталима који су на било који начин допринијели изради овог завршног рада и мом образовању.



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

САДРЖАЈ

Списак слика

Скраћенице

ADID - ***A**ndroid a**D**vertising **ID**entifier (*раније: ***A**ndroid **D**evice **ID**entifier*)
јединствени кључ андроид уређаја

AUID - ***A**noki **U**ser **ID**entification – Anoki*-ијев кључ за распознавање корисника

API - ***A**pplication **P**rogramming **I**ntefrace* - спрега за коришћење апликације од стране других програмера

CRUD - ***C**reate **R**ead **U**pdate **D**elete* – основни начини рада са подацима, прављење, читање, измјене и брисање

DAO - ***D**ata **A**ccess **O**bject* – приступни објекат за податке у *ROOM* Базама Података (БП)

DTO - ***D**ata **T**ransfer **O**bject* –класе и објекти намијењени за пренос података са једне на другу платформу (нпр. са интернета у базу, из базе у апликацију...)

DTV - **D**igital **T**ele**V**ision - Дигитална телевизија**VHS** - **V**ideo **H**ome **S**ystem - средство за снимање емитованог садржаја код куће

DSP - ***D**igital **S**ignal **P**rocessor* – чипови посебне намјене обраде слике/звука

DP - ***D**ensity-Independent **P**ixels* – мјера за величину компоненти сразмјерно прилагодљива величини екрана

SP - ***S**cale-Independent **P**ixels*– мјера за величину слова сразмјерно прилагодљива величини екрана

EPG - ***E**lectronic **P**rogram **G**uide* - електронски програмски водич

GUI - ***G**raphhic **U**ser **I**nterface* - сликовно прилагођени приказ апликације за корисника

JSON - ***J**ava**S**cript **O**bject **N**otation* – најчешће коришћени облик преноса података преко интернета, настао као начин записивања објеката у програмском језику ***J**ava**S**cript* а касније се проширио на друге области

MVVM - ***M**odel **V**iew **V**iew**M**odel* - образац грађе софтвера заснован на 3 цјелине: приказу (*View*), подацима (*Model*) , и спони између њих са програмском логиком *(ViewModel)*

UX - ***U**ser e**X**perience* - дојам корисника током кориштења апликације

URL - ***U**niform **R**esource **L**ocator* **–** низ бројева или знакова који јединствено одрјеђују мјесто на интернету гдје се налази оно што тражимо

UI - ***U**ser **I**nterface* - (спољни) изглед апликације ка кориснику

1. Увод

Ново доба увијек доноси са собом нова очекивања, нове изазове и нове потребе. Тако је средином 1970их дошло до знатног напретка у развоју рачунара, смањења у величини као и знатног раста њихових могућности. Због тога, јавља се све већа потреба и све већа зависност од рачунара због њихове могућности обраде гомиле података невјероватном брзином како би се подигла учинковитост и повећала зарада. Због тога, све већа и већа улагања у развој рачунара врло брзо постају неминовност и они не само да помажу људима при обради података и у индустрији, него врло брзо улазе и у потрошачке производе и почињу се користити свакодневно. Од најприје личног рачунара врло брзо и бројни други кућни уређаји постају "мали рачунари". Усљед напретка технологије чипова, обични механички стројеви у кући, готово преко ноћи постају "паметни уређаји" са не само могућностима обављања првобитне намјене (нпр. загријевање хране на шпорету приликом кувања) него и почињу пружати разноразне додатне могућности (као нпр. Подешавање дужине припреме хране на одријеђеним температурама на новим индукционим плочама и сл.).

Слично се дешава и са телевизорима и уопштено ***TV*** индустријом. Читав пренос сигнала који се одвијао испрва искључиво преко сигналних станица на земљи и антена за примање аналогног сигнала (тзв. „Земаљска Телевизија"), те помоћу сателита послатих у свемир (Сателитска ***TV***) итд. усљед развоја технологије почиње се преносити и кабловски, али и преко интернета. Ово почиње због потребе за приказ садржаја са јасним сликама и звука без шумова (тј. дијелова екрана који су сиви, пуни тачкица итд. као и чистог звука који је јасно разумљив) и назива се процесом „дигитализације". Овај процес подразумијева записивање свих података који се емитују у ***TV*** станицама (тзв. предајницима) у облику низа цифара (*eng. digits*) умјесто простог слања електромагнетних таласа различитих фреквенција. Како је низ цифара увијек јасан, недвосмислен и непромјењив у односу на спољне утицаје (као нпр. временске непогоде, планине, шуме, зграде и остале ствари које су ометале пренос електромагнетних таласа) овај прелазак је ријешио проблеме шума и лоше слике, али због своје величине знатно успорио пренос. Међутим напретком технологија не само бржег и јачег преноса него и нових начина за збијање (*eng compression*) података, нова дигитална ***TV*** успијева отклонити своје недостатке и преко ноћи постаје општеприхваћен начин емитовања сигнала. Пошто ради са подацима у облику непромјењивог низа цифара јасна слика и чист звук постају гарантовани, а могућности преноса додатних података не представљају никакве додатне препрјеке нити оптерећење за гледање садржаја. Због тога, очекивања првенствено тржишта, али и корисника расту, те сви желе знати не само за оне емисије и програме који се редовно пуштају него и филмове, серије, спортске утакмице и остале програме који се пуштају ванредно, како би могли одвојити вријеме унапријед и погледати их. Тада ***TV*** станице почињу правити часописе са недјељним а касније и мјесечним списком програма на сваком каналу, али због

ограниченог броја часописа, велике потражње и споре испоруке до крајњег корисника врло брзо се почиње са слањем тих података најприје у текстуалном облику (тзв. Teletext). Због своје сложености, непрегледности, као и уопште лошег изгледа почиње убрзани развој софтвера за простији, прегледнији приказ списка прогама сваког ***TV*** канала. То постаје могуће због уградње све више и више рачунарских чипова у нове дигиталне ***TV*** пријемнике (најчешће тзв. „телевизоре“) који усљед тога постају прави мали рачунари (данас познатији као „паметни уређаји“), те омогућују не само прости приказ списка програма, него добављање додатних података о самим емисијама, творцима, учесницима, самостално снимање емисија за касније гледање, а развојем мобилних телефона заказивање и прављење напомена и подсјетника за гледање који прерастају и читаве водиче зване ***EPG*** *(Electronic Programme Guide*). Ти водичи, у данашње вријеме врло брзог и ужурбаног начина живота омогућавају корисницима брзо, лако и поуздано испратити оне садржаје који их највише занимају, стога постоји и непрестана потреба за њиховим додатним усавршавањем.

Из тих разлога, јавила се и потреба за израдом програма тј. Таблет апликације за приказ садржаја *Anoki* канала као и електронског програмског водича како би корисници таблет уређаја били у могућности увијек испратити жељене емисије, гдје год се налазили и без потребе за посједовање великих, тешко преносивих савремених ***TV*** пријемника (најчешће тзв. „телевизора“). У овом раду је описан начин на који је остварена апликација која тежи надомјестити ограниченост тренутно постојећег софтвера јер је исувише гломазан, свеобухватан и ради само на великим ***TV*** пријемницима који имају веће могућности од малих преносних уређаја као што су таблети, мобилни телефони и сл. Опис овог рјешења раздијељен је у неколико поглавља:

У овом поглављу је направљен кратак увод у апликацију коју желимо направити и зашто нам је она потребна.

У сљедећем поглављу су описане теоријске основе дигиталне ***TV***, *Android OS*-а (Оперативни систем), *Kotlin* програмског језика, и кратак опис нове *Jetpack Compose* библиотеке коришћене за ***UI*** (*User Interface*).

У трећем поглављу описаћемо нацрт рјешења наше апликације даље објашњавајући Jetpack Compose, начин рада са њим као и са *Anoki* послужиоцем, Базом Података, и услугама које пружа и појаснити сам начин израде раздијељен у 5 техничких корака подробније описаних у четвртом поглављу.

У петом поглављу налазиће се слике крајњег изгледа, као и опис коришћења наше апликације.

У шестом поглављу извешћемо кратак закључак шта је постигнуто овом апликацијом и зашто.

У седмом поглављу налази се списак коришћених извора приликом писања рада.

За остварење овог рјешења користићемо разна окружења у сврху убрзавања и олакшавања израде. То су прије свих, главна област коју истражујемо – *Jetpack Compose* окружење (*eng library*) за израду корисничке спреге на потпуно нов начин. Такође, користићемо и ***MVVM*** образац грађе апликације (*eng project design pattern*) са три засебна слоја за приказ, податке и програмску логику. Треће окружење које ћемо користити је ***DaggerHilt*** које ће бити задужено за управљање нашом апликацијом и омогућити нам већи ниво прилагодљивости наше апликације ка различитим ***API***-јима користећи свој уграђени образац убризгавања зависности (*eng dependency injection*). За добављање података од *Anoki* послужиоца користићемо *RETROFIT* окружење за размјену података са *FAST TV* послужиоцима. На самоме крају, за рад са ***БП*** користићемо Android-ово окружење *ROOM* које нуди најпростији начин приступа и рада са *SQLite* Базама Података.

2. Теоријске основе

У овом поглављу су подробније описане теоријске основе дигиталне ***TV*** као области које смо се дотакли раније, као и главног оруђа које користимо за израду апликације, *Android* OS-а (Оперативни систем), *Kotlin* програмског језика, и кратак опис новог *Jetpack Compose* окружења кориштеног за цртање спољног изгледа апликације ка кориснику тј. ***UI***-а (*User Interface*).

2.1 Дигитална Телевизија (DTV)

У доба свеопште дигитализације, Дигитална ***TV*** (***DTV***) дјелује као природан сљедећи корак у развоју телевизије и уопштено свих јавних гласила (*eng. media*). Поред јасне слике и чистог звука без шумова, она доноси и мноштво додатних могућности, као што су:

- звук изузетно високе каквоће, али и разноликости (могућност додавања ефеката)
- поуздан сигнал на који не утичу спољне околности као невријеме (осим у случају оштећења опреме)
- слика у високој резолуцији тзв. ***HD*** (High Definition) итд.

Ипак можда и једна од кључних особина која обиљежава Дигиталну ***TV*** и која је убрзала њен развој је могућност тражења одријеђеног садржаја или додатних података о садржају (или било чему везаном за њега) на захтјев корисника тј. гледалаца. Ово ће заувијек промијенити начин на који људи посматрају телевизију и све везано за њу, од јавних гласила, ***TV*** станица, серија, филмова и свих врста садржаја који су на њој пуштани. Ово ће омогућити не само гледање искључиво онога што желимо, него и онда када желимо то гледати, омогућиће нам снимање унутар самог ***TV*** пријемника (што је раније било могуће само помоћу чувених видео рекордера који су снимали садржај на *VHS* касете или *CD*-ове) познатије као бафероване. Омогућиће нам и заустављање гледања програма уколико морамо нешто хитно урадити, а затим наставити гледати га без пропуштања ичега, враћање уназад уколико нешто желимо поново погледати или убрзавање дијелова који су нам досадни или их не желимо гледати и било које друге врсте управљања ониме што желимо гледати по личном нахођењу. Главна препрјека наглом преласку на ***DTV*** је свакако било то што подаци о садржају који су се преносили у облику цифара, за разлику од електромагнетних таласа су исувише гломазни, заузимају доста простора и оптерећују пропусни опсег. Но, овај недостатак је врло брзо превазиђен на два начина:

1) побољшањем саме опреме тј. њене јачине и брзине преноса садржаја;
2) збијањем садржаја на само најнужније посебним унапријед договореним поступцима како би се садржај могао у потпуности поново саставити из њих (познатији као *compression algorithms*)

Овиме су суштински уклоњени сви недостаци, а задржане све предности ***DTV*** која добија потпуну надмоћ над Аналогном TV и постаје главни и готово једини начин не само пуштања садржаја ка корисницима, него и слања садржаја на захтјев корисницима.

Како би све ово било могуће, било је неопходно појачати јачину и брзину рада ***TV*** пријемника што је омогућено новим чиповима, не само опште, него и посебне намјене за обраду сигнала као што су ***DSP*** (*Digital Signal Processing*) чипови, те најзад направити Оперативни Систем који ће управљати свим тим процесима и

омогућити искоришћење свих могућности Дигиталне ***TV***. За постизање свега овога неопходно је направити спону између Хардверског дијела (***DSP*** и осталих чипова) са садржајем који желимо приказати.

Да бисмо то постигли потребно је разумјети како руковати стварима на ниском нивоу апстракције, као и омогућити добављање тих података на високом нивоу који је у стању приказати корисницима садржај на једноставан и занимљив начин, јер дојам корисника и њихова жеља за коришћењем производа је пресудна за његов даљи развој у потрошачкој индустрији и представља главну мјеру сврсисходности. Стога, налик на познати *OSI* model и његових 7 слојева апстракције у размјени података, и унутар ***DTV***-а имамо различите слојеве софтвера који су задужени за сваки корак размјене података од њиховог доспијећа до приказивања кориснику.

Слика 2.1 Слојеви софтвера DTV пријемника

Приликом размјене података са нивоа цифара тј. Бита се увијек постепено „пењемо“ на све већи ниво апстракције како бисмо најзад, тумачећи податке по већ прописаним правилима успјели преточити их у ***TV*** садржај који се може гледати. На сличан начин и када корисник жели гледати одријељени садржај, његов захтјев се са највишег нивоа увијек “спушта” на ниво цифара који се може послати хардверски (преко каблова) до неког послужиоца (eng server) који ће опет подизати ниво апстракције како би разумио шта тражимо итд. Итд. Можемо слободно рећи да изглед, начин рада и размјене података између ових слојева наликује стогу, гдје се ствари увијек додају на крај (или врх) док се приликом уклањања увијек најприје скидају ствари с краја (или врха). Оваква структура се у програмирању назива *LIFO* (*Last-In-First-Out*), а њен изглед стог (*eng. stack*), отуда и назив *Software Stack* за читаву структуру слојева софтвера у ***DTV***-у. На слици

јасно раздвајамо 3 главна слоја:

1) хардверски слој ***HAL*** (*Hardware Abstraction Layer*) који је задужен за обраду бита који стижу путем каблова, као и претварање података са ***PAL*** слоја у бите како би се могли прослиједити даље - тај дио је најчешће већ намјештен у оквиру самог OS-а који користимо на нашем уређају. Софтвер писан за овај слој је најчешће написан у некој врсти *Assembler* језика (у зависности од архтитектуре уређаја) и/или неком од „нижих“ програмских језика (нпр. *C*).

2) платформски слој ***PAL*** (*Platform Abstraction Layer*) који је задужен за добављање података о разноразним врстама услуга (*eng services*) и прослеђивање кориснику, од самог A/V садржаја преко превода, програмске шеме па све до додатних података о ономе шта гледамо (нпр. о редитељу филма, глумцима, итд. итд.). Софтвер писан за овај слој је најчешће написан у неком од „нижих програмских језика“. ***PAL*** се најчешће односи на неки оперативни систем који користимо (у овом случају Android) те има за задатак обезбиједити све оно што је потребно програмерима за прављење нових апликација.

3) апликативни слој ***AL*** (*Application Layer*) је слој који је задужен за прихват података са средњег слоја и њихов приакз кориснику на занимљив и једноставан начин за коришћење како би се што више побољшао кориснички дојам коришћења (*User Experience*) и самим тиме наш производ био што боље прихваћен на тржишту. Софтвер овог слоја је писан у програмским језицима „вишег нивоа“ како би се посветило више пажње самом добављању података и уређивању изгледа њиховог приказа кориснику.

Слој који се налази негдје између ***AL*** и ***PAL*** (али суштински се убраја у ***AL***) омогућава програмерима додатни ниво поједностављења оних саставних дијелова апликације који су превише сложени (као нпр. начин рада *Demultiplexer*-а који је задужен за раздвајање садржаја и метаподатака те обраду тих метаподатака). Постојање овог међуслоја омогућава програмерима апликација усредсриједити се искључиво на захтјеве корисника и занемарити сложени поступак рада са нпр. *Demultiplexer*-ом. Пошто се налази у средини овај слој се такође зове и „средњим слојем“ (eng *Middleware*) ***DTV*** *Software Stack*-а. Софтвер у овом слоју је најчешће писан од стране произвођача уређаја тј. ***DTV*** пријемника који користимо.

У овом случају то је предузеће *Anoki*, а како бисмо добавили податке који су нам потребни за прављење апликације неопходно је проучити упутства за коришћење која нам они достављају уз своје уређаје (познатији као *documentation*).

Највећа препрјека у развоју оваквог софтвера је управо та што због широке распрострањености, неопходно је направити нешто што неће у многоме зависити од разноврсне архитектуре која се може затећи на уређајима прављеним од различитих произвођача, јер на тај начин можемо најбрже направити софтвер који ће радити на свим уређајима и покрити највећи могући дио тржишта.

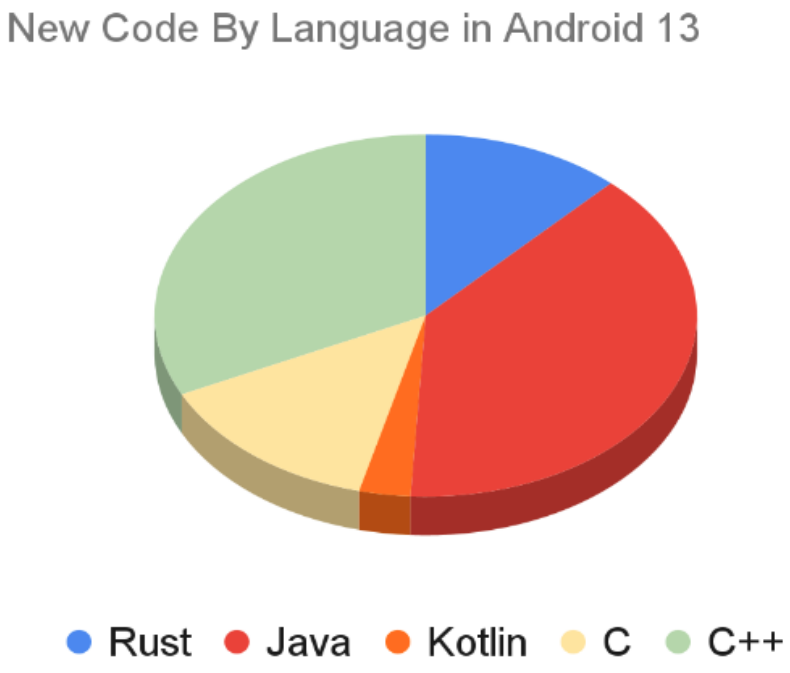
Као рјешење ове препрјеке у софтверу ***DTV***-а наметнуо се *Android* оперативни систем, који се већ показао као одлично рјешење на другим “паметним уређајима”, првенствено мобилним телефонима, због своје прилагодљивости свакој врсти хардверске архитектуре на разноразним врстама уређаја.

2.2 Android оперативни систем

Android OS своју прилагодљивост вуче из тога што је заснован на језгру Линукса. *Android* у почетку бијаше подухват са јавним приступом коду (*Open source*) истоименог предузећа 2003. године. Иако замишљен као софтвер за нове камере, врло брзо прераста у *OS* за преносиве уређаје,

са нагласком на моблинми телефонима који у почетку имаху цјелокупну QWERTY тастатуру. Касније, након куповине од стране *Google*-а 2005. године, она постаје дио самог софтвера као и већина познатих апликација затвореног кода које сада постају нераздвојиви дио сваког *Android OS*-а на уређају као нпр. *Gmail, YouTube, Google Maps, Google Chrome*, итд. Као *OS* који је прављен за мале преносиве уређаје врло ограничене количине меморије и процесорске моћи, те и прилагодљив различитим архитектурама хардвера, *Android* врло брзо (већ 2008.) постаје најраспрострањенији *OS* на мобилним уређајима од нових „паметних мобилних телефона“ преко таблета, све до паметних сатова, наочара итд. Већ 2012. *Android* постаје најраспрострањенији *OS* за мобилне уређаје, а од 2020. претпоставља се да преко 75% свих мобилних уређаја користи *Android OS* а од осталих 25% најпознатији је *iOS* од *Apple*-а.

Према најновијим процјенама преко 3 милијарде (тј. 3.000.000.000) уређаја данас користи *Android* (око 70% укупног броја уређаја), не само већ споменути „мали рачунари“ него и кућни уређаји попут фрижидера, микроталасне пећнице, справе за вјежбање, банкомати и уопште било која справа са екраном и неком врстом ***GUI***-а. Главни разлог тога је што је и након куповине од стране *Google*-а *Android* остао софтвер отвореног кода, доступан свима на увид, али не само то, него и допуњавање, измјене и прилагођавања различитим уређајима, потпуно бесплатно! Управо та, огромна распрострањеност, пријемчивост (због бесплатног софтвера) и прихваћеност на тржишту, довела је до великог броја програмера и уопште познавалаца *Android*-а те је приликом преласка на нове „паметне телевизоре“ баш *Android* постаје први избор за *OS*, гдје се показао изврстан. Као и *Linux* језгро и његово језгро је највише писано у *C* и *JAVA*-и, али је и надограђено у *Rust* и *Kotlin*-у.



Слика 2.2 Удио различитих програмских језика у језгру *Android OS*-а

Поред тога што је састављен из *JAVA*-е (и *Kotlin*-а) највећим дијелом, и огроман удио апликација за *Android* су такође написане у *JAVA*-и (и *Kotlin*-у), због једне врло јединствене ствари - *JVM* (*Java Virtual Machine*), механизма који се без обзира на архитектуру или софтвер који се налази „испод“ њега увијек прилагођава подлози и омогућава извршавање *JAVA* кода. То је, у случају сваке апликације, омогућавало на врло једноставан начин да она ради без обзира на то која врста *Android*–а се налази на уређају, али је омогућавао исто то и за сам Android OS који није зависио од тога која верзија Linux Kernel-а се налазила на уређају. Kotlin као надоградња и побољшање *JAVA*-е је задржао овај начин рада те штавише користи исту *JVM* као и *JAVA*.

2.3 Kotlin програмски језик

Kotlin је програмски језик са отвореним приступом коду (*eng open source*) настао као жеља *JetBrains* предузећа да своје огромно искуство стечено прављењем радних окружења за програмирање у бројним програмским језицима као што су: *JAVA, Python*, и др. искросите у израду програмског језика који ће објединити све предности тада највише коришћених језика као што су *JAVA, C#, Python, Scala*, итд. и обједине их све у нови “језик опште намјене”. Рад на стварању овог језика започео је 2010. године под вођством Андреја Бреслава у Русији, а прва објављена верзија на тржиште је изашла 2011. године.

По свом начину рада *Kotlin* је најсличнији *JAVA* програмском језику, те га многи сматрају „надоградњом *JAVA*-е насталом у Русији“. Стога баш као и *JAVA* и *Kotlin* добија своје име по истоименом острву, у заливу између Финске и Русије. И заиста *Kotlin* у много чему подсјећа на *JAVA*-у, најприје по начину превођења (eng compiling) гдје се сав *Kotlin* код у **.kt** датотекама преводи у *JAVA*-ине .class датотеке са машинским наредбама (тзв. “*byte code*”) који се затим шаље и покреће на истој *JVM* машини као и *JAVA*. Тиме је Kotlin наслиједио изузетну прилагодљивост *JAVA*-е било којој архитектури, било којег уређаја што ће умногоме помоћи његовом ширењу на тржишту.

Kotlin је „статички типизиран“ објектно оријентисан језик, као и већина горе наведених језика којима развојни тим бијаше надахнут приликом његове израде, но оно што *Kotlin* издваја од његових прједака и даје му превласт над већином осталих језика данас су гомиле надоградњи као и библиотека.

Kotlin прије свега пружа одлично рјешење за изузетно несносну потешкоћу различитих случајева рада са неодријеђеним и непостојећим вриједностима (различитих врста!), најприје уводећи разлику између њих са засебним појмовима: null, Nothing, Unit, Any и nullable. Док се у *JAVA*-и скоро све провјере и уврштавања вриједности дешавају прије компајлирања и узрокује често враћање *NullPointerException* и сличних грјешака, у *Kotlin*-у се свака од ствари дешава у различитим дијеловима превођења гдје се нпр. Unit као ознака за неодријеђену функцију добавља накнадно, приликом извршавања, и тако не само доста смањује учесталост грјешака приликом компајлриања, него и постојање *nullable* типова, тј. типова промјењљивих које могу узети и ***null*** вриједност. За разлику од *JAVA*-е, гдје су null могли бити само сложени типови (објекти) овдје је то могуће и код основних типова као што су Int, Float, String, Boolean, итд. Иако ријеч *nullable* не постоји као дио синтаксе *Kotlin*-а ови типови промјењљивих се записују са ? знаком: *Int?, String? МојаКласа?* итд. и *Kotlin* прави итекако јасну разлику између њих и *non-nullable* типова.

```
fun foo(nullableParam: Int?) : Unit
{
    var myInt : Int = 5;
    var myNullableInt : Int? = null;
    myNullableInt = nullableParam;
    myInt = myNullableInt;
    myInt = nullableParam?:0;
    myNullableInt = nullableParam?:5;
    myInt = myNullableInt;
}
```

Слика 2.3 примјер рада и распознавања Nullable вриједности у Kotlin-у

Као што видимо на слици 2.3 рад са *nullable* типовима (нпр Int?) тј. неодријеђених вриједности је потпуно безбиједан јер ће се *Kotlin* и прије компајлирања вјешто бринути о њој, те нас увијек опомињати и водити рачуна да не бисмо случајно покушали користити неодријеђене вриједности на недозвољене начине. Такође, *Kotlin* ће омогућити да на једноставан начин и омогућимо лаган прелазак из неодријеђених у одријеђене типове помоћу условне додјеле вриједности са оператором ?: (након : стављамо вриједност коју ћемо узети у случају да *nullable* промјењљива тренутно има вриједност коју *non-nullable* (нпр. Int) промјењљиве не могу садржати.

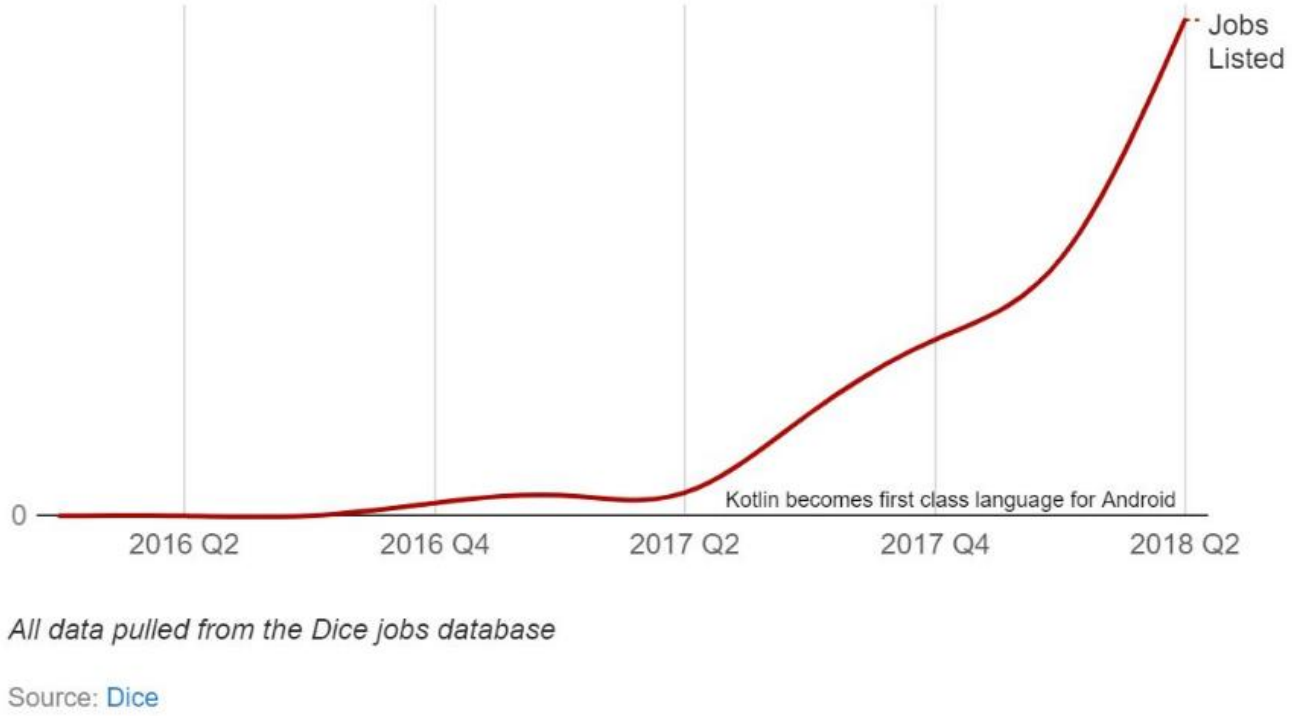
Како би *Kotlin* што лакше замијенио огромну количину софтвера за не само *Android* уређаје него и за *Web* апликације, и бројне друге платформе на којима је *JAVA* због своје прилагодљивости усљед *JVM* механизма имала превласт над свим ранијим језицима, творци *Kotlin*-а успијевају

осмислити начин како у потпуности омогућити свима лак прелазак на *Kotlin* омогућавајући потпуно уклапање старог софтвера куцаног у *JAVA*-и *JavaScript*-у па чак и бинарном коду куцаном за старије *JVM* у новим *Kotlin* пројектима.

Kotlin такође додаје потпуно нов начин прослеђивања функција као параметара, као и смијештања истих у промјенљиве и прављење безимених функција унутар самих наредби извршног кода (*lambda functions*) из *C#* -а. Но, као што је случај са главним предностима *JAVA*-е и оне из *C#* -а такође бивају надограђене и усавршене. Поред већ поменутог прослеђивања функција као параметара (што је уистину било могуће у *C* и *C++* -у преко показивача) омогућено је и прослеђивање lambda функција, али и прихват неодријеђених функција као параметара, које остају неодријеђене све до тренутка извршавања када се прослеђују као параметри.

Већина већ описаних могућности итекако бијаше надахнута Scala програмским језиком, који такође посједује (скоро) све од поменутих ствари, те је надоградња таквог језика свакако била један од најтежих изазова али и главних покретача за настанак *Kotlina*. Управо због тога одлука је пала на кориштење *JVM* механизма како би се омогућио програмски језик разноликих могућности налик на *Scala*-у усавршити тако што ће се његово веома споро превођење убрзати на ниво *JAVA*-иног превођења, као и омогућити бољу прилагодљивост различтиим архитектурама (због своје *Linux* онове која је мала и способна радити и са врло ограниченим хардвером намјенских рачунарских система).

Све нове могућности обезбјеђују *Kotlin*-у потпуну надмоћ над свим до тада доступним програмским језицима, а потпуно уклапање и са старим *JAVA* пројектима доводи до незапамћено брзог раста употребе *Kotlin*-а на тржишту, а нарочито након 2019. године када га *Google* проглашава за главни и препоручени језик развоја Android апликација. Због наглог раста Kotlin убрзо, стварањем нових библиотека и допуна, постаје не само језик за развој програмске логике (познатији као *back-end)* него почиње замјењивати и језике који бијаху првенствено описног типа и кориштаху се за исцртавање садржаја на екрану (***GUI***). Тако све чешће виђамо *Kotlin* и у описима спољног изгледа апликације умјесто дотада устаљених описних јеизка као што су: *HTML,* CSS, *XML* и сл. У чему је најзаслужнији *Google* који и сам улаже у развој Kotlin-а, иако је то идаље језик са јавно доступним кодом, правећи низ библиотека и механизама за развој Android апликација, а свакако најпознатија библиотека за ***GUI*** развој постаје *Google*-ова *Jetpack Compose* библиотека која уводи потпуно нови начин развоја будућих *Android* апликација.



Слика 2.4 Успон Kotlin –а након подршке Google-а

2.4 Jetpack Compose библиотека

Jetpack Compose библиотека представља радно окружење за прављење (исрцтавање) ставки ***GUI*** апликација, засновано на *Kotlin* језику. Цијели развој је покренуо *Google* и иако у потпуности у њиховом власништву идаље је отвореног кода тј. јавно доступан на увид свима. *Jetpack Compose* као једна од најновијих библиотека у *Kotlin*-у (која је званично изашла на тржиште 2021. године) нам пружа потпуно нови и другачији начин гледања на развој спољног изгледа наше апликације (***GUI***) јер за разлику од већине досадашњих окружења за израду ***GUI***-а је усредсријеђена на то ШТА треба нацртати (и КАДА) за разлику од досадашњег приступа гдје се постепено, корак по корак описивало КАКО би нешто требало изгледати.

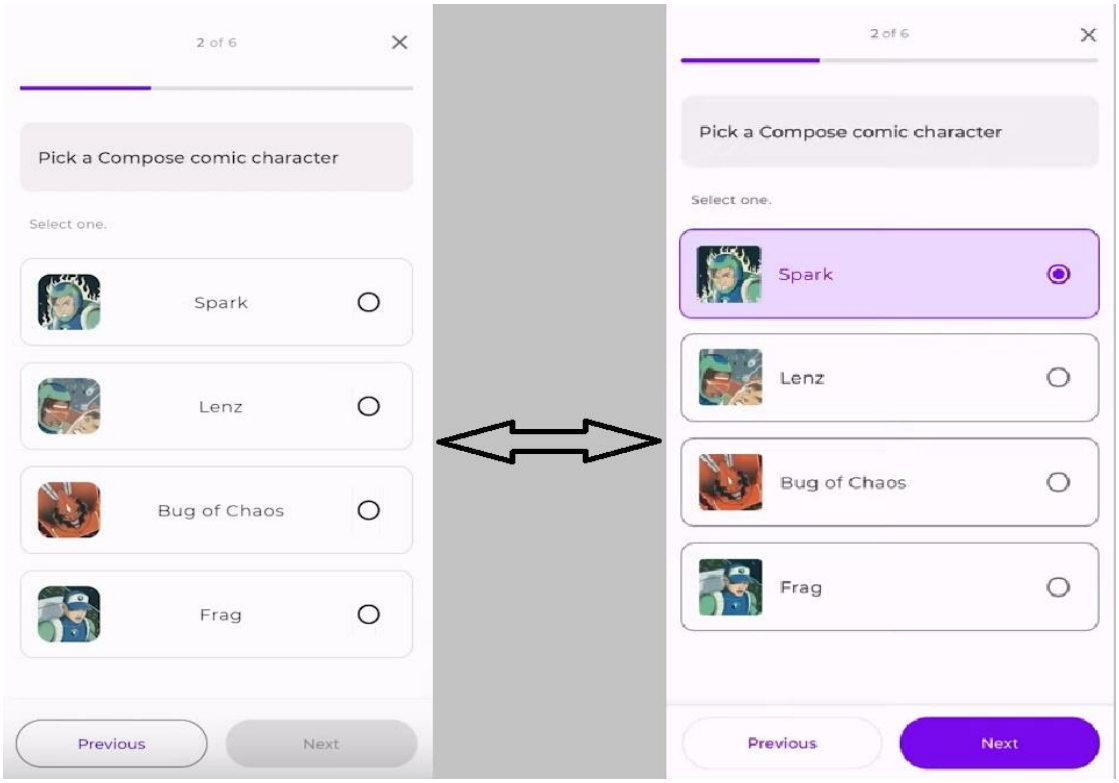
Нпр. У *XML*-у бисмо само навели називе елемената које желимо приказати, затим бисмо их морали пронаћи са findViewById те попуњавати садржај set методама у потпуно засебном дијелу, заједно са кодом и тако посебно за сваки елемент. Пошто није постојао начин да се елементи повежу морали бисмо, уколико желимо да промјена на једном од елемената утиче на остале, поставити тзв. „ослушкиваче“ (*eng Listeners*) који би пратили сваку измјену и ручно мијењали све остале елементе по потреби. У *Jetpack Compose* -у међутим, све је то могуће урадити на једном мјесту, унутар једне *composable function* задужене за описивање изгледа једног дијела прозора у цјелости, а њене различите елементе бисмо повезивали помоћу посебних *state* промјењивих задужених за праћење стања битних за рад апликације (као нпр. да ли је дошло до измјене на неком елементу екрана као

што је *RadioButton* и сл.) које су способне у потпуности самостално бринути се о примјењивању свих промјена стања на осталим елементима.

Иако ово итекако олакшава израду ***GUI***-а чинећи га много смисленијим и једноставнијим, главни недостатак *Jetpack Compose* је у томе што као нова библиотека још увијек нема све могућности које посједује развој у *XML* и његовом *Views* окружењу (као нпр. Палете са алатима која је омогућавала превлачењем и кликовима миша исцртавање елемената, или графовски приказ свих прозора у апликацији и увезивање кликом миша). Због итекако крупних недостатака, чинило се да *Jetpack Compose* ипак неће заживјети.

Кључна предност коју *Jetpack Compose* има у односу на *XML Views* је потпуна логичка раздвојеност сваке *composable function* приликом исцртавања екрана тј. Могућност поновног исцртавања само једног дијела (или појединих дијелова) екрана приликом измјене или са back-end дијела или са самог тог спољног (***UI***) дијела. Ова особина је знатно убрзала рад апликација куцаних у *Jetpack Compose* -у што је омогућило не само опстанак *Jetpack Compose* -а на тржишту него и постепени прелазак свега на *Jetpack Compose*, који би свакако могао постати главно средство за прављење ***GUI***-а у будућности, уколико пронађе начин уградити и оне могућности на које су *XML Views* програмери навикнути од раније.

Како бисмо боље објаснили начин рада и зашто је *Jetpack Compose* бржи од *XML Views*-а навешћемо сљедећи примјер:



Слика 2.5 Промјена стања између два екрана након одабира одговора

Док ће *XML* поново цртати сваку ставку на (сваком) екрану приликом сваког клика на било које од поља), *Jetpack Compose* ће када први пут кликнемо на неки од одговора, промијенити изглед тог одговора, омогућити клик на „Next“ дугме, а остале елементе оставити онаквима какви су и били (под условом да су одвојени у засебне *composable function* -е) а у свакој наредној промјени одговора, ће само вршити замјену приказа између (тј. поновно исцртавање) новоизабраног и староизабраног одговора док све друго остаје непромијењено.

Очевидно је како код сложенијих приказа ово може итекако значајно утицати на брзину извршавања апликације када не морамо непрестано изнова исцртавати цијеле прозоре.

3. Нацрт рјешења

Када смо се упознали са основама рада главних оруђа за израду наше апликације, можемо подробно осмислити и описати нашу замисао на који начин би се могла направити наша апликација.

Дакле, наш задатак је направити апликацију која ће омогућити корисницима *Anoki* услуга добавити њихове ***TV*** сервисе (канале) на *Tablet* уређају, на брз и једноставан начин пребацивати канале гледајући њихов разнолики садржај, притом имајући могућност увида у програмску шему за наредна 24 часа, не само канала који је тренутно упаљен, него и свих осталих доступних канала. Затим могућност заказивања подсјетника за гледање емисија у будућности које корисник не би желио пропустити. Након заказивања подсјетника, омогућити трајно памћење свих подсјетника и приказ истих унутар апликације у програмској шеми, те пред сами почетак емисије (1 минут раније) приказати обавјештење о почетку емисије и понудити кориснику могућност преласка на канал на којем се та емисија приказује. Уколико се корисник предомисли, и не жели гледати неку емисију, омогућити му уклањање подсјетника, као и накнадног обавјештења о истој. Уколико корисник жели гледати неколико емисија које игром случаја почињу истовремено, потребно је омогућити му да јасно одабере једну од њих у оквиру јединственог обавјештења.

Битно је напоменути да је ово само нацрт рјешења, и претпоставка како би било најбоље урадити ствари, постоји вјероватноћа мањих измјена током саме (техничке) израде, уколико се суочимо са ограничењима софтвера или пронађемо бољи начин за урадити оно што желимо.

3.1 Спољни изглед апликације

Као што је већ описано и у самим захтјевима, те и у склопу претходног поглавља, за израду спољног изгледа ћемо користити *Jetpack Compose* окружење. Оно што преостаје је одриједити из којих све дијелова се састоји наша апликација, те раздијелити их на засебне цјелине у оквиру *composable function*-а, како бисмо што мање успорили рад наше апликације.

Прво што уочавамо је мјесто за приказ главног садржаја ***TV*** канала - оно ће се простирати преко цијелог екрана, а током приказа осталих елемената, налазиће се у позадини, тј. остали елементи ће се исцртавати преко њега, дакле то је прва ставка коју можемо у потпуности издвојити од осталих.

Наредна ставка би свакако могао бити низ канала који се појављује када кликнемо било гдје

на екрану, а служи за избор канала који тренутно желимо гледати, али може нас одвести и ка списку емисија у програмској шеми одабраног канала. Стога, закључујемо како би то такође могло чинити засебну цјелину.

Треће што можемо издвојити је списак емисија из програмске шеме одабраног канала који се појави кликом на „подробнији приказ" садржаја одабраног канала из претходне ставке што ће чинити трећу цјелину.

Четврту и посљедњу издвојену цјелину приказа кроисничке спреге (***GUI***-а) чини прозорчић обавјештења (*eng alert dialog*) који служи за приказ обавјештења о почетку заказаних емисија.

Са овом подјелом ћемо моћи најбоље искористити главне предности *Jetpack Compose*–а без претјераног усложњавања кода наше апликације, зато што ћемо приликом кориштења апликације издијељене на овај начин знатно смањити количину ставки које се исцртавају поново.

3.2 Добављање садржаја који желимо приказати

Након рашчлане начина израде спољног изгледа, потребно је добавити и сам садржај који желимо њиме приказати. Пошто се захтјеви нашег задатка односе на Anoki предајника, тј. желимо приказати њихов садржај потребно је сазнати како можемо захтијевати од Anoki-ја да нам га испоруче. Како овај дио садржи доста пословних тајни међусобне сарадње *PT-PK* и *Anoki*-ја опис ће само загребати површни начин рада и приступа садржају.

Да бисмо сазнали како приступити садржају неопходно је успоставити сарадњу са неким од предајника (у овом случају Anoki као посрједник) који ће нам испоручити жељени садржај. Након успостављања сарадње добићемо приручник за употребу (*eng documentation*) њихових сервиса тј. ***API*** у којем ће бити објашњено како шаљемо захтјеве за испоруку њиховог садржаја и вршимо размјену података са њиховим послужиоцима (*eng. servers*).

Сваки корисник *Anoki* услуга се чува у њиховој бази података заједно са сервисима које је уплатио у свом пакету услуга. Anoki препознаје свог корисника тако што приликом куповине услуга додјељује свом купцу кључ за распознавање (у виду знаковног низа) назван *Anoki User Identification (AUID).* На основу њега *Anoki* зна који корисник жели користити њихове услуге, те самим тиме зна и шта је узео и шта је потребно испоручити.

Но, како постоји могућност да ови кључеви лако „процуре", *Anoki* такође са корисницима договори списак уређаја које ће користити, те их повеже на исти *AUID* кључ за распознавање . То се постиже тако што се за сваки од уређаја добави њихов јединствени кључ познатији као *Android Device Identifier* (скраћено *ADID*).

Трећи вид заштите је омогућавање услуга само на одријеђеним *IP* адресама.

Након што смо прошли све провјере потребно је преузети добављени садржај од *Anoki*-ја,

обрадити га и уклопити га у нашу апликацију. Додатне провјере које постоје, нису од нарочитог значаја, те се њима нећемо посебно бавити.

3.3 Програмска логика

Посљедње што је преостало је осмислити начин како повезати претходна два дијела, тј. Како добављени садржај од *Anoki*-ја прилагодити нашој апликацији и омогућити не само приказ, него и управљање њиме у складу са задатим захтјевима наше апликације. Како је ово заправо врло чест случај у области ***DTV***-а, јер приликом израде великих апликација увијек имамо различите врсте програмера који раде на засебним дијеловима, тако већ одавно постоје разноразни начини који су се бавили баш тиме.

Управо због тога настају посебне врсте грађе кода свих апликација које омогућавају двије кључне ствари: 1) јасну одвојеност и што већу независност сваке цјелине 2) могућност лаког увезивања датих цјелина. Рјешења која су произашла називају се обрасцима грађе софтвера (*eng Software Design Patterns),* неки од најпознатијих су: *Model-View-Controler* (***MVC***) и *Model-View-ViewModel* (***MVVM***).

У области развоја Android апликација, најзаступљенији на тржишту је ***MVVM*** образац, који ћемо и ми користити, али најприје морамо описати како он изгледа, те појаснити његов начин рада.

MVVM образац добио је назив по 3 главне раздвојене цјелине: Подаци (*Model*), Прикази (*View*) и Програмска Спрега/Логика (*ViewModel*). Поред ове 3 имамо и још неколико мањих цјелина које нису обавезне, те постоје у зависности од потреба апликације. С тим у вези нећемо се освртати на све могуће цјелине, него ћемо споменути само оне које ће нама бити потребне, са наравно главним освртом на 3 најважније, M-V-VM. Главна сврха прављења свих ових цјелина је:

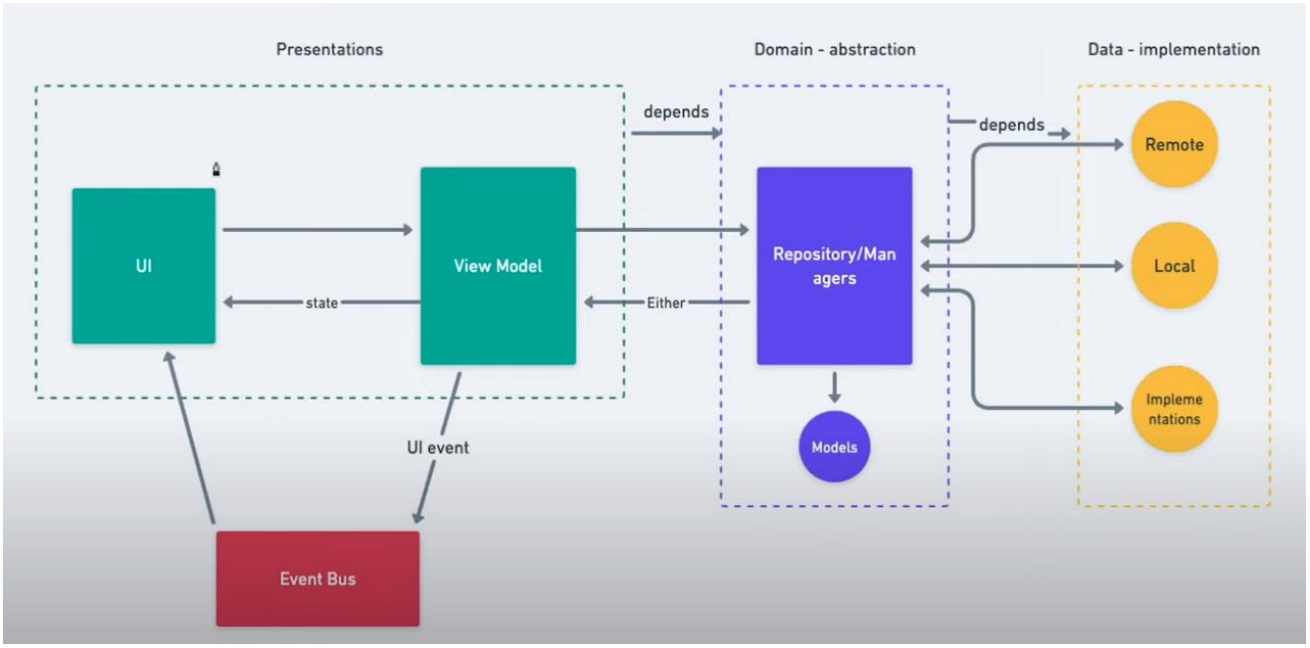
а) олакшано тестирање апликације (лакше проналажење грјешака),

б) лакше одржавања (лакше је додавати нове ствари и дорађивати постојеће),

в) повећана прилагодљивост (замјеном неке од цјелина омогућавамо да наш код може радити и за неке потпуно друкчије врсте података или уређаја

Главна намјена ***MVVM***-а је да, раздјељујући задужења свакој од засебних цјелина олакша проналажење грјешака, избјегне све могуће зависности од врсте уређаја предајника, корисника, радних окружења и база података које се користе. То постижемо правећи слојевиту рашчлану, прављењем засебних промјенљивих које чувају податке о стањима у којима се налазе ставке с главног приказа (***UI***-а) или спреге ка логичкој цјелини (*ViewModel*–а). Код сваке апликације, писан

по ***MVVM*** обрасцу је распоређен у 3 главна слоја приказана на слици испод:



Слика 3.1 Нацрт MVVM обрасца грађе апликација

Сваки од датих слојева представља један од 3 нивоа апликације:

А) ***presentation*** слој се бави прије свега изгледом апликације, али садржи и спрегу ка осталим дијеловима апликације, која је врло често уско везана за сам опис њеног изгледа (зато што она садржи промјењиве које описују стање различитих ствари с главног приказа, такође, приказ позива ствари из спреге). Он је најчешће рашчлањен на 2 дијела:

1) који садржи главни приказ и све чиниоце спреге ка њему (најчешће називана исто као и сам главни приказ) и

2) који садржи мање цјелине (остале приказе који се позивају унутар неког другог и служе као помоћни) које називамо „***components***”

Б) ***data*** слој, који садржи сирове податке и врши њихову обраду. Као што се да наслутити, ***data*** слој је поприлично уско везан за свако рјешење или ***API*** који користимо у нашој апликацији. С тим у вези у Data слоју најчешће имамо сљедеће 4 ствари:

1) функције и методе које представљају позиве ка ***API***-јима других апликација које користимо (нпр. у овом случају *Anoki* сервиси), смјештени у ***remote*** одјељак (за удаљене позиве)

2) сирове податке у изворном облику (у којем их добијамо од ***API***-ја које користимо), које смијештамо у ***models*** одјељак

3) функције и методе које обрађују сирове податке и прилагођавају их оном облику који је потребан за управљање и приказ у ***Presentation*** слоју, смјештени у ***repository*** одјељак (или

implementation , зато што представљају накнадно одријеђене функције и методе из трећег ***Domain*** слоја)

4) (необавезан) одјељак ***local*** који садржи све што је неопходно за опис Базе Података, уколико је користимо: описи табела, веза између њих, упита које можемо позивати над табелама, итд.

B) ***Domain*** слој, слој који служи за размјену између ***Data*** и ***Presentation***. Он садржи сву позадинску логику апликације која се извршава ван спреге ка главном приказу (тј. ViewModel-a) ту се налазе најчешће 2 ствари:

1) подаци с којима наша апликација управља како би исцртала и приказала жељени садржај (смјештени у одјељку models)

2) спискови функција и метода, неопходних за обраду добављених података са API-ја или Базе података како би могли бити приказани преко ***Presentation*** слоја. Главна сврха овог одјељка, који се зове ***repository*** је смањење зависности од појединачних ***API***-ја или база података које користимо, и повећање прилагодљивости нашег програма различитим Базама Података и ***API***-јима, уколико исправно усвоје (eng *implement*) правила приступа остатку наше апликације.

Управо постојање овог слоја је разлог његове велике распрострањености у развоју *Android* апликација, јер се нарочит нагласак ставља на повећање прилагодљивости различитим ***API***-јима. Више о изради и примјени овог обрасца у наредном поглављу.

4. Израда рјешења

Са завршеним нацртом рјешења, имамо довољно предзнања како бисмо могли прићи на његову израду.

Увиђамо најприје да је потребно пронаћи начин како ћемо добавити садржај за корисника од стране *Anoki*-ја (тј. Како добавити садржај ***TV*** канала и њихове програмске шеме). Очигледно, рјешење за то налазимо у *Anoki* упутству, али видимо да се њиховом ***API***-ју приступа преко *GET*, *PUT* и осталих *HTTP* захтјева (преко мреже), које не можемо користити без приступа интернету. Како бисмо омогућили приступ апликације интернету неопходно је додати сљедећу измјену унутар **`AndroidManifest.xml`** датотеке:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET"/>

    <!-- added this: android:name=".LiveTvEpgApp" -->
    <uses-permission android:name="android.permission.POST_NOTIFICATIONS" />
    <application
        android:name=".LiveTvEpgApp"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
```

Слика 4.1 Додавање дозволе коришћења интернета у AndroidManifest.XML

Када смо омогућили приступ апликације интернету потребно је утврдити на који начин можемо правити захтјеве и слати их *Anoki* ***API***-ју. За то нам је потребно посебно окружење које ће *HTTP* ***API*** претворити у Kotlin и обрнуто. *RETROFIT* је типски безбиједан *HTTP* клијент за *Android* који ће нам омогућити баш то, да из кода шаљемо *HTTP* захтјеве када се за то стекну услови. Како бисмо користили *RETROFIT* неопходно је укључити га у грађу наше апликације, тј. додати сљедеће зависности унутар **`build.gradle.kts`** датотеке унутар app директоријума:

```
    //RetroFit
    implementation("com.squareup.retrofit2:retrofit:2.9.0")
    implementation("com.squareup.retrofit2:converter-gson:2.9.0")
```

Слика 4.2 Додавање захтјева за RETROFIT окружење у build.gradle.kts (:app)

Међутим како бисмо могли покренути *RETROFIT* у оном тренутку када се испуне услови за слање захтјева, као и да би се апликација могла покренути и без неопходних података које добијамо од *RETROFIT*-а, морамо јавити апликацији да ће те податке добити у тренутку када јој буду потребни иако они нису доступни прије њеног покретања. Управо то се омогућује

механизмом тзв. „убризгавања зависности“ (eng *Dependency Injection* скр. ***DI***). Како бисмо омогућили овај механизам (који се од скора сматра и обрасцем развоја софтвера) морамо пронаћи окружење које нам га пружа. То је у нашем случају ***DaggerHilt***, окружење које нам омогућава брзо и лако подесити ***DI*** гдје год је нужно, без потребе да ручно правимо сваку од датих класа, објеката и позива од којих зависи рад наше апликације, а стара се и за позивање и укључивање зависности у тренутку када су нам оне неопходне. То се постиже помоћу тзв. Модула које смијештамо у ***DI*** одјељку (*eng package*) о чему ћемо више причати у наставку.

Уколико желимо сачувати податке о подсјетницима направљеним приликом претходних покретања апликације (нпр. ујутро смо заказали подсјетник за емисију која почиње у 18ч увече, када завршимо дневне обавезе) морамо пронаћи мјесто гдје ћемо трајно сачувати те податке, јер се приликом гашења апликације све сачувано у парчету ***RAM*** меморије додијељеном апликацији брише неповратно. Дакле, потребно је направити и Базу Података у којој ћемо чувати све што нам је потребно како бисмо могли „поново заказати све (старе) подсјетнике” приликом сваког новог покретања апликације.

Као што видимо, потребно је користити доста сложених ствари како бисмо направили нашу апликацију, стога израду ћемо подијелити на неколико сљедећих корака:

1) Израда корисничке спреге (исцртавање ***GUI*** ставки прилагођећних *Jetpack Compose* окружењу)

2) Преуређивање апликације (***GUI***-а) према ***MVVM*** обрасцу и правилима, те додавање програмске логике

3) уклапање кода у оквире ***DaggerHilt*** окружења и његовог извршног Модула који би управљао позивима ка ***API***-јима које користимо (*RETROFIT FAST* послужилац)

4) додавање унутрашње ***БП*** и трајног чувања преко *ROOM*-а, као и управљање подсјетницима преко *CountDownTimer* механизма

5) дорада постојећег, заштита од пуцања и додавање оличја апликације преко *Lottie*

4.1 Израда корисничке спреге

Свака *Kotlin* апликација се покреће од `MainActivity.kt` и класе MainActivity, унутар које се налази метода *onCreate,* која се позива приликом покретања апликације. Унутар ње ћемо позвати нашу главну *Jetpack Compose* функцију која ће прекривати цијели екран и садржаће остале *Jetpack Compose* функције као подцјелине из раније објашњених предности које овај начин рада пружа у *Jetpack Compose* окружењу (а првенствено због независног и засебног извршавања сваке од

функција приликом промјене стања). Према томе исцртавање наше апликације ће бити садржано у:

- **HomeScreen.kt** (главна функција за цијели екран)

- **MenuListItem.kt** (функција за исцртавање низа канала)

- **EPGlist.kt** (функција за исцртавање списка емисија на одабраном каналу)

- **EPGitem.kt** (функција за исцртавање сваке ставке са списка)

- **EPGreminder.kt** (функција која исцртава дугме за заказивање подсјетника)

- **ReminderDialog.kt** (функција која исцртава прозорчић за подсјетнике)

Свака од ових датотека садржи по једну *@Composable* функцију што назначава нашем компајлеру да се у њој налази *Jetpack Compose* код за исцртавање екрана. У *Jetpack Compose*-у свака *@Composable* функција је увијек састављена из других уграђених *@Composable* функција које описују шта желимо на екрану, свака од њих има подразумијевани параметар типа **Modifier** који служи за опис њених особина као што су висина, ширина, увученост (*eng padding*), боју позадине и сл. Неке од тих функција које користимо најчешће су:

* Surface(modifier: Modifier) – описује компајлеру да се унутар ње налази радна површина унутар које ће нам омогућити исцртавање елемената (тј. *@Composable* функција) на екран. Параметар modifier нам омогућује описати величину као и мјесто гдје ће се радна површина налазити,...

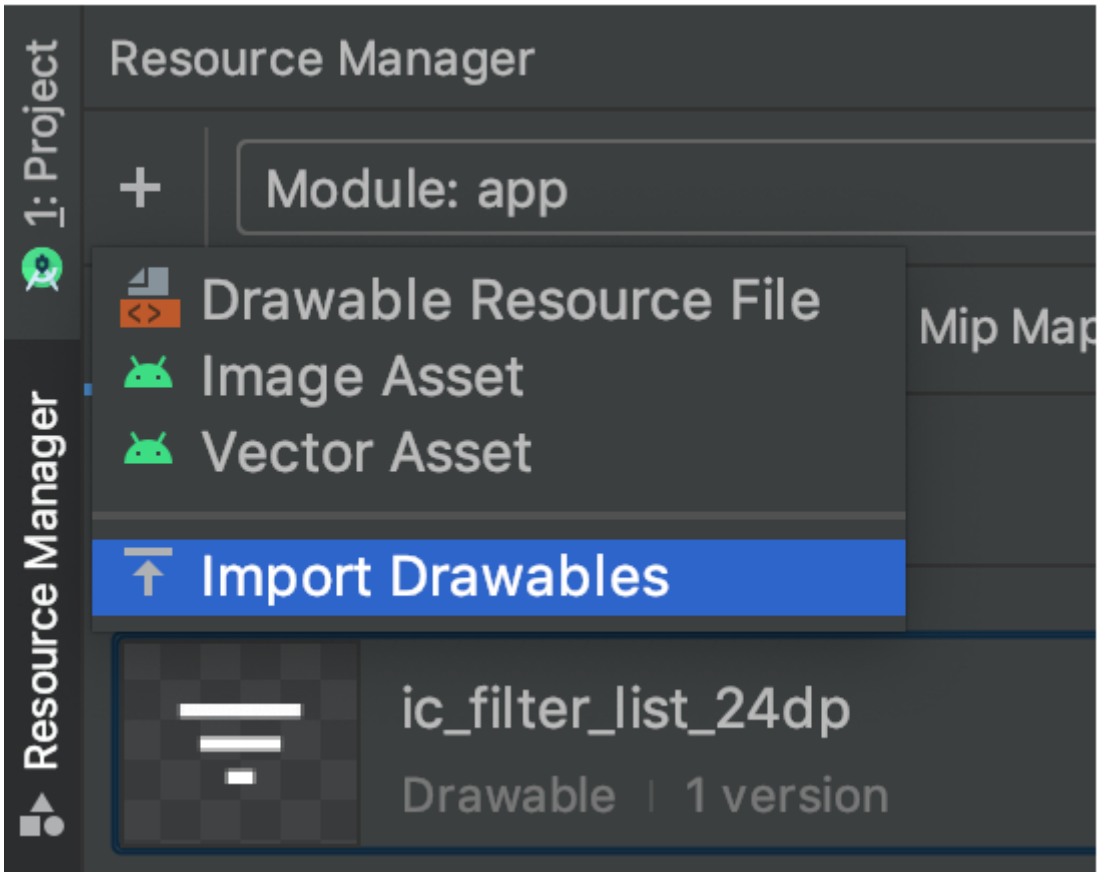
* Box(...) – описује компајлеру да све *@Composable* функције унутар њега ће бити подразумијевано збијане једна преко друге (*eng Boxed in*) изузев уколико се унутар њихових Modifier параметара не одриједе неке измјене (које ће се примјењивати у односу на својства елемента тј. Box-а унутар кога се налазе)

* Column(...) и Row(...) – говоре компајлеру да ће све *@Composable* функције унутар њега бити подразумијевано смијештане унутар једног стубца (*eng Column*) или реда/врсте (*eng Row*), једни иза других (испод односно десно један од другог). Као и Surface и Box, и овдје се размак између елемената, распоред, и све остале особине могу подешавати унутар предодријеђеног параметра Modifier који подразумијевано поприма особине од оног елемента изнад њега или описаних унутар Теме апликације (*eng Theme*)

* LazyColumn(...) и LazyRow(...) – раде исто што и Column(...) и Row(...) с том разликом што нам омогућавају и двије битне новине: 1) исцртавање не унапријед познатог броја елемената (што користимо када број елемената није познат у тренутку компајлирања, него у тек у тренутку извршавања) 2) пружају подразумијевану могућност листања садржаја (*eng scrolling*) без иакаквих подешавања, уколико он не може стати на предодријеђени му простор.

* `Text(text, modifier, color, fontSize : SizeUnit, textAlign, fontStyle,...)` – служи за приказ текста на екрану, а параметри *color*, *textAlign*, *fontStyle* имају већ предодређени скуп могућих вриједности (*enumeration*) у односу на то шта представљају, док се fontSize због различитих величина уређаја на којима се израђује софтвер не мјери у обичним *Pixel*–има него у *Scale-independent-Pixel* (нпр 15***sp***), *text* и *modifier* су већ очигледно типова *String* и `Modifier`

* `Image(painter, contentDescription : String, modifier, contentScale)` – служи за исцртавање слике. Најприје је потребно убацити жељену слику у Android Studio унутар Resource Manager (Drawable) одјељка код прегледа грађе датотека апликације (горњи лијеви угао):



Слика 4.3 Убацивање слика преко Resource Manager-а

Свака слика коју убацимо у Resource Manager (примјер на слици ic_filter_list_24dp) добија кључ који је одрјеђује (типа Int) који је потребно прослиједити параметру `painter` преко *painterResource* конструктора на сљедећи начин:

```
import com.example.livetvepgapp.R
 ...
      Image(
              painter = painterResource(R.drawable.ic_filter_list_24dp),
              contentDescription = uiState.channelsList[i].name,
              modifier.fillMaxSize(),
              contentScale = ContentScale.FillBounds
      )
```

Слика 4.4 Примјер исцртавања слике убачене у Resource Manager

Најприје увеземо све из нашег Resource Manager-а који се налази унутар R пакета од наше апликације (у овом случају `com.example.livetvepgapp` је име пакета наше апликације), затим кључ слике (**`R.drawable.име_слике`**) треба прослиједити као параметар контруктора; `contentDescription` иако суштински не нарочито битан је ипак обавезан параметар и као вриједност прима било који стринг који представља „опис слике“ но не утиче ни на који начин на сам приказ слике. `modifier` у овом случају описује величину слике у односу на родитељски елемент, у који смо је смјестили (у овом случају ће да се прикаже преко цијеле површине родитеља), а `contentScale` говори да ће сразмјерност висине и ширине слике бити прилагођена родитељском елементу (уколико је шира развућиће се увис, а уколико је виша, развућиће се у ширину).

* `AsyncImage(...)` – је врло сличан `Image( )` и користи се када нам није познато коју слику желимо приказати у тренутку компајлирања, него тек у тренутку извршавања апликације, суштинска разлика у параметрима је једино у томе што је `painter` замијењен са `model` који прима URL на којем се налази слика, записан као *String* тип. Овиме ћемо замијенити `Image( )` када будемо користили ***Anoki***-јеве податке добављене са њихових послужилаца, а у почетку ћемо користити `Image( )` за пробне приказе са нашим, измишљеним (*eng mockup*) подацима, те послије, добавивши их, замијенити све са `AsyncImage(...)`.

* `Button(modifier, onClick)` – исцртава дугме, а `modifier` описује већ познате особине као и код претходних елемената. `onClick` служи како би описао шта ће се извршити приликом клика на дугме, и њему се додјељује блок наредби, ламбда функција или нека друга функција, више о томе у наставку.

Како би се све написано помоћу ових *@Composable* фунцкија исцртало, неопходно је позвати их из главне тачке покретања пројекта – класе **`MainActivity.kt`** . Једну од претходно наведених функција ћемо одабрати као главну (у овом случају то је **`HomeScreen.kt`**) коју ћемо позвати, а затим унутар ње исцртавати све остале (као и касније, када будемо проширивали апликацију са програмском логиком, допуњавати је са позивима ка осталим слојевима апликације). За сада наша **`MainActivity.kt`** изгледа овако:

```
...
class MainActivity: ComponentActivity() {
       override fun onCreate(savedInstanceState: Bundle?) {
           CoroutineScope(Dispatchers.Default).launch {
               withContext(Dispatchers.Main) {
                   setContent {
                        LiveTVepgAppTheme {
                             Scaffold(modifier = Modifier.fillMaxSize()) {
                                  innerPadding -> HomeScreen( modifier = Modifier
                                  .padding(innerPadding).fillMaxSize() )  }
                         } /*Theme*/ } /*Content*/ } /*Context*/  } /*Coroutine*/ } /*Create*/
} /*MainActivity*/
```

Слика 4.5 Почетни изглед **`MainActivity.kt`** класе

Најприје како би Android уопште знао да је то класа од које креће извршавање апликације, неопходно је наслиједити класу **`ComponentActivity`** и преклопити њену *onCreate* методу. Пошто исцртавање свега што смо задали изискује извијесно вријеме за извршавање (због великог броја споријих задатака (као нпр. читање базе података) тзв. *suspending functions*) неопходно је ставити га у засебну нит из скупине главних нити што се у *Kotlin*-у остварује коришћењем механизма који се зове *Coroutines & Dispatchers* (начин рада је сличан *Threads & pooling* у C++ стога се нећемо нарочито освртати на њега).

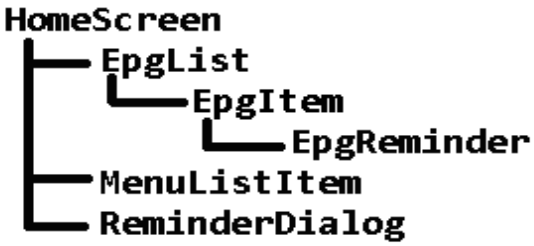
Унутар Корутине потребно је покренути нашу апликацију у складу са њеним основним подешавањима (као нпр. боје, величине слова, текста и сл.) која се налазе у њеној Теми (eng Theme). Тема увијек носи исти назив као и њена апликација (у овом случају *LiveTVepgAppTheme*) и представљена је у засебном ***ui*** пакету унутар **`Theme.kt`** као обична *@Composable* функција у којој се прави објекат Теме (`MaterialTheme`) без које наш уређај неће знати како покренути апликацију. Наша Тема изгледа овако:

```
@Composable
fun LiveTVepgAppTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    // Dynamic color is available on Android 12+
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if(darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(context)
        }

        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }

    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content,
    )
}
```

Слика 4.6 Почетни изглед **`Theme.kt`** датотеке

Након подешене Теме, пошто апликацију развијамо за таблет уређај, ради лакшег развоја бирамо предодређену расподјелу за основне дијелове прозора апликације (линије менија, статусне линије као и радне површине) прилагођене за све врсте мобилних уређаја, направљену од стране Android развојног тима – *Scaffold* који ће увијек водити рачуна о томе да се виде дијелови са тастерима на дну, као и статусна линија на врху (са стањем батерије и отворенима апликацијама, обавјештењима итд. итд.). По тој расподјели ћемо издвајати сав доступан простор за исцртавање наше апликације (*`Modifier.fillMaxSize()`*).

Изглед главног прозора ћемо подробније описати касније јер ће се у њему доста тога мијењати током осталих корака израде рјешења. До тада ћемо направити рашчлану осталих ставки тј. дијелова (*eng Components*) екрана који ће се засебно исцртавати приликом измјена (како је осмишљено према *Jetpack Compose*-у). Унутар функције за исцртавање главног прозора (`@Composable fun HomeScreen(...)`) налазе се дијелови за исцртавање елемента листе канала, (функција `@Composable fun MenuListItem(...)`), прозора за подсјетнике (функција функција `@Composable fun ReminderDialog(...)`), као и програмске шеме тренутно одабраног канала (функција `@Composable fun EpgList(...)`). Унутар се налази дио за исцртавање једне ставке програмске шеме тј. једне емисије (функција `@Composable fun EpgListItem(...)`) , а унутар ње се налази дио који исцртава дугме за заказивање подсјетника (функција `@Composable fun EpgReminder(...)`). Примјећујемо да ово подсјећа на стабло по којем се извршава исцртавање корисничке спреге. Ми ћемо кренути од листова ка коријену тог стабла које, поједностављено, изгледа овако:



Слика 4.7 Изглед стабла за исцртавање корисничке спреге у *Jetpack Compose*-у

A)EpgReminder функција

Функција која исцртава дугме за заказивање подсјетника изгледа овако:

```
@Composable
fun EpgReminder(
    programmeToDraw: TvProgramme,
    modifier: Modifier,
    onClick: ()-> Unit
) { ... }
```

Слика 4.8 Потпис *@Compoable* функције за исцртавање дугмета у *Jetpack Compose*-у

Прво питање које се поставља је, чему читава засебна функција за само једно дугме? Одговор лежи управо у већ описаном начину функционисања *Jetpack Compose*-а као и сложенијег начина рада овог дугмета. Потребно је не само омогућити заказивање подсјетника преко овог дугмета, него и њихово уписивање у БП, као и покретање механизма за одбројавање времена до приказивања подсјетника. У случају поновног притискања, потребно је отказати подсјетник и обрисати све већ поменуте измјене, и на крају, извршити измјене приказа на корисничкој спрези у тренутку чим дође до промјене стања. Управо та промјена стања у тренутку је разлог потребе за издвајањем ове функције, јер ово стање се мијења не само ручним отказивањем подсјетника

(кликом на дугме) него се мијења и самим почетком емисије (дакле зависи и од тренутног датума и времена), али о томе више у самом опису заказивања подсјетника. Стога је она издвојена како би се све те измјене што прије и стално могле примјењивати засебно од исцртавања свега осталог, а њене кључне ставке су:

ПАРАМЕТРИ:

1. **`programmeToDraw`**: објекат типа **`TvProgramme`**, који представља емисију тј. програм на који се односи подсјетник
2. **`modifier`**: објекат типа **`Modifier`**, који омогућава прилагођавање изгледа компоненти остатку апликације, преузимајући родитељски **`Modifier`**
3. **`onClick`**: Ламбда функција која се извршава када корисник кликне на програм (може служити за извршење додатне логике при постављању/скидању подсјетника). Прослеђивање се врши на овај начин како би се заправо у времену извршавања добавила функција која се треба извршити из позадинског (*eng backend*) дијела са програмском логиком, која тек у том тренутку добија вриједности за своје параметре (уколико их има) и тиме стиче услове за извршавање

ИЗГЛЕД И ОПИС: *`selectedBackgroundColor`*, *`backgroundColor`*, *`reminderSetColour`*, и *`reminderNotSetColour`* су предодријељене боје које се користе у различитим стањима подсјетника. Прозор је сачињен од једног `Box` елемента који је у зависности од стања подсјетника попуњен различитим бојама и садржајем. Стање подсјетника (објашњено раније) је описано тзв. „промјенљивом стања“ `isReminderSetState` која изгледа овако:

```
val isreminderSetState: MutableState<Boolean> = rememberSaveable { mutableStateOf( programmeToDraw.isReminderSet) };
```

Механизам „промјенљивих стања“ је ново, али доста моћно средство које се често употрјебљава у *Jetpack Compose*-у (као што ћемо видјети касније) стога ћемо га укратко и појаснити. Иако наизглед подсјећа на обичне помоћне промјенљиве за праћење вриједности познатије као „flags“ много је моћнији, јер је заправо он тај који шаље захтјеве за поновно исцртавање *Jetpack Compose* компоненти чим дође до промјене његове вриједности. Он такође и брине о томе да се измјене примијене на све оне дијелове корисничке спреге које „ослушкују“ тј. ишчекују промјене на њему, као и све остале промјенљиве чија вриједност зависи од њега.

Приказ тих стања је сљедећи; уколико је подсјетник заказан потребно је на одабраном програму приказати иконицу часовника са *`reminderSetColour`* тј. са позадином свијетло плаве боје, уколико пак тај програм, односно емисија, није одабрана (није на њу корисник тапнуо) часовник ће бити мањи са позадином у боји као и картица за приказ емисије (*`backgroundColor`*) у програмској шеми (тј. провидан). Уколико подсјетник није заказан на одабраној картици се прикаузје часовник са прозирном позадином, у боји (*`selectedBackgroundColor`*) одабране картице, док се на осталим картицама сав садржај проглашава провидним (тј. не приказује се).

Б)EpgItem функција

Ова функција **`EpgItem`** је *Jetpack Compose* компонента која приказује јединицу тј. ***EPG*** емисију у оквиру листе телевизијских програма у апликацији. Она нуди могућност кориснику да кликне на програм и подеси или уклони подсјетник за тај програм и изгледа овако:

```
@Composable
fun EpgItem(
    programmeToDraw: TvProgramme,
    selectedProgramme: TvProgramme?,
    modifier: Modifier,
    onClick: (selectedProgrammeId: String)-> Unit
    onReminderClick: (selectedProgrammeId: String)-> Unit
) { ... }
```

Слика 4.9 Потпис функције за исцртавање ставке програма из EPG листе у *Compose*-у

ПАРАМЕТРИ:

1. **`selectedPrograme`**: Објекат типа **`TvProgramme`**, који представља емисију тј. програм који се тренутно одабран.
2. **`programmeToDraw`**: Објекат типа **`TvProgramme`**, који представља емисију тј. програм који се тренутно исцртава.
3. **`modifier`**: објекат типа **`Modifier`**, који омогућава прилагођавање изгледа компоненти остатку апликације, преузимајући родитељски **`Modifier`**
4. **`onClick`**: Ламбда функција која се извршава када корисник одабере неки програм тј. емисију. Ова функција је преузета од родитеља (**`EpgList`**) и служи замијенити вриједности промјењиве стања између новоодабране емисије и оне која бијаше одабрана прије ње. Такође, задужен је и за бригу о овом стању и осталих емисија тако што омогућава свакој емисији да буде одабрана у једном тренутку.
5. **`onReminderClick`**: Ламбда функција која се извршава када корисник кликне на простор у којем ће се налазити дугме из **`EpgReminder`**. У питању је иста функција која је названа **`onClick`** унутар **`EpgReminder`**, а коју на овај начин просљеђујемо са родитељског **`EpgList`** ка **`EpgReminder`** као коначном одредишту.

ИЗГЛЕД И ОПИС: *`selectedBackgroundColor`*, *`backgroundColor`* су предодређене боје које се користе у зависности да ли је емисија тренутно одабрана или не. Свака од картица за емисију је постављена унутар *`Column`* провидне позадине, што значи да ће елементи сваког програма бити приказивани један испод другог. Разлог издвајања ове компоненте у засебни стубац (иако се већ налази унутар *LazyColumn*-а у оквиру родитељске **`EpgList.kt`** је могућност да уколико корисник случајно кликне између 2 различите картице (тј. емисије) не би затворио цијели списак емисија ***EPG***-а, што је захтијевано понашање уколико се кликне изван картица емисија (улијево или

удесно, тј. све што не чини увучени простор између двије емисије).

Након тога, цијели садржај је смјештен у један ред, како би се одјељак од 80% који садржи назив и вријеме трајања емисије и простор за дугме подсјетника исцртавали један поред другог и попунили картицу по ширини (због односа ширине и дужине картице од 4:1). Затим је унутар првог елемента реда начињен подстубац за назив и вријеме трајања емисије. Ово је учињено ради боље искоришћености простора предодређеног за картице, који је довољне висине за 2 реда слова. На овај начин такође (што је и главни разлог овакве одлуке) је знатно шири простор за приказ назива и дужине трајања сваке емисије, који испрва бјеше подијељен на 2 дијела (по ширини, сваки по 40% укупне ширине), те на овај начин бјеше удвостручен (на 80% а 20% на дугме за подсјетнике). Ово корисинику омогућава видјети цијели назив емисије и њено трајање у већини случајева. Како би пак било могуће прочитати назив и дужину трајања у цјелости, оба поља су означена са *`basicMarquee Modifier`*-ом који ће уколико не постоји довољно мјеста за приказ цијелог текста, све претворити у трчећа слова те омогућити да се све прочита. У зависности од тога да ли је емисија одабрана или не бира се одговарајућа боја картице (према захтјевима корисника), те уколко јесте одабрана емисија, њену картицу је потребно увећати за 20% у ширину у односу на остале које нису одабране (по 10% са обје стране) како би била лакше уочљива. Дугме се исцртава позивом **`EpgReminder`**), а исцртава се и након што је емисија већ почела (ако још увијек траје), уколико је корисник укључио апликацију тек након њеног почетка и није био обавијештен путем прозорчића о почетку емисије.

B)EpgList функција

Ова функција приказује ***EPG*** тј. списак телевизијских програма на тренутно одабраном ***TV*** каналу и прима листу програма омогућавајући кориснику изабрати програм и управљати подсјетницима, а изгледа овако:

```
@Composable
fun EpgList(
    epgItemsToDraw: List<TvProgramme>,
    onClick: (selectedEpgProgrammeId: String)-> Unit
    onReminderClick: (selectedProgrammeId: String)-> Unit
) { ... }
```

Слика 4.10 Потпис функције за исцртавање EPG листе у *Compose*-у

ПАРАМЕТРИ:

1. **`epgItemsToDraw`**: Листа типа **`TvProgramme`**, који представља списак свих емисија тј. програма који се приказују на одабраном ***TV*** каналу
2. **`selectedProgrammeId`**: Кључ за проналажење тренутно одабране емисије типа **`TvProgramme`** у списку свих емисија које се приказују на одабраном ***TV*** каналу

3. **`onClick`**: Ламбда функција која се извршава када корисник одабере неки програм тј. емисију. Ова функција је преузета од родитеља (**`HomeScreen`**) и служи замјенити вриједности промјењиве стања између новоодабране емисије и оне која бијаше одабрана прије ње, а кориштена је и подробније појашњена у **`EpgItem`**.
4. **`onReminderClick`**: Ламбда функција која се извршава када корисник кликне на простор у којем ће се налазити дугме из **`EpgReminder`**. У питању је иста функција која је названа **`onClick`** унутар **`EpgReminder`**, а коју на овај начин прослеђујемо са родитељског **`HomeScreen`** ка **`EpgList`** (и касније **`EpgReminder`** као коначном одредишту).

ИЗГЛЕД И ОПИС: Садржај списка (тј. свака емисија) је одвојена унутар засебне *`Column`* компоненте, провидне позадине. Разлог издвајања ове компоненте у овај засебни стубац (иако се већ налази унутар *LazyColumn*-а у оквиру родитељске **`HomeScreen.kt`** је могућност да уколико корисник случајно кликне између 2 различите картице (тј. емисије) не би затворио цијели списак емисија ***EPG***-а, што је свакако захтијевано понашање уколико се кликне изван картица емисија (улијево или удесно, тј. све што не чини увучени простор између двије емисије). Такође било је неопходно омогућити затварање списка уколико се јасно кликне изван простора за картице емисија, али онемогућити га уколико се кликне на наслов цјелокупног приказа (који говори да се ради о програмској шеми за тренутни дан).

Програми се исцртавају проласком кроз Листу **`epgItemsToDraw`** у *Kotlin*-овој врсти *foreach* петље **`programmeToDraw`** постаје емисија која је на реду за исцртавање. Током проласка ћемо провјерити да ли је тренутна емисија за исцртавање она коју је корисник одабрао поредећи је са прослијеђеним кључем, уколико је **`programmeToDraw.id == selectedProgrammeId`** позваћемо **`EpgItem`** прослиједивши му ту емисију и као **`programmeToDraw`** и **`selectedProgramme`** уколико није, нећемо оптерећивати меморију прослеђивањем било чега за одабрану емисију јер нам није неопходно за исцртавање **`EpgItem`**.

На овај начин смо обезбиједили да *Jetpack Compose* омогући приказ Листе емисија, одабир емисије и подсјетника и заказивање истих на најбржи могући начин.

Г)ReminderDialog функција

Ово је *@Composable* функција која исцртава прозорчић за обавјештења о заказаним емисијама које почињу ускоро. Према захтјевима, потребно је корисника обавијестити минут прије почетка сваке заказане емисије о њеном почетку, те уколико се корисник није предомислио, и идаље жели погледати заказану емисију, одмах га пребацити на ***TV*** канал на којем ће се она ускоро приказивати. *ReminderDialog* функција изгледа овако:

```
@Composable
fun ReminderDialog(
    programmes: List<TvProgramme>,
    onConfirmClick: (channelID: String)-> Unit
    onDismissClick: ()-> Unit
) { ... }
```

Слика 4.11 Потпис функције за исцртавање прозора за подсјетнике

ПАРАМЕТРИ:

1. **`programmes`**: Листа типа **`TvProgramme`**, који представља списак свих емисија тј. програма који почињу за 1 минут (тј. за које је потребно приказати обавијестити корисника да ускоро почињу)
2. **`onConfirmClick`**: Ламбда функција која се извршава када корисник одабере програм тј. емисију коју жели гледати. Ова функција је преузета од родитеља (**`HomeScreen`**) и служи мијењању ***TV*** канала тј. пребацивању на онај канал на којем се налази одабрана емисија, а биће подробније објашњена касније.
3. **`onDismissClick`**: Ламбда функција која се извршава уколико корисник не жели погледати ниједну од заказаних емисија које почињу ускоро, позива се кликом на *Dismiss* дугме или било гдје изван прозорчића обавјештења. И ова функција је преузета од родитеља (**`HomeScreen`**)) јер се у њему, као главном дијелу корисничке спреге, налазе све везе ка *ViewModel*-у и читавој програмској логици која се извршава у позадини.

ИЗГЛЕД И ОПИС: Цијели прозор је замишљен као **`Dialog`** тј. уграђена врста прозорчића у *Jetpack Compose*-у како бисмо били увјерени да ће он бити смјештен тачно на средини екрана (и по ширини и по дужини), јер ће о томе бринути сам *Jetpack Compose*. Након покушаја кориштења већ уграђене функције у *Jetpack Compose*-у прављене посебно за обавјештења (**`AlertDialog`**) прешли смо на ручно исцртавање елемената прозора, усљед њене немогућности за приказивање више од 2 дугмета (што значи да можемо корисника обавијестити о највише једној емисији), као и чињенице да се већ уграђени распоред дугмића (*Confirm* - десно *Dismiss* – лијево) није могао мијењати што није одговарало корисничким захтјевима (Confirm - лијево а Dismiss – десно). Још једна мана **`AlertDialog`**-а је и што се ти дугмићи нису могли помијерати, стога смо се одлучили за ручно исцртавање свега користећи **`Card`** функцију.

`Dialog` је постављен тако да се омогући затварање и кликом било гдје изван његове површине (`dismissOnClickOutside = true`). Затим, његова површина, исцртана у **`Card`** је заобљених ивица, са насловом „Watchlist Reminder“. У зависности од броја емисија за које желимо приказати обавјештење, исцртавање прозора се дијели у двије врсте – обавјештење за 1 емисију, и обавјештење за више емисија на основу дужине Листе **`programmes`** из параметара. Ако је само један програм у Листи, приказује се текст у којем је наведен пуни наслов програма тј. емисије која почиње за минут, уз

дугме „WATCH NOW“. Уколико је пак више програма у листи (**`programmes.size > 1`**), приказује се текст који ивјештава корисника да више програма почиње ускоро, те се појављују дугмићи за сваки програм са којим корисник може да одабере „WATCH (име програма)“ за сваки од њих понаособ. Све је распоређено у једном реду (*`Row`*), како бисмо Watch и Dismiss дугмиће размијештали једне поред других. Уколико постоји више програма који се требају понудити кориснику, дугме за сваки програм се исцртава унутар заједничког ступца (`Column`) како бисмо обезбиједили да се приказују један испод другог. Свако *WATCH* дугме, садржи и ознаку ***TV*** канала и емисије на којој се она налази како би се прослиједило назад програмској логици која тако сазнаје који ***TV*** канал корисник жели гледати позивом `onConfirmClick(tvChannelID)`. Док се кликом на Dismiss или било гдје ван прозора позива `onDismissClick()`.

На овај начин постижемо смислен и већ устаљен начин кориштења за управљање прозорчићима (*pop-ups, dialogs,..*) корисника из других апликација и омогућавамо брз одабир програма који жели гледати у реалном времену.

Д)MenuListItem функција

Листа канала ће се налазити с лијеве стране екрана, као стубац са подразумијеваним превлачењем (у *Kotlin*-у *`LazyColumn`*), а сваки члан те листе тј. ***TV*** канал ћемо засебно исцртавати због лакшег примјењивања измјена његових стања: да ли се тај ***TV*** канал тренутно приказује на главном екрану, је ли његов програмски водич отворен и сл.). Исцртавање те ставке изгледа овако:

```
@Composable
fun MeunListItem(
    tvChannel: TvChannel,
    modifier: Modifier = Modifier,
    activeChannelId: String,
    selectedEpgChannelId: String,
    onClick: ()-> Unit
) { ... }
```

Слика 4.12 Потпис функције за исцртавање *TV* канала

ПАРАМЕТРИ:

1. **`tvChannel`**: Објекат типа **`TvChannel`**, означава ***TV*** канал који се тренутно исцртава.
2. **`modifier`**: Објекат типа **`Modifier`** који омогућава прилагођавање изгледа компоненти. остатку апликације, преузимајући родитељски (у овом случају из **`HomeScreen`**)
3. **`activeChannelId`**: кључ за ***TV*** канал који корисник гледа у тренутку исцртавања
4. **`selectedEpgChannelId`**: кључ за ***TV*** канал чији је списак емисија корисник отворио (уколико је отворен, уколико није имаће неважећу тј. null вриједност). У питању је иста функција која је названа **`onClick`** унутар **`EpgReminder`**, а коју на овај начин прослеђујемо са родитељског **`EpgList`** ка **`EpgReminder`** као коначном одредишту.

5. **`onClick`**: Ламбда функција која се извршава када корисник одабере неки ***TV*** канал. Ова функција је преузета од родитеља (**`HomeScreen`**) и има више намјена. Уколико се приказују само ***TV*** канали, служи за мијењање ***TV*** канала који желимо гледати. Уколико је већ отворен приказ ***EPG***-а (кликом на посебну стрјелицу о којој ћемо више причати у одјељку о **`HomeScreen`**-у) кликом на ***TV*** канал у листи заправо требамо мијењати ***EPG*** тј. мијењаћемо ***TV*** канал чији се ***EPG*** приказује.

ИЗГЛЕД И ОПИС: Ова компонента приказује један ***TV*** канал у Менију (смјештеном са лијеве стране екрана у **`HomeScreen`**-у) са одговарајућом сликовном ознаком ***TV*** канала и различитим описним ознакама приликом исцртавања како би се лакше разликовали ***TV*** канали у различитим стањима:

1) ***TV*** канал који се тренутно гледа

2) ***TV*** канал чији се ***EPG*** тренутно приказује

3) остали ***TV*** канали

Приказивање канала се остварује на сљедећи начин: **`Surface`** се, као и до сада, користи као радна површина за компоненту. Боје су одријеђене према Теми апликације и **`modifier`**-у (`MaterialTheme.colorScheme.primary`), а додатно се уређује увлачењем (*eng padding*), обликом (заобљени углови) и величином итд. **`Row`** и **`Column`** Користе се за распоређивање приказа. **`Row`** служи како би ограничио ширину коју ***TV*** канал може заузети, а **`Column`** како би се ограничила његова висина.

Уколико се ***TV*** канал који исцртавамо тренутно гледа (`tvChannel.channelId == activeChannelId`) потребно је оивичити га свијетлозеленом бојом, међутим уколико се отвори приказ ***EPG***–а потребно је уклонити је, и замијенити цртежом малог *„play button“*-а, а ***TV*** канал чији је ***EPG*** тренутно приказан потребно је оивичити свијетлоплавом бојом. Приликом сваког новог клика неопходно је освјежавати стања као и већ поменуте приказе који описују дата стања, такође омогућено је и да се стања нагомилавају, те уколико се истовремено и гледа и приказује ***EPG*** неког ***TV*** канала приказаће се и *„play button“* и свијетлоплава ивица. Остале ***TV*** канале (који нити се гледају нити чији се програмски водич приказује) није потребно додатно мијењати у приказу, но само уклонити све додатке које су на себи имали. Све описано се дешава простим кликовима по ***TV*** каналима на које се одговара позивом `onClick()` функције, која за сада у коду, а касније (по MVVM обрасцу) ће ову, већ описану логику одрађивати унутар посебне методе под називом `onClick()` која се прослеђује из **`HomeScreen`**–а преко параметра **`onClick.`**

За приказ ознака и оличја ***TV*** канала користићемо **`Image(...)`** и наше, измишљене слике ћемо подметати умјесто правих, док не успоставимо везу са послужиоцем и не превучемо праве

податке са *Anoki* ***TV*** канала, а након тога ћемо све **`Image`** замијенити са **`AsyncImage(...)`**, што ће нам омогућити да се исцртавање изврши након што подаци са *Anoki* послужиоца стигну (како бисмо спријечили пуцање апликације приликом покретања у случају да до исцртавања дође прије него што стигне одговор са *Anoki* послужиоца.

Ђ)HomeScreen функција

Ова функција представља не само главну функцију за исцртавање приказа екрана ка кориснику, него и спону ка цјелокупној прорамској логици преко међуслоја ***MVVM*** обрасца, тј. *ViewModel*-а, а преко њега и даљње, суште и дубоке програмске логике (као нпр. добављање података са послужиоца и из ***БП***, обрада, претрага, брисање, итд. итд.). С тим у вези, битно је напоменути да ће ова функција претрпити и доста измјена и допуна у даљим корацима развоја наше апликације, али и да ће служити за размјену података и са осталим функцијама упоредо са *ViewModel*-ом. Управо због њене важности, а према обрасцу, *ViewModel* ће носити њено име, а како будемо проширивали апликацију, прошириваћемо и ову функцију, те ће о њој бити доста ријечи и касније, а за сада, она изгледа овако:

```
@Composable fun HomeScreen(modifier: Modifier = Modifier)
```

ИЗГЛЕД И ОПИС: Параметар `modifier` је преузет из **`MainActivity`** класе. Већина програмске логике која је описана у `onClick` ламбдама претходних функција се за сада налази баш у **`HomeScreen`**-у гдје се оне записују или као мање, помоћне функције или се читаве дефиниције тих функција просљеђују као параметар унутар великих заграда { }. Као и до сада, користимо `Surface` као подлогу за исцртавање садржаја, која у овој функцији заузима цијели екран. Све остало ће се налазити унутар `Box` јер желимо „збити садржај“ преко цијелог екрана (приказ Листе канала, ***EPG*** списка емисија, прозорчића за подсјетнике итд.). Као главни дио ове функције свакако јесте приказ садржаја (тј. емисије) одабраног ***TV*** канала преко цијелог екрана, који се преноси уживо са *Anoki* послужиоца (за сада се ту налази слика умјесто правог садржаја који ћемо покупити након успостављања везе са послужиоцем). Кликом било гдје на таквом екрану се отвара Листа ***TV*** канала описана у **`MenuListItem`** поред које се исцртава посебна стрјелица за отварање ***EPG*** списка емисија са одабраног ***TV*** канала. Кликом било гдје изван приказаних Компоненти можемо брзо затворити све и наставити неометано гледати садржај одабраног ***TV*** канала. Све описано је смјештено унутар `Row`-а како бисмо обезбиједили да се исцртавају с лијева на десно, једно поред другог.

Као први елемент те врсте се налази стубац за исцртавање елемената са уграђеним клизачем (`LazyColumn`) унутар којег се исцртавају сви ***TV*** канали позивом **`MenuListItem`**. Испод ње налази се постепено затамњена позадина која се протеже од црне ка провидној од лијеве ивице екрана све до стрјелице за приказ ***EPG*** списка емисија са одабраног ***TV*** канала. Већ поменута стрјелица поред

приказа ***EPG***-а садржи анимацију, прије исцртавања ***EPG***–а показује у смијеру гдје ће се он исцртати, а након клика и исцртавања ***EPG***–а она се окреће у супротном смијеру и показује ка Листи ***TV*** канала. Овиме се постиже врло смислено схватање корисника да кликом на њу може „извући додатне податке“ и „повући их назад“ када му више нису потребни без оптерећивања додатним текстом или дугмићима који би преплавили екран.

```
val epgArrowRotation by animateFloatAsState(
    targetValue = if (uiState.isEpgMenuDisplayed) 180f else 0f,
    animationSpec = tween(durationMillis = 1000),
    label = "EPG Arrow Rotation" );
Image( //EpgMenu rotatable arrow
    painter = painterResource( id = R.drawable.white_right_arrow_pointer),
    contentDescription = "pointerClose",
    modifier = Modifier
        .size(60.dp)
        .align(Alignment.CenterEnd) //effectively same as alignment AlignmentCenterVertically
        .clickable { viewModel.onAction(HomeScreenAction.OnEpgMenuArrowClick
        .background(Color.Transparent)
        .graphicsLayer( rotationZ = epgArrowRotation),
    contentScale = ContentScale.Fit,);
```

Слика 4.13 Изглед функције за исцртавање *TV* канала

Као што видимо анимација се прави додјелом низа пријелазних *float* вриједности које послије додајемо у `modifier` преко `graphicsLayer` атрибута, на крају потребно је то примијенити над елементом (у овом случају слика стрјелице). Дакле, *`animateFloatAsState`* ће окренути стрјелицу за 180 степени када се прикаже ***EPG*** списак, а вратити је назад када се списак скрије. Овдје такође видимо (унутар атрибута `clickable`) како ће изгледати позиви функција програмске логике које ћемо из **`HomeScreen`**-а постепено пребацивати и проширивати у *ViewModel*-у и осталим дијеловима апликације.

На крају `Row`-а се налази ***EPG*** списак, који отварамо кликом на већ објашњену стрјелицу. Сви ови прикази су условни, дакле, морају бити испуњени одријеђени услови како би се исцртали на екрану (тј. приказали кориснику). То постижемо коришћењем промјењивих стања о којима смо причали раније. Овдје користимо за то двије промјењиве *`isEpgMenuDisplayed`* и *`isChannelsListVisible`*. Посљедњи дио **`HomeScreen`**-а је такође условни приказ прозорчића за подсјетнике који је остварен позивом **`EpgReminder`** функције а којим управља *`isReminderVisible`*. Све промјењиве стања ће послије бити пребачене у посебан дио ***ViewModel***-а под називом **`HomeScreenUiState`** као што се види у коду за приказ стрјелице, а о чему ћемо више причати у наредном кораку израде.

4.2 Преуређивање апликације према *MVVM* обрасцу

До сада смо се бавили готово искључиво исцртавањем компоненти корисничке спреге тј. ***GUI***-а користећи само врло просту програмску логику тамо гдје је било неопходно. Но као што

смо могли видјети кључни недостаци овога су то што не можемо заправо заказати подсјетнике за гледање емисија. Можемо направити захтјев за заказивање, али немамо никакав начин за покретање тих подсјетника, самостално, када за то дође вријеме, без икаквог клика или било које друге врсте радње са корисничке стране. Такође, са досадашњим начином рада немамо никакву могућност добављања правих података са *Anoki* послужиоца, а поред тога немамо ни могућност сачувати све наше измјене и подешавања трајно, како нам се не би сви заказани подсјетници брисали приликом сваког новог покретања апликације. Управо због тога је неопходно извршити свеобухватно преуређивање апликације на начин који ће нам омогућити ове и остале напредније радње. За то ћемо користити ***MVVM*** образац уређивања апликације.

Према ***MVVM*** обрасцу, све из претходног поглавља се бавило углавном исцртавањем разноразних дијелова корисничке спреге, користећи врло мало програмске логике и података. Како су ове 3 ствари у ***MVVM*** обрасцу врло јасно раздвојене, неопходно је извршити значајне измјене. Све до сада рађено представља ***View*** дио, из ког ћемо избацити сву програмску логику (нпр. дефиниције ламбда функција које се позивају приликом разних кликова са *onClick* параметрима) и пребацити је у ***ViewModel*** дио, спомињан у претходном кораку. Уколико је у питању сложенија врста програмске логике (као нпр. размјена података са ***БП*** или послужиоцем), она ће бити додатно раслојена у ***domain*** и ***data*** слојеве, као што је приказано на слици 3.1, страна 16.

Да бисмо даље радили са подацима, неопходно је направити нашу уопштену представу како би требали изгледати објекти ***TV*** канала и ***TV*** програма (тј. емисије), који се користе унутар наше апликације. Према ***MVVM*** обрасцу са слике – потребно је направити ***domain*** одјељак у којем ће се налазити пакет ***models*** за ове класе:

```
package com.example.livetvepgapp.domain.models

data class TvChannel(
    val channelId: String,
    var logo : String,
    var name: String,
    var isFavourite: Boolean = false,
    var genre: List<String>,
    var playbackURL: String,
    var EPG: MutableList<TvProgramme> = mutableListOf(),
    var EPGtimestamp : Long? = null
);
```

```
package com.example.livetvepgapp.domain.models

data class TvProgramme
(
    val id: String, //id = channelId + ":" + contentId + " " + startTime,
    val contentId : String, //AnokiContentId
    var name: String = "Unknown",
    var duration: String = "",
    var startDateTime : ZonedDateTime,
    var endDateTime : ZonedDateTime,
    var isReminderSet : Boolean = false,
    var reminderTime : String? = null,
    var description : String = ""
);
```

Слика 4.14 Наша представа *TV* канала и *TV* програма

Како су ове класе врло сличне и у JAVA-и и у C++-у нећемо се нарочито осврнати на њих, но ћемо само додати напомену да је због смањења броја линија кода, у *Kotlin*-у одлучено направити посебну врсту класа које ће подразумијевано бити налик тзв. *POJO* класама (*Plain Old Java Object*). Како је њихова главна сврха била складиштење и приступ подацима, такве класе су назване ***data class*** и у *Kotlin*-у је за њих довољно само написати изглед параметарског конструктора (те додијелити подразумијеване вриједности атрибутима по потреби), без писања *get/set* Метода, Деструктора итд. а *Kotlin* ће сам уочити да их нема и написати их за нас.

Сљедеће што је на реду је израда ***ViewModel*** дијела. Он се најчешће састоји из класе у којој се чувају све промјенљиве стања како би биле на једном мјесту и лакше се пратило стање, као и да би се заштитиле од недозвољених промјена вриједности. Та класа, као и ***ViewModel*** носе назив по главној функцији ***GUI***-а у којој се користе.

На самоме крају преостаје нам да поред издвајања промјенљивих стања издвојимо и преосталу програмску логику из фунцкије, то ћемо учинити записивањем свих onClick ламбда функција на једно мјесто унутар ***ViewModel*** класе, као њене методе. Како би се лакше испратило преношење свих тих метода, установљен је договор да се по ***MVVM*** обрасцу све методе са корисничке спреге смијештају у засебан Интерфејс који ће такође носити исти предметак као и главна функција ***GUI***-а. Такође је потребно све сложеније радње (као нпр добављање података) раздвојити у засебне Интерфејсе и класе за чије називање нема нарочитих правила, те добијају имена по својој намјени, у нашем случају то ће за сада бити за добављање података о ***TV*** каналима.

На основу ових правила потребно је направити сљедеће класе и Интерфејсе: **`HomeScreenViewModel`**, **`HomeScreenAction`**, **`HomeScreenUiState`** и **`ChannelRepository`**.

A)HomeScreenAction Интерфејс

Овај Интерфејс садржи све методе које у претходном кораку бијаху записане као Ламбда функције и извршаваху се на неки клик. Из тог разлога није потребно посебно објашњавати шта свака од ових метода ради него ћемо их само побројати.

```
package com.example.livetvepgapp.presentation.homescreen

sealed interface HomeScreenAction
{
    data objet OnScreenClick : HomeScreenAction
    data class OnChannelClick(val channelID: String) : HomeScreenAction
    data object OnEpgMenuArrowClick : HomeScreenAction
    data class OnEpgItemClick(val programmeID: String) : HomeScreenAction
    data class OnProgrammeReminderClick(val programmeID: String) : HomeScreenAction
    data class OnReminderConfirmClick(val channelID: String) : HomeScreenAction
    data object OnReminderDismissClick : HomeScreenAction
}
```

Слика 4.15 Списак метода у *HomeScreenAction* Интерфејсу

- onScreenClick – кликом било гдје на екрану отвара Листу канала или затвара све што је отворено
- onChannelClick – кликом на ***TV*** канал мијења ***TV*** канал за гледање или приказ ***EPG***-а
- onEpgMenuArrowClick – кликом на стрјелицу приказује и скрива ***EPG*** одабраног ***TV*** канала
- onEpgItemClick – кликом на ставку тј. емисију одабира је или поништава одабраност
- onProgrammeReminderClick – кликом на дугме заказујемо и отказујемо подсјетнике

- onReminderConfirmClick – кликом на дугме у подсјетнику прелазимо на ***TV*** канал заказане емисије
- onReminderDismissClick – кликом на дугме бришу се подаци о свим заказаним емисијама о којима смо обавијештени у подсјетнику

Б)HomeScreenUiState класа

Све до сада коришћене промјењиве стања, као и оне које ћемо касније користити смијештамо у ову класу како бисмо их заштитили од недозвољених измјена, као и да бисмо имали све на једном мјесту. Пошто нам служи само за добављање података, закључујемо да ће бити у питању ***data class***, а како до сада нисмо имали промјењиве стања везане за обавјештења о подсјетницима, додаћемо их све сада у ову нову класу:

```
data class HomeScreenUiState
(
    val isInitialized : Boolean = false,
    val isChannelsListVisible : Boolean = false,
    val isEpgMenuDisplayed : Boolean = false,
    val isReminderVisible : Boolean = false,

    val channelsList : List<TvChannel> = listOf(),
    val numberOfChannels : Int = 0,
    val activeChannel : TvChannel? = null, // channel playing on the main screen

    val selectedEpgChannel : TvChannel? = null, // channel whose Epg is currently displayed

    val selectedProgramme : TvProgramme? = null, // selected programme in the displayed Epg list
    val firstScheduledProgramme : MutableList<TvProgramme?> = null,

    // making reminder notifications
    val reminderData : ReminderData? = null,
    val reminderNotificationDisplay : Int = 0,
)
```

Слика 4.16 Списак промјењивих стања у *HomeScreenUiState*

Назив *UiState* означава управо то да све ове промјењиве стања утичу само на начин приказивања корисничке спреге, описан у програмској логици *ViewModel*-а. Пошто смо већину ових промјењивих већ користили и спомињали раније, укратко ћемо појаснити само намјену оних које су додате сада.

- firstScheduledProgramme – овдје чувамо емисију која је тренутно прва на реду за приказивање обавјештења (тј. почиње раније од свих осталих), разлог због којег је Листа је што можемо имати више емисија које почињу истовремено
- reminderData – овдје се чувају подаци о првом подсјетнику који је на реду за приказ обавјештења, у претходној промјењивој смо чували емисије, те је овдје довољно чувати само вријеме њиховог почетка и краја, без Листе
- reminderNotificationDisplay – промјењива која прати број емисија које ће бити приказане у обавјештењу за подсјетнике

- isInitialized – промјенљива коју ћемо касније користити за приказивање оличја апликације и заштиту од пуцања

Након убацивања *UiState*–а у *ViewModel*, потребно је све досадашње промјенљиве стања у **HomeScreen**-у као и његовим подфункцијама замијенити са онима из *UiState*-а, обезбиједивши да сви користе јединствен и исти објекат те класе. Управо због тога ћемо га додати у *ViewModel*, а затим покупити из *ViewModel*-а у **HomeScreen** и приступати му на сљедећи начин:

```
val viewModel : HomeScreenViewModel = ViewModelHomeScreen()
val uiState by viewModel.uiState.collectAsState()
if (uiState.isEpgMenuDisplayed) { ... } // uiState use example
```

Више о начину употребе биће разјашњено у одјељку о *ViewModel*–у преко ког ће се *UiState* даље прослеђивати ка корисничкој спреги.

B)HomeScreenViewModel класа

Ова класа представља не само главну спону, него и срж наше апликације, у њој ће се одвијати не само главна програмска логика наше апликације него и водити брига о промјенљивима стања, слати и примати захтјеви ка и од ***БП*** и *Anoki* послужиоца итд. Започињемо тако што најприје направимо све методе побројане у **HomeScreenAction**-у, а то постижемо на сљедећи начин:

```
fun onAction(action : HomeScreenAction) {
    when (action) {
        is HomeScreenAction.OnChannelClick -> { onChannelClick(action.channelID }
        is HomeScreenAction.OnScreenClick -> { onScreenClick() }
        is HomeScreenAction.OnEpgMenuArrowClick -> onEpgMenuArrowClick()
        is HomeScreenAction.OnEpgItemClick -> onEpgItemClick(action.programmeID)
        is HomeScreenAction.OnProgrammeReminderClick -> onProgrammeReminderClick(action.programmeID)
        is HomeScreenAction.OnReminderConfirmClick -> onReminderConfirmClick(action.channelID)
        is HomeScreenAction.OnReminderDismissClick -> onReminderDismissClick()
        // else -> {}
    }
}
```

Слика 4.17 Преклапање апстрактних *HomeScreenAction* функција у методе *HomeScreenViewModel*-а

Иако се наизглед чини сложеним, ово преклапање тј. наредба *when-is* није ништа друго до *Kotlin*-ов напреднији начин *if-else if-else* петље (иако можда много више личи на *switch*), његова напредност се огледа у томе што ће *Kotlin*-ов Интерпретер кода водити рачуна о покривености свих случајева прије него што уопште дође до компајлирања апликације и напомињати нас о изостављеним случајевима. Све ове функције су већ описане и објашњене у ранијим корацима те се на њих нећемо освртати осим у случају неких значајнијих измјена у наредним корацима.

Према ***MVVM*** обрасцу, *UiState* се користи на сљедећи начин унутар *ViewModel*–а:

```
    private val _uiState = MutableStateFlow(HomeScreenUiState());
    val uiState: StateFlow<HomeScreenUiState> = _uiState.asStateFlow();
```

Посљедњи представља копију првог која се не може измијенити и служи за прослеђивање унутар корисничке спреге (на начин приказан на претходној страници), док је први доступан за коришћење само унутар *ViewModel*–а. Разлог прављења објекта на овај начин је обезбјеђивање да на нивоу наше апликације постоји само један једини објекат типа *HomeScreenUiState* (*eng*

Singleton) јер имамо само једну врсту главног екрана. Ограничење које нам ово доноси је то да не можемо изравно мијењати један по један атрибут овог објекта, него је потребно приликом освјежавање вриједности било којег од атрибута то учинити помоћу посебног приватног конструктора копије:

```
_uiState.update {
                it.copy(
                    activeChannel = _uiState.value.channelsList[0],
                    isInitialized = true )
}
```

Дакле, измјена је могућа једино тако што за сваку промјенљиву прослиједимо нову вриједност стања преко Методе *update*, а њен посебан приватни конструктор копије нам омогућава направити објекат који ће имати исте вриједности свих атрибута осим оних које желимо промијенити.

Како бисмо све ово припремили за коришћење потребно је попунити одгаварајуће вриједности у промјенљиве стања прављењем *uiState* објекта, као и добавити податке о ***TV*** каналима и њиховим емисијама, прије него што се започне исцртавање главне странице (**`HomeScreen`**) која ће их користити. Како се објекат **`HomeScreenViewModel`**-а прави већ на самом почетку главне странице (**`HomeScreen`**) неопходно је у том кораку обавити све наведено. За то нам служи **`init`** метода. За сада она ће само добављати наше измишљене и унапријед предодријећене (*eng Mockup*) податке, али ћемо је допуњавати у наредним корацима израде апликације.

Како је добављање и слање података у **`init`** методи, сложенији и захтјевнији поступак, неопходно је издвојити га из **`HomeScreenViewModel`**-а у посебну врсту класа које се зову *Repositories*, јер служе складиштењу података. Према ***MVVM*** обрасцу, такве класе се (ради повећања независности од платформе на којој се ради) у слоју ***domain***-а праве као Интерфејси са апстрактним методама које је потребно преклопити у слоју за складиштење правих података (***data***) како би се могле лакше прилагођавати потребама наше апликације простом имплементацијом Интерфејса и преклапањем његових метода.

Г)ChannelRepository класа

Изглед ове класе, као и сваке друге за обраду и складиштење података (у овом случају података о ***TV*** каналима) се ни у чему суштински не разликује од *Repository* класа из *RESTful Web* апликација. Једина разлика је у томе што је у домену наше апликације потребно направити Интерфејс ради повећања прилагодљивости апликације различитим платформама и ***API***-јима. Ове класе садрже Методе ***CRUD*** операција над подацима. За сада ова класа ће обрађивати само наше измишљене податке, док не оспособимо повезивање са *Anoki* послужиоцем и базом података, а изгледаће овако:

```
interface ChannelRepositoryInterface
{
    suspend fun mockUpFillChannels();

    suspend fun getChannelsListSize(): Int;
    suspend fun getAllChannels(): List<TvChannel>;
    suspend fun getChannelEPG(channel: TvChannel): List<TvProgramme>?;

    suspend fun findChannelById(id: String): TvChannel?;

    suspend fun saveChannel(channel: TvChannel): Boolean;
    suspend fun updateChannel(channel: TvChannel): Boolean;

    suspend fun removeChannel(channel: TvChannel): Boolean;
    suspend fun removeChannel(id: Int): Boolean;

    fun findProgrammeById( channel : TvChannel, id: String): TvProgramme?
    fun findProgrammeById(id: String): TvProgramme?;
    fun findProgrammesByStartTime(startTime : ZonedDateTime): MutableList<TvProgramme>;
    fun findProgrammesByStartTime(startTime : Long): MutableList<TvProgramme>;

    fun getAllProgrammes(): List<TvProgramme>;
    fun getChannelId(programme: TvProgramme): Int;

    fun saveProgramme(programme: TvProgramme): Boolean;
    fun updateProgramme(programme: TvProgramme): Boolean;

    fun removeProgramme(programme: TvProgramme): Boolean;
    fun removeProgramme(id: String): Boolean;
    fun fillProgrammes(channelsList: MutableList<TvChannel>)
}
```

Слика 4.18 Потписи метода *ChannelRepository* класе

Како будемо додавали приступ ***БП*** и *Anoki* послужиоцу тако ћемо и по потреби допуњавати ову класу новим Методама и мијењати њихов начин рада.

4.3 Уклапање у DaggerHilt и повезивање са послужиоцем

Успјели смо направити апликацију која ради са нашим измишљеним подацима, али како направити апликацију која може примити праве податке о ***TV*** каналима са послужиоца и обрађивати их?

За то нам је потребно најприје омогућити приступ интернету, како хардверски, повезивањем на мрежу, тако и софтверски, додавањем могућности унутар **`build.gradle`** датотеке (слика 4.2). Након тога потребно је знати гдје на интернету (тј. на ком ***URL***-у) се налази послужилац којем желимо приступити, каквог облика мора бити захтјев и како га послати. Све ово је могуће урадити и ручно, али би то беспотребно и умногоме усложњило наш код, те одузело нам много времена. Како не бисмо цијели овај поступак, описан подробније у претходном поглављу, радили ручно изнова и изнова, потребно нам је неко средство које ће то само чинити за нас када је потребно. Како бисмо то обезбиједили потребне су нам двије ствари:

1) ***DaggerHilt*** као средство које ће управљати током извршавања апликације и благовремено обављати позиве ка послужиоцу како бисмо увијек имали доступне најновије податке

2) ***RETROFIT*** као окружење које ће се побринути за прављење, слање захтјева и добављање одговора са послужиоца

Најприје је неопходно омогућити коришћење ***DaggerHilt***–а. Да бисмо то учинили, потребно је унутар **`build.gradle (:app)`** датотеке за подешавање наше апликације дати дозволе за њихово коришћење, тако што ћемо унијети сљедеће линије кода:

```
    //DaggerHilt
    implementation("com.google.dagger:hilt-android:2.46")
    kapt("com.google.dagger:hilt-android-compiler:2.46")

    kapt("androidx.hilt:hilt-compiler:1.2.0")
    implementation("androidx.hilt:hilt-navigation-compose:1.2.0")

    // ViewModel Dagger-Hilt lifecycle
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.1")
    // ViewModel utilities for Compose
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.1")
    // LiveData
    implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.8.1")
    Lifecycles only (without ViewModel or LiveData) //
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.8.1")
    Lifecycle utilities for Compose //
    implementation("androidx.lifecycle:lifecycle-runtime-compose:2.8.1")
    Saved state module for ViewModel //
    implementation("androidx.lifecycle:lifecycle-viewmodel-savedstate:2.8.1")
```

Слика 4.19 Додавање зависности за *DaggerHilt* и *ViewModel* у build.gradle

Како сада управљање радом апликације препуштамо ***DaggerHilt***-у, потребно је то назначитии и *Kotlin* Компајлеру.

Први корак је направити класу која ће представљати нову тачку покретања са назнаком да је у питању ***DaggerHilt***-управљана апликација. Ова класа најчешће носи исто име као и апликација:

```
@HiltAndroidApp
class LiveTvEpgApp : Application()
```

Такође, потребно је у **`AndroidManifest.xml`** унутар <application> ознаке навести да је ово нова тачка покретања додавањем назива на сљедећи начин:

```
        <application
        android:name=".LiveTvEpgApp"
        ... > ... </application>
```

Пошто је нова тачка покретања апликације ***DaggerHilt***, потребно је главну класу (**`MainActivity.kt`**) описати ***DaggerHilt***–у као мјесто од којег ће покретати апликацију те:

```
class MainActivity: ComponentActivity()
```

мијењамо у

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity()
```

Трећи корак је пребацивање свих класа од којих наш *ViewModel* зависи из његових атрибута у његов конструктор ***DaggerHilt***-овом техником убризгавања зависности (eng *Dependency Injection*). Пошто се тип који убризгавамо одрјеђује тек приликом извршавања апликације, можемо у *ViewModel* конструктор убацити Интерфејс и омогућити да наша апликација ради за било коју

класу која га имплементира, не зависећи од било које конкретне врсте класе која га имплементира. Такође овиме јамчимо и њихову јединственост на нивоу апликације (*eng Singleton*) јер ***DaggerHilt*** може прослиједити исту инстанцу објекта те класе и у случајевима када имамо више од једне инстанце *ViewModel*-а, чиме не само да смањујемо утрошак радне меморије него и повећавамо вјеродостојност и ажурност вриједности које се у њима налазе, јер сви дијелови апликације читају и пишу на исто мјесто. Због тога:

```
class HomeScreenViewModel: ViewModel()
{   private val channelRepository: ChannelRepository
  ...
}
```

мијењамо у:

```
@HiltViewModel
class HomeScreenViewModel @Inject constructor(
    private val channelRepository: ChannelRepositoryInterface
) : ViewModel()
{ ... }
```

такође, у **`HomeScreen`** мијењамо инстанцу ***`HomeScreenVeiwModel`***-а у:

```
val viewModel: HomeScreenViewModel = hiltViewModel()
```

Врло смислено питање које се поставља је; ако у конструктор прослеђујемо само Интерфејс, а имамо 2 или више класа које га насљеђују, како ***DaggerHilt*** може знати коју од њих узети, или гдје ће уопште наћи класу која насљеђује тај Интерфејс?

У ту сврху користимо Модуле, посебне врсте класа са јединстеном инстанцом које се у *Kotlin*-у називају *object module* (суштински су исто што и *Singleton*). Ове класе служе за прослеђивање објеката, класа или било које врсте података која није позната или одријељена до тренутка извршавања тј. позива неке функције у нашој апликацији тако што ће рећи ***DaggerHilt***–у: „ако ти је потребно то, овдје ћеш га пронаћи“. Ови Модули се најчешће праве тако да једна инстанца опслужује све захтјеве на нивоу цијеле апликације. Како бисмо пак назначили ***DaggerHilt***–у да нам је довољна само једна инстанца мораћемо ставити назнаке (*eng Annotation*) `SingletonComponent`. За до сада коришћену класу `ChannelRepository` то ће изгледати овако:

```
package com.example.livetvepgapp.di
@Module
@InstallIn(SingletonComponent::class)
object AppModule
{
    @Provides // dependency
    @Singleton
    fun provideChannelRepository(): ChannelRepositoryInterface
    {
        return ChannelRepository();
        // return ChannelRepositoryImpl();
    }
```

Слика 4.20 Прослеђивање класе *DaggerHilt*-у преко Модула

Према смијерницама ***MVVM***-а се Интерфејсима заправо најчешће дају имена која би требала имати класа (у овом случају `ChannelRepository`) док саме класе, које имплементирају те

Интерфејсе, добијају имена са наставком „Impl” (у овом случају `ChannelRepositoryImpl`), јер их само имплементирају, али ради лакшег разумијевања користићемо устаљене називе из других области. Модули се према ***MVVM*** обрасцу смијештају у засебан пакет ***di*** што се односи на „***D**ependency **I**njection*“ јер се управо у том тренутку и на том мјесту врши убризгавање параметара од којих зависи покретање дијелова наше апликације.

Исти поступак у овом кораку ћемо урадити и за сваку наредну (*Repository*) класу за податке (тј. Интерфејс који имплементирају) или ***БП*** када их будемо додавали – додаћемо их као параметре у убризганом конструктору ***DaggerHilt***-а за наш **`HomeScreenViewModel`**, а затим у зависности од њихове намјене направити функцију (и по потреби Модул) који ће рећи ***DaggerHilt***-у гдје може пронаћи то што му је потребно. Предност овога је очевидно у томе да можемо понудити и више од једне класе уколико нека од њих није доступна или одговарајућа.

А сада, када смо пребацили управљање апликацијом на ***DaggerHilt*** можемо прићи на убацивање *RETROFIT*-а и добављање података са *Anoki* послужиоца.

Да бисмо послали захтјев ка *Anoki* послужиоцу, требамо се присјетити свега што нам је за то потребно (`advertismentId, anokiUserId, publicIpAdress`) и како га добавити (описано у поглављу 3.2).

За добављање `advertismentId` је потребно приступити метаподацима наше апликације, и пронаћи га међу њима. Најприје нам је потребна *Repository* класа која ће га добављати, а након добављања је потребно ***DaggerHilt***-у објаснити када је потребан и гдје га убацити. Поред већ постојећег **`ChannelRepository`** направићемо и **`AdvertismentIdRepository`** на потпуно исти начин (Интерфејс у ***domain*** слоју, а затим класу у ***data*** слоју која га имплементира). Пошто је тражење `advertismentId`–а ствар која изискује доста програмског времена, неопходно је назначити *Kotlin*-у да је покреће искључиво у Корутинама других нити како не бисмо успорили рад наше апликације. Стога додајемо кључну ријеч `suspend` у потпис методе за добављање `advertismentId`–а те наша класа изгледа овако:

```
class MiscellaneousDataRepository @Inject constructor(
    private val app: Application
): MiscellaneousDataInterface
{
    override suspend fun getAdvertisingId(): String? {
        try {
            val adInfo = AdvertisingIdClient.getAdvertisingIdInfo(app.applicationContext)
            Log.d(TAG, "Advertisment ID fetched! Value: ${adInfo.id}");
            return adInfo.id
        } catch (e : Exception) {
            e.printStackTrace()
            return null
        }
    }
}
```

Слика 4.21 Добављање `advertismentId` -а помоћу ***DaggerHilt***-а

Примјећујемо да ова класа прима параметар врсте ***Application*** који је неопходан за приступање метаподацима (*applicationContext*) наше апликације. Поставља се питање, гдје ћемо га пронаћи? Сви подаци о апликацији се налазе иначе унутар **`MainActivity`** класе, те је потребно добавити их одатле. Јасно увиђамо да наша *Repository* класа нема никакву везу са **`MainActivity`** класом, те како бисмо добавили податке о апликацији, неопходно је прослиједити их у **`HomeScreen`**, а затим одатле, у **`HomeScreenViewModel`** одакле бисмо их преузели у нашој *Repository* класи.

Како не бисмо ово радили приликом сваког позива сличних Метода, и олакшали развој, ово можемо такође препустити ***DaggerHilt***-у на управљање. Преко ***DI*** обрасца, назначићемо ***DaggerHilt***-у, да он сам добави ***Application*** и њене метаподатке (*applicationContext*) наше апликације. Како се они сада налазе у ***DaggerHilt***-овој **`LiveTvEpgApp`** класи, он већ унапријед зна гдје се они налазе и како их добавити, није потребно правити посебну Методу унутар Модула (*object module*) која ће се старати о томе.

Сљедеће на реду је, добављање `anokiUserId`–а за шта нам је потребан и претходно добављени `advertismentId`. Дакле, потребно је и унутар *Repository* класе за `anokiUserId` добавити метаподатке апликације и поновити цијели поступак од раније. Међутим, ***DaggerHilt*** нам омогућава обрасцем убризгавања ових предуслова и зависности, прослиједити их као параметре Методе, а он ће сам обезбиједити све што је потребно.

С тим у вези, додајемо сљедеће у **`AppModule`**:

```
@Provides
@Singleton
fun provideAdvertismentIdRepository(app: Application): AdvertismentIdInterface
{
    return AdvertismentIdRepository(app);
    // return AdvertismentIdRepositoryImpl(app);
}
```

Сада, када знамо како поново добавити `advertismentId`, можемо послати захтјев ка *Anoki* послужиоцу те добавити `anokiUserId`. За то нам је потребно у *Anoki* упутствима коришћења (*eng documentation*) пронаћи ***URL*** и ***API*** за приступ послужиоцу те послати *RETROFIT* захтјев, који ће изгледати овако:

```
@Provides //dependency
@Singleton
fun provideAnokiUserApi(): AnokiUserApi
{
    val baseUrl = "**API_base_url_placeholder**"
    return Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
        .create(AnokiUserApi::class.java);
}
```

Слика 4.22 Слање *RETROFIT* захтјева за *AnokiUserId API*

Када смо добавили тај ***API*** потребно је преузети његову Методу која нам је потребна на сљедећи начин (по упутствима):

```
interface AnokiUserApi {

    @GET("/**API_Method_URL_suffix**")
    suspend fun getUserId(
        @Query("deviceId") deviceId : String,
    ): Response<ResponseBody>
}

data class AnokiUserResponse(@SerializedName("userId") val userId : String)
```

Слика 4.23 Преузимање методе за добављање *AnokiUserId* са *AnokiAPI*-ја

Пошто желимо добавити податке, *RETROFIT*-у ћемо преко назнаке `@GET` послати *GET* захтјев са одговарајућим ***URL***-ом из ***API***-ја. А назнаком `@Query` које параметре прослеђујемо. Међутим, како ова метода не враћа само `anokiUserId` (у очекиваном *String* облику), него и још гомилу додатних ствари, неопходно је издвојити га из одговора који добијемо од ***API***-ја. То ћемо учинити приликом преузимања података из *Repository* класе. Пошто је ријеч о позиву ка послужиоцу, потребно је обликовати га као *RETROFIT* захтјев, али без икаквог ***URL***-а, како не бисмо погодили погрешну адресу. Пошто суштина није научити посебности рада са *Anoki* сервисима, као и због заштите начина пословања, приказаћемо само потпис методе из Интерфејса:

```
interface AnokiUserInterface
{
    @GET("/")
    suspend fun fetchUserId(
        @Query("deviceid") deviceId : String,
    ): String?
}
```

Слика 4.24 Добављање *AnokiUserId* из наше *Repository* класе преко *RETROFIT* позива

Када смо коначно добили наш `anokiUserId` потребно је убацити га у нашу апликацију, дакле, убризгати као параметар наш Интерфејс **`AnokiUserInterface`** који га добавља, а затим објаснити ***DaggerHilt***-у како га пронаћи унутар **`AppModule`** Модула:

```
    @Provides
    @Singleton
    fun provideAnokiUserRepository(//provideAnokiChannelRepository
        api : AnokiUserApi, //AnokiEpgApi
    ): AnokiUserInterface//AnokiChannelInterface
    {
        return AnokiUserRepository(api); // return AnokiChannelRepositoryImpl
    }
```

Исти поступак треба учинити и за сваки наредни захтјев ка послужиоцу (са мањим разликама), но пошто је овај дио потпуно исти, у опаскама (*eng comments*) са стране смо навели како ће изгледати иста метода за добављање података о ***TV*** каналима.

Након добављеног `anokiUserId`–а преостаје нам још добавити јавну ***IP*** адресу наше мреже како бисмо могли послати захтјев за добављање ***TV*** канала. Уколико не знамо напамет јавну ***IP*** адресу наше мреже, то постаје сложенији поступак од претходна два за који нам је потребан још један ***API***. Међутим, овај ***API*** је прављен од стране *Google*-а и у склопу је *RETROFIT*-а, те је јавно доступан свима. ***API*** којем желимо приступити је *https://api.ipify.org/* и изгледа овако:

```
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {

    @Provides
    @Singleton
    fun provideIP(): Retrofit {
        val ipifyBaseUrl = "https://api.ipify.org/"
        return Retrofit.Builder()
            .baseUrl(ipifyBaseUrl)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
}
```

Слика 4.25 Слање захтјева за IP API преко *RETROFIT* позива

За ову намјену смо направили посебан ***DaggerHilt*** Модул под називом **`NetworkModule`** ради побољшања прегледности. Како овај ***API*** враћа одговоре у облику низа 0 и 1 (тзв. *Bytestream*) неопходно је превести га у облик читљив за нашу апликацију. Пошто *RETROFIT* такође има и ***API*** за издвајање података о ***IP***-у у ***JSON*** облику, додаћемо `GsonConverterFactory` који ће га претворити у ***JSON***. Међутим, за приступ тим подацима нам је потребан још један позив ***API***-ја који ће ***JSON*** објекат записати у његов *String* облик те ће он најзад бити читљив нашој апликацији:

```
interface IpServiceApi {
    @GET("https://api.ipify.org?format=json")
    suspend fun getPublicIp(): Response<IpResponse>
}

data class IpResponse(
    @SerializedName("ip") val ip: String
)
```

Након тога, потребно је извући из ове гомиле ***IP*** метаподатака ***IP*** адресу у облику читљивом за нас и *Anoki* послужилац. За то ћемо направити *Repository* класу (и Интерфејс) који ће користити тај ***API***, а изгледаће овако:

```
class IpServiceRepository @Inject constructor(
    private val ipService: IpServiceApi
) : IpServiceInterface {
    private val _ip = MutableLiveData<String>()
    val ip: LiveData<String> = _ip
    override suspend fun fetchPublicIp(): String? {
        withContext(Dispatchers.Main) {
            try {
                val response = ipService.getPublicIp()
                if (response.isSuccessful) {
                    _ip.value = response.body()?.ip ?: ""
                }
                else _ip.value = "Error: ${response.code()}"
            }
            catch (e:Exception) {
                _ip.value = e.toString();
                e.printStackTrace();
            }
        }
        Log.d("TAG", "fetchPublicIp: ${_ip.value}")
        return _ip.value
    }
}
```

Слика 4.26 Добављање IP адресе из *RETROFIT JSON IP API*-ја

Као и увијек, на самоме крају, потребно показати ***DaggerHilt***–у како то пронаћи, додавањем у његов Модул (у овом случају **`NetworkModule`**), у који додајемо методу за ***API***:

```
    @Provides
    @Singleton
    fun provideIpService(retrofit: Retrofit): IpServiceApi {
        return retrofit.create(IpServiceApi::class.java)
    }
```

и методу за *Repository* класу:

```
    @Provides
    @Singleton
    fun provideIpServiceInterface(api : IpServiceApi): IpServiceInterface {
        return IpServiceRepository(api);
    }
```

Након убризгавања и ове *Repository* класе у конструктор нашег *ViewModel*-а (**`HomeScreenViewModel`**), сада најзад имамо све што нам је потребно за слање захтјева који ће добавити ***TV*** канале и све њихове податке са *Anoki* послужиоца. За то нам је пак потребан нови ***API*** јер је претходно кориштени служио само за добављање `anokiUserId`–а. Нови ***API*** ћемо назвати **`AnokiEpgApi`** и он ће изгледати овако:

```
interface AnokiEpgApi
{
    @GET("/*method_URL_suffix2*")
    suspend fun getChannelList(
        @Query("country") countryCode: String = "USA",
        @Query("auid") userId: String = "",
        @Query("deviceid") deviceId : String = "",
        @Query("ip") ip : String = "",
    ): Response<ArrayList<TvChannelAnokiDTO>>

    @GET("/*method_URL_suffix3*")
    suspend fun getProgrammeList(
        @Query("auid") anokiUserId : String,
        @Query("startEpoch") startEpoch : Long,
        @Query("endEpoch") endEpoch : Long,
        @Query("channelIds") channelIds : String,
        @Query("genre") genre : String,
        @Query("country") country: String,
    ) : Response<ArrayList<TvChannelProgrammesAnokiDTO>>

}
```

Слика 4.27 Потписи метода *API*-ја за добављање *TV* канала и емисија

У приложеном видимо да је потребно направити и засебне ***data class***-е за прихват података из одговора *Anoki* послужиоца, јер нам неће бити потребни сви подаци из њих, а неки подаци ће вјероватно недостајати, те је нужно направити и функције које ће преписивати неопходне податке из тих класа, **`TvChannelAnokiDTO`** и **`TvChannelProgrammesAnokiDTO`**, за пренос података (***D**ata **T**ransfer **O**bject*) у класе које користи наша апликација из ***domain.models*** пакета направљене раније. Те функције ће се налазити унутар раније поменуте *Repository* класе, која рукује са подацима о нашим ***TV*** каналима – **`ChannelRepository`**. Међутим, потребна нам је још једна *Repository* класа која ће податке из ***DTO*** класа прослиједити до *ViewModel*-а у којем ће се налазити и **`ChannelRepository`**. Стога правимо и **`AnokiChannelRepository`** класу која ће обрадити *RETROFIT Response* тип који добијемо као одговор од *Anoki* послужиоца и претворити га у Листу ***DTO*** објеката које можемо

обрадити унутар **`ChannelRepository`** методе *updateAllEPGs* о којој ћемо више говорити касније.

На крају је, као и увијек неопходно објаснити ***DaggerHilt***–у гдје и како може пронаћи све нове *Repository* класе и ***API***-је које су додате у апликацију унутар Модула **`AppModule`** што ћемо учинити на сљедећи начин:

```
@Provides //dependency
@Singleton
fun provideEpgApi(okHttpClient: OkHttpClient): AnokiEpgApi
{
    val baseUrl = "**Anoki's channels API base URL**"
     return Retrofit.Builder()
            .baseUrl(BASE_URL_PROD)
            .addConverterFactory(GsonConverterFactory.create())
            .client(okHttpClient) // injects additional API-keys in request header before sending it to API
            .build()
            .create(AnokiEpgApi::class.java);
}

@Provides
@Singleton
fun provideOkHttpClient(headerInterceptor: RetrofitHeaderInterceptor) : OkHttpClient
{
    return OkHttpClient().newBuilder()
        .addInterceptor(headerInterceptor)
        .build()
}

@Provides
@Singleton
fun provideHeaderInterceptor() : RetrofitHeaderInterceptor
{
    return RetrofitHeaderInterceptor()
}

@Provides
@Singleton
fun provideEpgChannelRepository(
    api : AnokiEpgApi,
): AnokiChannelInterface//AnokiChannelRepository
{
    return AnokiChannelRepository(api); // AnokiChannelRepositoryImpl
}
```

Слика 4.28 Убацивање *API*-ја и *Repository* класа за добављање канала у *DaggerHilt*

Као што видимо, овај захтјев је друкчији од претходног јер је уз њега било потребно послати и додатне ***API*** кључеве за *Anoki* послужилац. Како се не бавимо радом *Anoki* послужиоца, нећемо се освртати на те кључеве, али ћемо се осврнути на механизам кориштен за њихово слање. У питању је *Interceptor*, који нам омогућује пресрести ***HTTP*** захтјев у *RETROFIT*-у прије него што он оде на своје одредиште и допунити га, или у потпуности измијенити, по потреби. За то нам је свакако потребан и *OkHTTPClient* који ће датим измјенама из *Interceptor*-а преклопити претходни захтјев. Сам поступак пресретања је врло једноставан састоји се у прављењу наше класе која ће наслиједити *Interceptor* Интерфејс и преклопити његову Методу *Intercept*. У њој је потребно направити потпуно нов захтјев, који ће затим бити прослијеђен *OkHTTPClient*–у и прегазити стари прије него што он буде послат. Прављење новог захтјева у *Interceptor*-у се врши ручно, додавањем сваког његовог дијела и изгледа овако:

```
class RetrofitHeaderInterceptor : Interceptor {
    override fun intercept(chain:Interceptor.Chain) : Response {
        val originalRequest = chain.request()
        val newRequest = originalRequest.newBuilder()
            .header(KEY_HEADER, PRODUCTION_KEY)
            .build();
        return chain.proceed(newRequest);
    }
}
```

Слика 4.29 Пресретач Retrofit HTTP захтјева

Након убризгавања и ове **`AnokiChannelRepository`** класе (тј. њеног Интерфејса) у конструктор *ViewModel*-а и преписивања добављених података у наше класе преко **`ChannelRepository`** у *ViewModel*-у можемо најзад замијенити све измишљене (*mockup*) податке које смо до сада користили за исцртавање апликације и замијенити их правим. Како смо и раније све чували у објектима наших класа, веће измјене неће бити неопходне и све ће се свести на замјену `Image` у `AsyncImage` компоненте јер су слике оличја ***TV*** канала (*eng. logos*) сада представљене повезницама (*eng. links*) на интернету умјесто кључевима елемената додатих одјељак *ResourceManager*-а (*Drawables*) наше апликације.

4.4 Додавање Базе Података и механизма за подсјетнике

Како сада најзад користимо и добављамо стварне податке, можемо кренути у праву израду подсјетника и обавјештења. У ту сврху начинићемо методу *`makeReminderTimer`* која ће се позивати приликом заказивања подсјетника у *`onProgrammeReminderClick`* Методи.

Међутим, податке о подсјетницима потребно и негдје дуготрајно чувати, како не би били обрисани након сваког гашења апликације, јер тиме било који подсјетник за каснију емисију губи смисао када се апликација угаси и поново покрене. С тим у вези, потребно је најприје направити ***БП*** која ће бити задужена за трајно чување подсјетника, а затим све подсјетнике учитавати приликом сваког новог покретања апликације. За то ћемо користити *ROOM library* окружење које је направљено од стране *Google*-овог *Android* тима. *ROOM* нам омогућава коришћење *SQLite* базе података на једноставнији и безбједнији начин пресликавајући ***data class***–е у релационе табеле *Object Relational Mapping* механизмом (***ORM***). Он олакшава прављење и рад са ***БП*** тако што се брине о врстама веза између података као и вјеродостојности и исправности података у ***БП***.

Како бисмо могли користити *ROOM* неопходно је најприје додати сљедеће зависности у **`build.gradle`** датотеку:

```
    //Room dependencies - version 2.6.1
    implementation("androidx.room::room-gradle-plugin:2.6.1")
    kapt("androidx.room::room-compiler:2.6.1")
```

Слика 4.30 Додавање зависности за *ROOM* у *build.gradle*

Након тога, можемо започети са прављењем ***БП*** наше апликације која ће бити смјештена у ***database*** Пакету унутар коријенског директоријума наше апликације. Све неопходно за прављење и рад са ***БП*** смијештамо у Пакет ***local*** унутар ***data*** слоја наше апликације Како бисмо започели потребно је прво направити тзв. *Entity* класу која представља обичну *Kotlin* ***data class***-у од које ће бити направљена табела у ***БП***:

```
@Entity
data class ReminderData
(
    @PrimaryKey(autoGenerate = false)
    val programmeId : String,    //id = channelId + ":" + contentId + " " + startTime
    val programmeStartTime: Long, //ZonedDateTime converted to Epoch format
    val programmeEndTime: Long, //ZonedDateTime converted to Epoch format
)
```

Слика 4.31 *Entity* класа за *ROOM* Базу Података

Entity класу одликују назнака *@Entity* која говори *ROOM*-у да је потребно направити од ње табелу (назив табеле у ***БП*** можемо навести у загради) у ***БП*** и назнака *@PrimaryKey* која говори да је у питању кључ табеле, који ће у овом случају бити ручно прављен и унапријед одријељен за сваки слог који се уноси.

За слање упита и добављање података из *ROOM* ***БП*** тј. њених табела користи се посебна инстерфејса која се назива ***DAO*** приступним објектом за податке (***D****ata* ***A****ccess* ***O****bject*).

```
@Dao
interface ReminderDataDao
{
    @Upsert //@Insert + @Update
    suspend fun insertReminderData(reminder: ReminderData);

    @Delete
    suspend fun deleteReminderData(reminder: ReminderData);

    @Query("DELETE FROM ReminderData WHERE programmeStartTime = :currentTime")
    suspend fun deleteRemindersByDate(currentTime: Long)

    @Query("DELETE FROM ReminderData")
    suspend fun clearRemindersData();

    @Query("DELETE FROM ReminderData WHERE programmeEndTime < :currentTime")
    suspend fun deleteOutdatedReminders(currentTime: Long);

    @Query("SELECT * FROM ReminderData ORDER BY programmeStartTime ASC")
    suspend fun getAllReminders() : List<ReminderData>

    @Query("SELECT * FROM ReminderData WHERE programmeStartTime > :currentTime ORDER BY programmeStartTime LIMIT 1")
    suspend fun getFirstReminder(currentTime: Long) : ReminderData?
}
```

Слика 4.32 *DAO* Интерфејс за упите над *ROOM* Базом Података

Препоручено је да ***DAO*** увијек носи име по табели на коју се односи, а поред већ описаних *@Query* и познатих *@Insert @Update* и *@Delete* назнака постоји и *@Upsert* која ће увијек додати нови слог у ***БП*** с тим што, уколико већ постоји слог са истим кључем измијениће његова остала поља умјесто додавања новог слога. Битно је напоменути да сви упити ка ***БП*** изискују одријељено вријеме те је неопходно да све методе буду извршаване ван главне нити, унутар неке Корутине, те

је кључна ријеч *suspend* обавезна.

Након направљених табела (преко *Entity* класе) и упита (преко ***DAO*** Интерфејса) преостаје нам још израдити ***abstract class***-у која ће представљати нашу ***БП***:

```
@Database(
    entities = [ReminderData::class],
    version = 1,
)
abstract class LiveTvEpgDatabase: RoomDatabase() {
    abstract val reminderDao : ReminderDataDao
}
```

Слика 4.33 Класа која представља *ROOM* Базу Података

Унутар *@Database* је потребно у пољу *entities* навести од којих све класа (као табела) ће се ***БП*** састојати, друго поље нам говори колико пута су вршене измјене над ***БП***. Сваки пут када додамо нову табелу, поље или везу између табела, неопходно је повећати број *version* поља. Као што видимо свака класа која представља ***БП*** мора наслиједити *RoomDatabase* и садржати у себи као поља све ***DAO*** објекте. На крају, преостаје нам убацити ***БП*** у нашу апликацију тако што ћемо објаснити ***DaggerHilt***–у (унутар **`AppModule`**) гдје је може пронаћи:

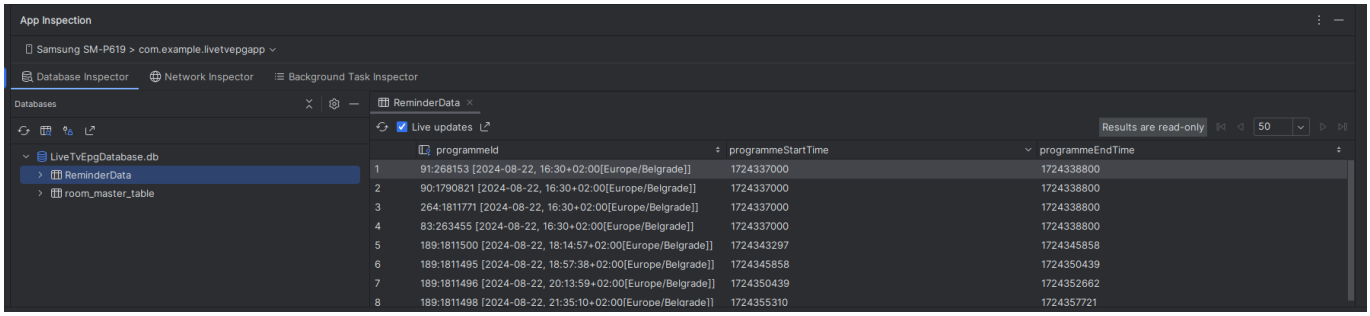
```
    @Singleton
    @Provides
    fun provideAppDatabase(@ApplicationContext appContext: Context): LiveTvEpgDatabase {
        Log.d("AppModule", "Creating database at request!");
        val database = Room.databaseBuilder(appContext, LiveTvEpgDatabase::class.java, "LiveTvEpgDatabase.db")
            .build()
        return database
    }
```

Слика 4.34 Добављање *ROOM* Базе Података преко *DaggerHilt*-а

Посљедњи корак је додавање у Конструктор *ViewModel*-а који ће је убризгати као зависност, јер ћемо у *ViewModel*-у позивати упите над ***БП*** током корисникових покрета унутар апликације.

Ово је много лакши начин за додавање ***БП*** у нашу апликацију, јер је истовјетан за све што смо до сада користили (*Repository* класе, позиве ка послужиоцу, ***API***-је,...), а замијенио је прављења бројних Интерфејса који би представљали догађаје како из *ViewModel*-а тако и корисничке спреге у *Jetpack Compose* окружењу.

Уколико желимо провјерити исправност рада ***БП*** то можемо учинити отварајући у *Android Studio*-у *App Inspector* → *Database Inspector* и пратити измјене уживо.



Слика 4.35 Изглед Базе Података у *Database Inspector*-у

Механизам за управљање подсјетницима (*CountDownTimer*)

Сада нам преостаје направити механизам који ће пратити тренутно вријеме и приказивати обавјештења о емисијама са заказаним подсјетницима пред њихов почетак. Метода *makeReminderTimer* је кључна за руковање подсјетницима у апликацији и задужена је за стварање и управљање *CountDownTimer*-ом, механизмом који одбројава вријеме до почетка сљедећег ***TV*** програма за који је постављен подсјетник.

Прије позива Методе, при самом покретању апликације потребно је начинити неколико припрема:

```
private val reminderAlarmsMap : MutableMap<Long, MutableList<ReminderData>> = mutableMapOf()
private var notificationTimer : CountDownTimer? = null;
private var notificationEpoch: Long = 99999999999;
```

Мапа `reminderAlarmsMap` нам служи за смијештање свих направљених подсјетника, како оних добављених из ***БП***, тако и оних који су заказани приликом тренутног покретања апликације (раније). Њен кључ представља вријеме почетка емисије, док вриједност под тим кључем представља Листу подсјетника за све емисије које почињу у то (исто) вријеме. Промјењљива типа *CountDownTimer* тј. `notificationTimer` служи за смијештање и покретање механизма који ће, када му се додијеле вриједности, почети одбројавати вријеме потребно од тренутка у којем је покренут до тренутка када је потребно приказати обавјештење о заказаној емисији. `notificationEpoch` нам служи као помоћна промјењљива која прати вријеме почетка емисије (или више емисија) која је прва на реду (тј. почиње прије свих осталих), и требала би увијек бити смјештена унутар `notificationTimer`-а. За коришћење овога нам је неопходно у **`build.gradle`** додати:

```
// working with dates, time, epochs...
coreLibraryDesugaring("com.android.tools:desugar_jdk_libs:2.0.4")
```

Ријеч је о посебном облику бројног записивања времена ради лакшег и бржег поређења које урачунава и временске зоне о чему се може више прочитати на страницама у изворима.

Посљедњи корак припрема је учитавање свих подсјетника из ***БП*** у `reminderAlarmsMap`. Уколико су подсјетници у потпуности истекли (тј. и вријеме почетка и вријеме завршетка емисије је у прошлости) они се бришу и из ***БП*** и из Мапе за подсјетнике. Пошто је *makeReminderTimer* Метода исувише обимна и сложена, потрудићемо се само објаснити како и шта она ради.

Када смо завршили све наведене припреме, те добавили податке о ***TV*** каналима са послужиоца, Метода *makeReminderTimer* се покреће у 2 случаја:

1) Уколико се у подацима из ***БП*** учитаним у `reminderAlarmsMap` налазе само они подсјетници код којих је вријеме почетка емисије у прошлости, али емисије још увијек трају, Метода се неће покренути докле год корисник не закаже неку нову емисију

2) Уколико се у подацима из ***БП*** учитаним у налазе подсјетници чије вријеме почетка емисије је у будућности, Метода *makeReminderTimer* се покреће при покретању апликације.

Параметар који прослеђујемо како бисмо покренули *makeReminderTimer* Методу представља члан `reminderAlarmsMap` Мапе чији кључ има најмању вриједност (што у случају *Epoch* броја представља емисије које почињу прије свих осталих). Како бисмо били увјерени да ће се увијек прослиједити баш оне емисије које су прве на реду у `notificationEpoch` ћемо увијек смијештати најмањи број (тј. *Epoch* вријеме) које је у будућности из `reminderAlarmsMap` Мапе, те позивати Методу са:

```
    makeReminderTimer(reminderAlarmsMap[notificationEpoch]);
```

Након што се *makeReminderTimer* покрене, она ће се извршавати непрестано док год корисник не угаси апликацију, одбројавајући вријеме до почетка наредне заказане емисије, или у случају истека свих емисија, престаје са радом до поновног заказивања неке нове емисије. Пошто је вријеме извршавања ове Методе веома дуго (и траје у најкраћем од заказивања првог подсјетника до приказивања посљедњег) неопходно је измјестити је у засебну нит како не би зауставила рад апликације у потпуности – те је коришћење Корутина неопходно.

Механизам за одбројавање и приказивање подсјетника ћемо направити као нови објекат класе *CountDownTimer* на сљедећи начин:

```
notificationTimer = object : CountDownTimer(scheduleTimeSeconds*999, 999) {
                override fun onTick(millisUntilFinished: Long) {...}
                override fun onFinish() {...}
}
```

Слика 4.36 Прављење *CountDownTimer* објекта за одбројавање

Први параметар представља укупно трајање механизма до завршетка, изражен у милисекундама, након чега ће се позвати Метода *onFinish* из класе *CountDownTimer*, а други параметар говори након колико милисекунди ће се извршити 1 откуцај тј. Одбројавање, које ће позивати њену Методу *onTick*. Разлог из којег за изражавање времена у секундама користимо множилац 999 је повремено благо кашњење које се јавља усљед времена утрошеног на извршавање *onTick* Методе.

По облику Конструктора за *CountDownTimer* је очевидно зашто увијек морамо преклопити Методе *onTick* и *onFinish*. У нашем случају *onTick* Метода због већ раније описаног начина коришћења параметара *makeReminderTimer*-а и његових помоћних промјењивих овдје неће бити од нарочитог значаја. Она се може користити за брисање подсјетника оних емисија чије је приказивање завршено (тј. вријеме завршетка у прошлости) из ***БП*** и меморије, али пошто за проналажење подсјетника користимо Мапу ово нам неће бити неопходно. Већина претходно описане програмске логике ће се дешавати и унутар *onFinish* Методе.

Када дође вријеме за приказивање обавјештења о почетку емисије позива се *onFinish*

Метода у којој ћемо проћи кроз Листу свих подсјетника те добавивши све емисије са послужиоца пронаћи на које емисије се они односе, из којих ћемо извући на којем ***TV*** каналу (или каналима) се она(е) приказују. Истовремено, потребно је промијенити промјењиву стања `isReminderVisible` и прослиједити пронађене ***TV*** канал(е) са називом (тј. називима) емисија како би се могао исцртати **`ReminderDialog`**. Након приказивања, потребно је пронаћи наредни подсјетник, избацивањем старог из Мапе, те започети ново одбројавање (уколико имамо још подсјетника који нису истекли) поновним позивом *`makeReminderTimer`* Методе, чиме постижемо непрестано покретање и извршавање које смо спомињали на почетку. Уколико је `reminderAlarmsMap` испражњена *`makeReminderTimer`* престаје са радом јер је (посљедње) одбројавање завршено.

4.5 Заштита од грешака и пуцања, дорада апликације

По испуњењу свих задатих захтјева и могућности апликације, посљедњи корак у изради представља заштиту од могућих грешака или пуцања током извршавања и унапрјеђење постојећих могућности (уколико је потребно). Након бројних тестирања апликације, установљено је да се једина грешка јавља на самом покретању апликације. Уколико корисник приликом покретања кликне на екран, што отвара Листу ***TV*** канала, прије него што се садржај добави са послужиоца и прикаже, долази до пуцања апликације.

Како се поступак добављања података никако не може избјећи нити убрзати, једино могуће рјешење је онемогућити кориснику било какав утицај клика на апликацију док се она у потпуности не учита. То ћемо најлакше постићи тако што ћемо преко цијелог екрана привремено приказати нешто што ће се налазити изнад стварног приказа, те га склонити након што се апликација учита. Како је ова апликација надахнута *Anoki*-јевом стварном апликацијом за ***DTV*** пријемнике искористићемо њихову замисао оличја (*eng logo*) апликације као покретне слике (*eng animation*). Ријеч је о ***JSON*** датотеци коју ћемо убацити у ResourceManager на исти начин као што смо и убацивали наше слике у апликацију на почетку израде. Послије тога потребно је само исцртати је у главној **`HomeScreen`** Компоненти прије било чега другог, а затим је обрисати.

Само исцртавање покретних слика је врло сложен поступак, те нам је за то неопходно посебно окружење звано *Lottie*. Оно је убједљиво најпростије рјешење за исцртавање покретних слика у ***JSON*** облику, а како бисмо га користили потребно је у **`build.gradle`** додати сљедеће:

```
    // For Animated intro screen from a JSON file in Jetpack Compose
    implementation("com.airbnb.android:lottie-compose:6.0.0")
    //when not using Jetpack Compose: implementation("com.airbnb.android:lottie:3.4.0")
```

Све што је сада поребно учинити како бисмо исцртали оличје апликације приликом извршавања је додати сљедеће линије кода у **`HomeScreen`**:

```
...
val viewModel: HomeScreenViewModel = hiltViewModel()
val uiState by viewModel.uiState.collectAsState()
val lottieAnimation by rememberLottieComposition(spec = LottieCompositionSpec.RawRes(R.raw.intro_animation))
 if(uiState.channelsList==null || uiState.channelsList.isEmpty()) //uiState.isInitialized==false)
 {
    LottieAnimation(
          composition = lottieAnimation,
          modifier = Modifier.fillMaxSize(),
    );
 }
else { ... } //drawing main app display screen
```

Слика 4.37 Код за исцртавање *Lottie* покретне слике прије главног приказа

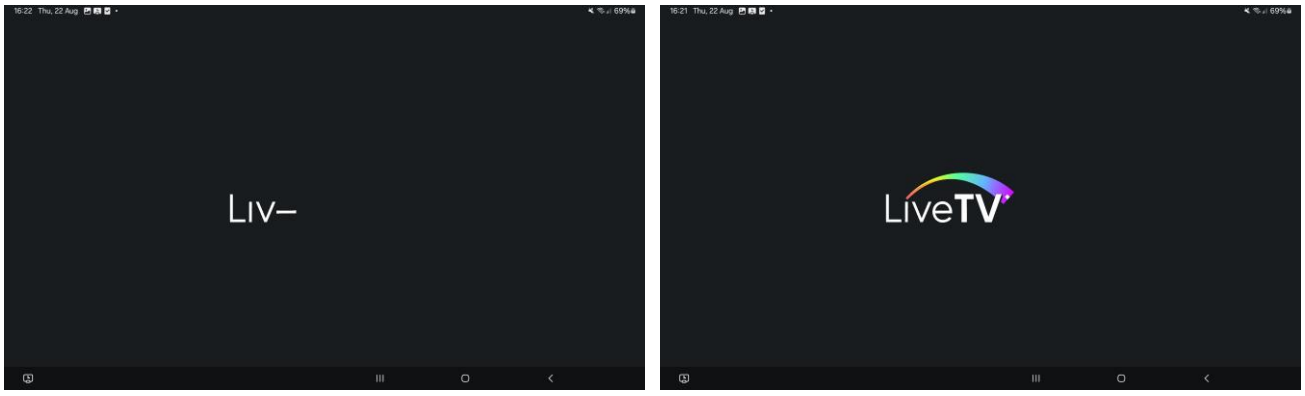
Чим се добаве подаци о ***TV*** каналима са послужиоца (након што ***DaggerHilt*** пошаље захтјев унутар Конструктора **`HomeScreenViewModel`**-а) промијениће се вриједност промјенљиве стања `isInitialized` (а ради и са обичним промјенљивим као нпр. Листа канала). Због свог начина рада, *Jetpack Compose*-а одмах поново покреће исцртавање цијелог **`HomeScreen`**-а те ће приликом тог (новог) исцртавања прескочити *Lottie* и приказати главни екран и остатак апликације. На овај начин смо обезбиједили да се наша апликација не може срушити и можемо прићи на сљедеће поглавље.

5. ИСПИТИВАЊЕ РАДА

Испитивање рада је вршено на Таблет уређају SAMSUNG SM-P619 и Xiaomi Redmi 9 телефону.

5.1 Покретање апликације

Начин рада апликације по корацима од њеног покретања до затварања је подробно објашњен у претходним поглављима, преостало је приказати крајњи изглед апликације корак по корак, редом како то изгледа за корисника:

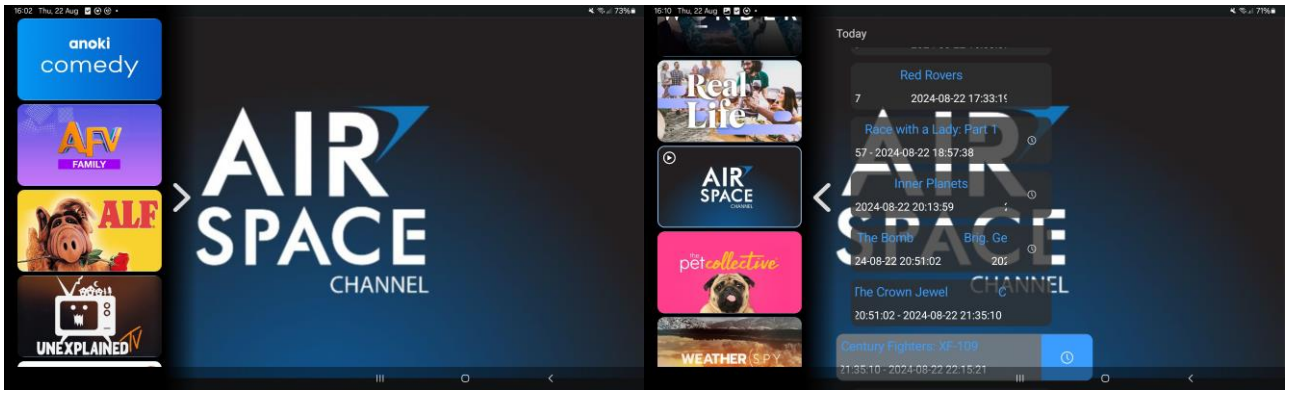


Слика 5.1 Исцртавање *Lottie* покретне слике прије главног приказа

Прво што видимо је покретна слика оличја апликације, израђена у кораку 4.5 са *Lottie*. По учитавању ***TV*** канала нам се отвара приказ једног од њих преко цијелог екрана.

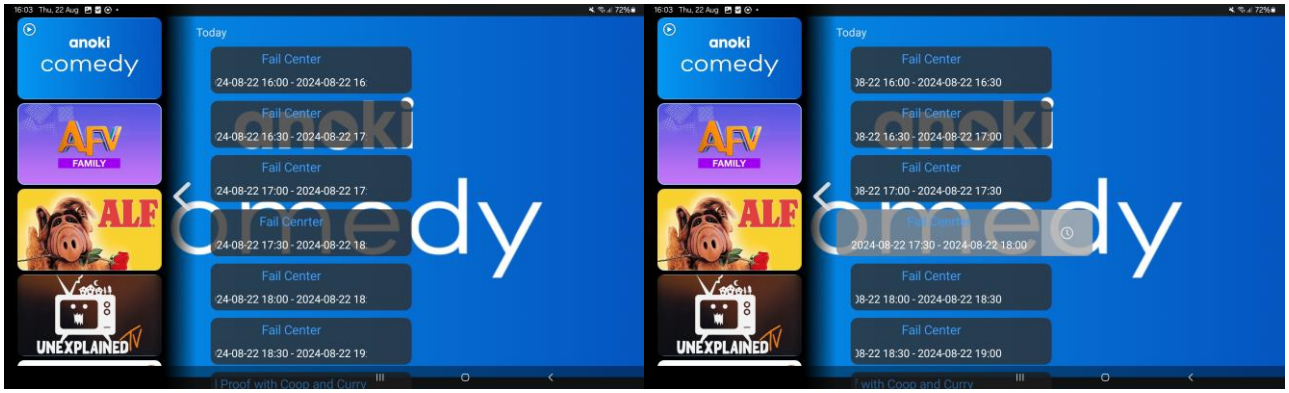
Слика 5.2 Изглед главног приказа након учитавања апликације

Кликом било гдје на екрану отвара нам се Листа канала у којој кликом на било који од чланова можемо промијенити ***TV*** канал који се тренутно приказује:



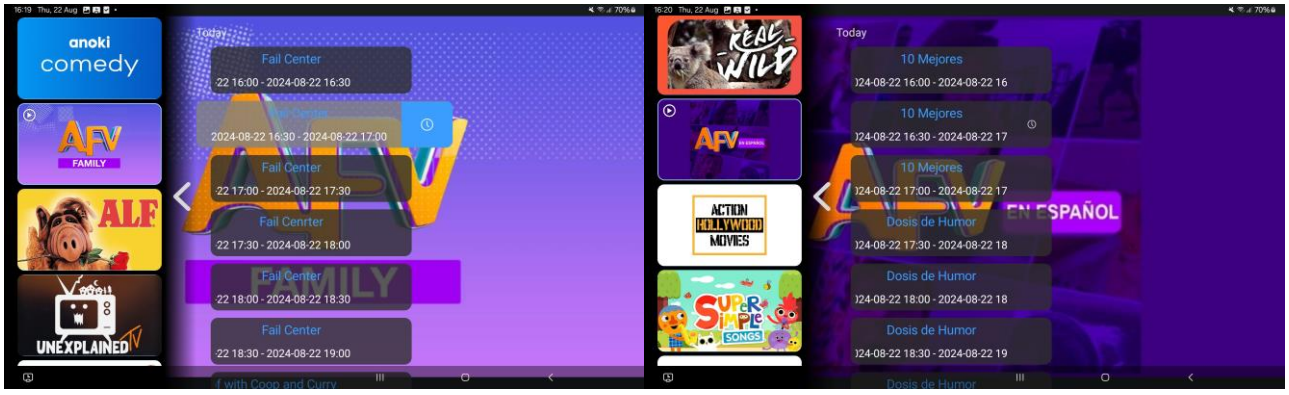
Слика 5.3 Приказ Листе канала и отварања програмске шеме

Отварањем Листе канала видимо и раније спомињану стрјелицу за приказ програмске шеме, као и изглед шеме са учитаним подсјетницима из ***БП***, уочавајући разлике између одабране емисије и неодабраних, као и емисија за које су заказани подсјетници (имају мали часовник) и оних који га немају (тј. немају заказан подсјетник).

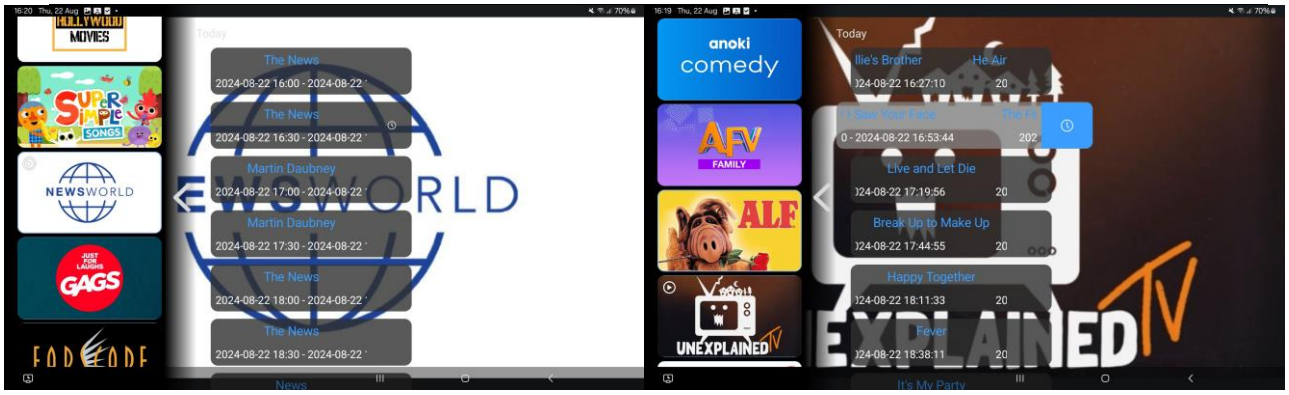


Слика 5.4 Први корак заказивања подсјетника

На овим сликама можемо видјети како шема изгледа када се тек отвори и прије него што има иједну емисију са заказаним подсјетником. Заказаћемо неколико емисија које почињу у исто вријеме (нпр. 16:30):

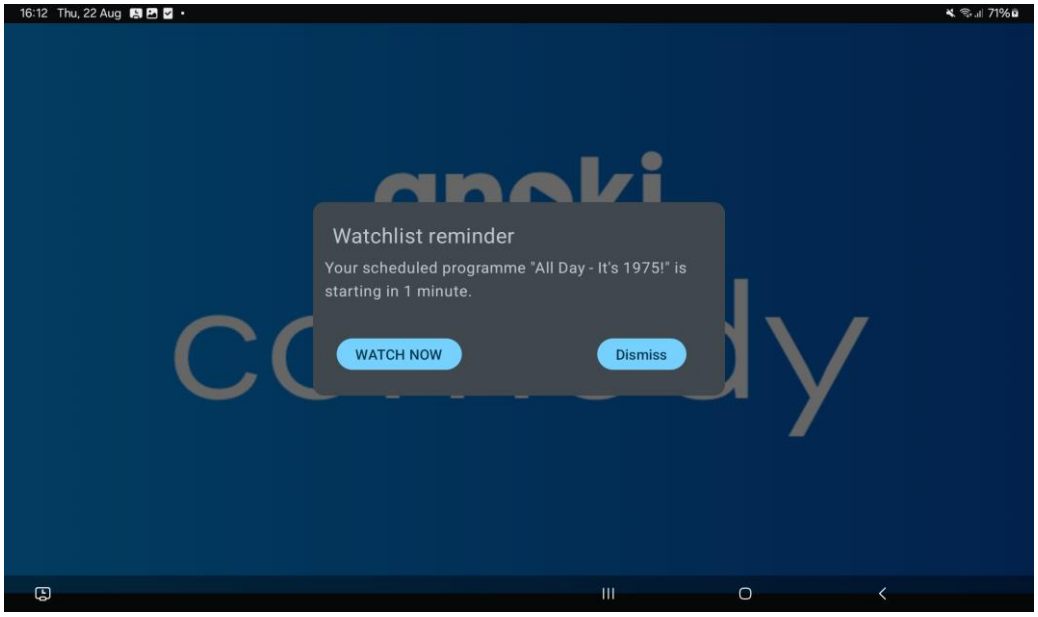


Слика 5.5 Заказивање емисија које почињу у истом тренутку (1. дио)



Слика 5.6 Заказивање емисија које почињу у истом тренутку (2. дио)

Оно што ће се десити када дође вријеме за приказивање обавјештења о почетку ових ***TV*** емисија (у 16:29) видјећемо на слици у поглављу 5.4. До тада, емисија заказана од раније, из ***БП*** ускоро почиње и излази нам прво обавјештење како би нас подсјетило на то:



Слика 5.7 Приказ обавјештења о заказаној емисији

Кликом на WATCH NOW отвара нам се ***TV*** канал за гледање емисије о 1975. години:

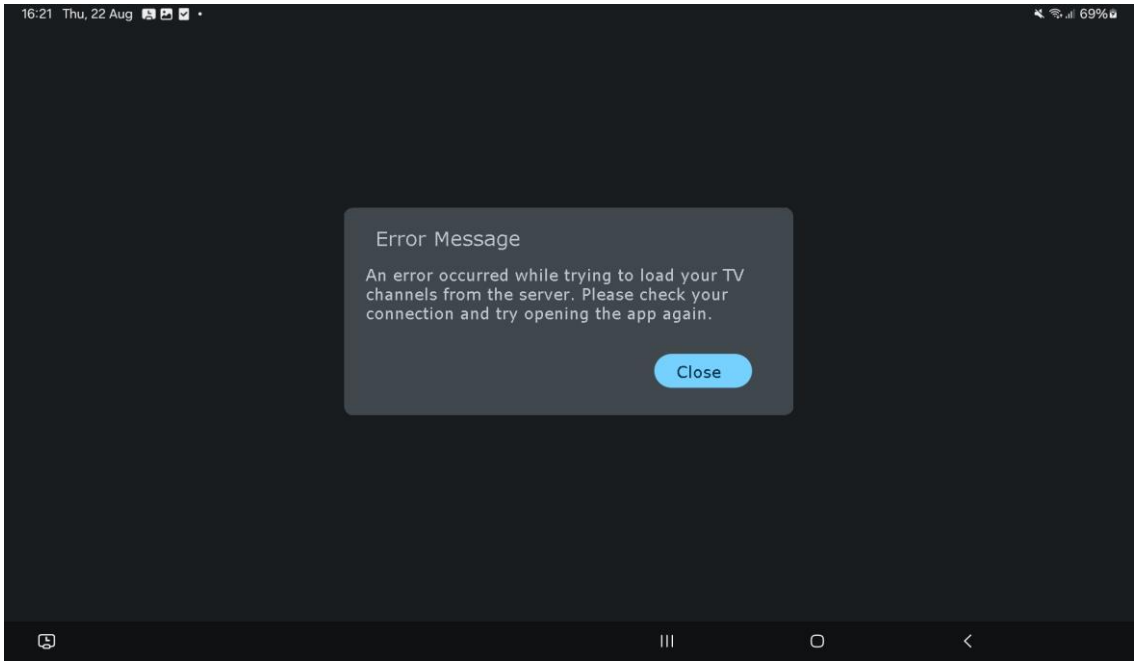
Слика 5.8 Исход потврдног одговора на обавјештењу

Ово је био кратак приказ основних могућности наше апликације. Све могуће грешке и остали посебни случајеви биће приказани и обрађени у наредним поглављима.

5.2 Руковање грешкама

Разноврсност покрета корисника је врло ограничена јер се кроз читаву апликацију, као што је описано у претходним поглављима, све радње обављају само кликом. То значи да је могућност грешке у раду врло мала због изузетно једноставног начина коришћења апликације, што и јесте најбитније при развоју ***TV*** апликација.

Једини случај у којем може доћи до грешке, који је превиђен у кораку 4.5, јесте да нисмо у могућности добавити ***TV*** канале од послужиоца. У том случају потребно је обавијестити корисника о грешци и упутити га шта може покушати како би је отклонио.



Слика 5.9 Порука о грешци када не можемо добавити *TV* канале

5.3 Новозаказана емисија почиње прије већ заказаних

У кораку 4.4.1 смо причали подробније о томе на који начин се праве и обрађују подсјетници за заказане ***TV*** емисије. Међутим, шта ће се десити уколико је одбројавање за приказ обавјештења неке заказане ***TV*** емисије већ почело, а корисник у међувремену жели подјестник за неку другу ***TV*** емисију која почиње прије ње? У том случају се отказује тренутно кориштени одбројач за подсјетник, нови подсјетник (`reminder`) се убацује у Мапу `reminderAlarmsMap` те се поново позива *`makeReminderTimer`*.

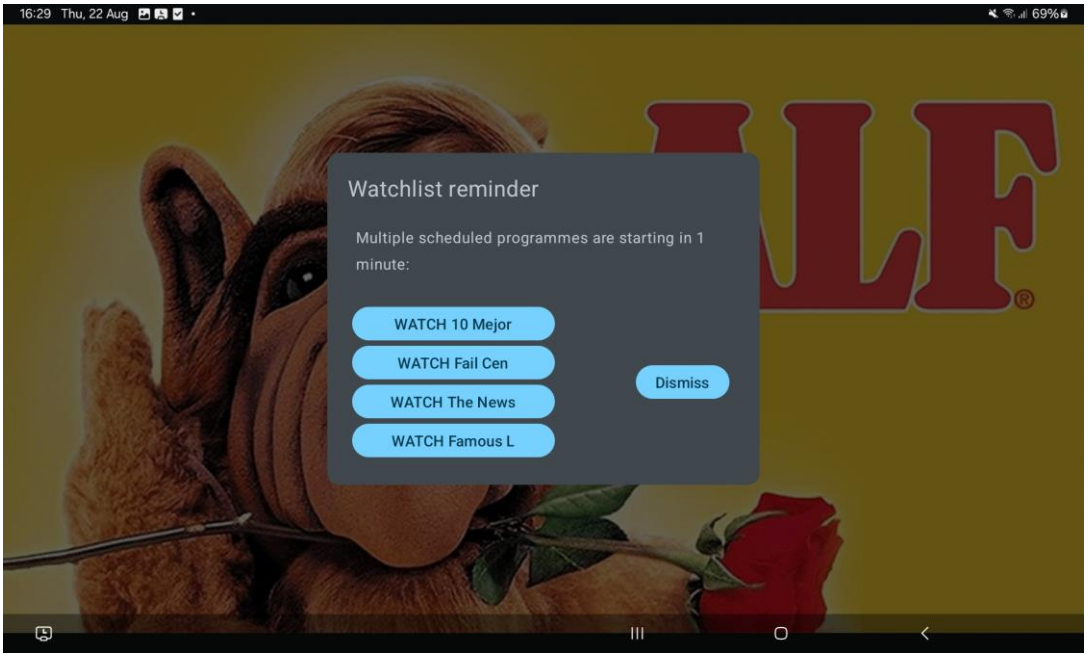
```
if(notificationTimer!=null && reminder.programmeStartTime<notificationEpoch)
{
     notificationTimer?.cancel();
     notificationEpoch = reminder.programmeStartTime
     makeReminderTimer(reminderAlarmsMap[reminder.programmeStartTime]);
     notificationTimer?.start();
}
```

Слика 5.10 Код избјегавања грешке код новозаказаних емисија

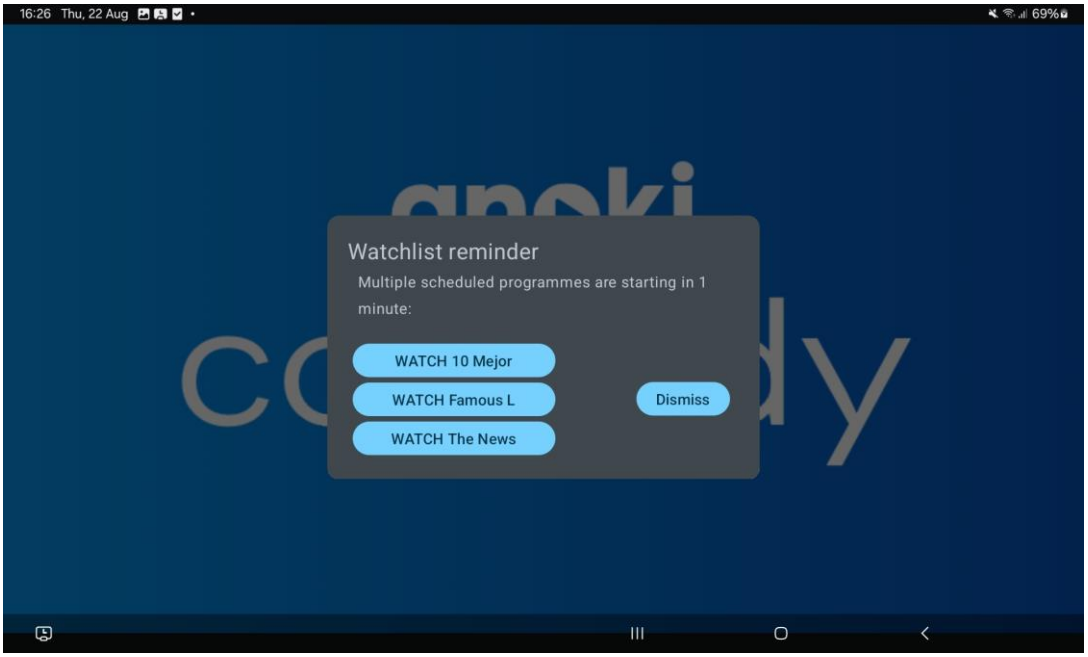
Поновним позивом *makeReminderTimer* ће отпочети ново одбројавање за (нову) прву емисију или скуп емисија који почињу прије осталих.

5.4 Заказано је више емисија које почињу истовремено

Уколико је заказано више емисија које почињу истовремено, потребно је уредити приказ са онолико дугмади колико има емисија, с тиме да не можемо унапријед знати колико ће их бити. Поред тога, прилагодљивост приказа треба бити над самим називима ***TV*** емисија, чији наслови су различитих дужина што проузрокује различиту величину дугмади и врло незграпан и ружан изглед апликације. Рјешење је ограничити приказ наслова емисије на првих неколико (у нашем случају 10) знакова. Након уведених измјена, прикази подсјетника изгледају овако:



Слика 5.11 Обавјештење о почетку 4 заказане емисије истовремењег почетка



Слика 5.12 Обавјештење о почетку 3 заказане емисије истовремењег почетка

6. Закључак

У оквиру рада је подробно објашњен и описан цијели поступак прављења као и начин рада једног сложеног система за рад са ***TV*** сервисима паметних ***DTV*** уређаја, у новом, савременом руху, радећи са не само најновијим технологијама него и на најнапрједнијим уређајима (таблети и мобилни телефони). Сврха је итекако очигледна, пратити нове технологије, у духу времена и омогућити приступ и кориштење апликације у сваком тренутку преко "џепних уређаја", али и не само то!

Још једна од кључних ставки је та да кориштењем новог *Jetpack Compose* окружења омогућавамо бржи рад тако што, раздијеливши рад на мање ставке, можемо приликом кориштења освјежавати само оне ставке чија стања се мијењају, а не цијели екран. Овај нови начин исцртавања ***GUI***-а ће заувијек промијенити свијет Android-а је описни и говори шта и како нацртати, а не како и када ускладити подешавања сваке ставке ***GUI***-а по корацима и без потребе учења ***XML***-а. Овакав начин итекако убрзава и сам развој апликације због своје једноставности и не учења додатног програмског језика (***XML***-а). Недостатци у односу на ***XML***, иако идаље постоје, дјелују у потпуности отклоњиви у блиској будућности те предвиђам да ће врло брзо постати и главни начин развоја апликација не само у ***DTV*** области, него и *Android* програмирању уопште. Лакши развој ***GUI***-а значи више времена за рад на побољшању дојма апликације на корисника (***UX***) те и већој потражњи на тржишту. Због свега тога *Jetpack Compose* ће несумњиво и знатно повећати број корисника *Kotlin*-а, али и подстаћи друге да направе слична окружења за развој ***GUI***-а.

Иако је истраживање *Jetpack Compose*–а главни разлог израде овакве апликације, не можемо се не осврнути на ***DaggerHilt*** као изванредну спону која нам изузетно олакшава уклапање и рад са веома разноврсним окружењима и чини истовремено руковање и Базама Података, и корисничком спрегом, и програмском логиком брзим, лаким и једноставним, без скоро икаквог додатног кода. ***DaggerHilt*** као такав већ увелико добија превласт над својим претходницима ***Dagger1*** и ***Dagger2***, те ће врло брзо, уколико се не појави нешто још боље, постати главно окружење за управљање радом *Android* апликација.

7. Коришћени извори

[1] Софтвер у Дигиталној телевизији 1 - др Милан Бјелица, др Никола Теслић, мр Велибор Михић - ФТН издаваштво 2017. ISBN 978-86-7892-949-6

[2] Android OS (16.8.2024.) https://www.britannica.com/technology/Android-operating-system

[3] Memory safe langauges in Android (19.8.2024.) https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[4] Kotlin and JVM beginners' guide (23.8.2024.) https://www.linkedin.com/pulse/kotlin-jvm-beginner-friendly-guide-manpreet-kaur

[5] Kotlin introduction (25.8.2024.) https://kotlinlang.org/spec/introduction.html

[6] Thinking in Compose (26.8.2024.) https://developer.android.com/develop/ui/compose/mental-model

[7] Anoki API (3.9.2024.) https://www.anoki.ai/privacy

[8] MVVM clean architecture pattern (21.9.2024.) https://medium.com/@ami0275/mvvm-clean-architecture-pattern-in-android-with-use-cases-eff7edc2ef76

[9] DaggerHilt design pattern (23.9.2024.) https://developer.android.com/training/dependency-injection/hilt-android

[10] RETROFIT - A type-safe HTTP Client for Android and JAVA (24.9.2024.) https://square.github.io/retrofit/

[11] ROOM library for data storage (25.9.2024.) http://www.broadband-forum.org/technical/download/TR-106_Amendment-6.pdf

[12] ROOM with Jetpack Compose (25.9.2024.) https://developer.android.com/jetpack/androidx/releases/room

[13] Getting Started with Lottie Animations in Android apps (28.9.2024.) https://lottiefiles.com/blog/working-with-lottie-animations/getting-started-with-lottie-animations-in-android-app/

[14] EpochTime and LocalDateTime in JAVA & Kotlin (20.9.2024.) https://www.baeldung.com/java-convert-epoch-localdate

[15] Epoch to human readable date converter (20.7.2024.) https://www.epochconverter.com/