**ТЕХНОЛОГИЧНО УЧИЛИЩЕ “ЕЛЕКТРОННИ СИСТЕМИ”**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

**Тема:** Мобилно приложение за социална мрежа за готвачи и любители на готвенето: **„EverydayChef“**

**Дипломант:**  **Научен ръководител:**

*Мартин Пенев Александър Илиев*

**СОФИЯ**

**2020**

Дата на заданието: 15.11.2019 г. Утвърждавам:..............................

Дата на предаване: 15.02.2020 г. /проф. д-р инж. Т. Василева/

**ЗАДАНИЕ**

**за дипломна работа**

на ученика Мартин Василев Пенев 12A клас

1. Тема: Мобилно приложение социална мрежа за готвачи и любители на готвенето - **“EverydayChef”**
2. Изисквания:
   1. Регистрация и логин чрез имейл и чрез фейсбук
   2. Създаване и менажиране на семейства или взимане на съществуващи такива от фейсбук
   3. Списък за пазаруване за всяко семейство
   4. Налични продукти за всяко семейство
   5. Създаване и менажиране на рецепти
   6. Оценяване на рецепти
   7. Любими рецепти на всеки един потребител
   8. Търсачка на рецепти по налични продукти
   9. Нотификации при различни събития
3. Съдържание:
   1. Обзор
   2. Същинска част
   3. Приложение

Дипломант :...........................................

Ръководител:..........................................

/Александър Илиев/

Директор:................................................

/доц. д-р инж. Ст. Стефанова/

Научен руководител:

**ВЪВЕДЕНИЕ**

Интepнeт oтдaвнa вeчe нe e пpocтo мяcтo, ĸъдeтo гyбиш вpeмeтo cи в нaпpaзнo oтвapянe нa paзлични пpoзopци, търсиш информация или убиваш време в игри. Ceгa тoй e нaвcяĸъдe и нaмиpa пpилoжeниe във вcяĸa нaшa eжeднeвнa дeйнocт. Все по-мащабното навлизане на мобилните технологии в ежедневието ни превърна създаването на мобилни приложения в доходоносна бизнес ниша. Хората свикнаха с едно натискане на бутон да правят транзакции, да пазаруват, да кандидатстват, да комуникират и да управляват деня си. Продуктите стават все по-модерни и умни, спестяват ни различни действия, доверяваме им се напълно, те вече са част от нашия живот и от самите нас. Устройствата, които всеки от нас все по-широко използва, стават все по-модерни и комфортни придобивки, които правят живота ни по-интересен и лесен.

В забързаният си живот ние се сблъскваме всеки ден с проблема недостиг на време. В резултат на това, започваме да правим компромиси с начина си на хранене, спорт, обличане и т.н. Обръщайки се за помощ към Интернет или написаната по този въпрос литература, за съжаление, констатираме, че тя е написана в описателен вид, приложенията не са добре и ясно структурирани. За дейности, които сами по себе си са свързани, се налага да се ползват различни приложения и тн., което прави използването им още по трудно в нашето динамично ежедневие.

В ежедневието си човек много пъти на ден си задава въпроса: „Какво ще закусвам, обядвам или вечерям днес?“. Понякога липсват продукти за желаното от него ястие, друг път липсват идеи за това какво да си сготви или знания за това как да го направи, а често липсват и трите. Приложението „EverydayChef“ има за цел да помогне за по-лесното и бързо решаване на подобни проблеми, съответно да улесни и направи по-приятно времето, изкарано в кухнята.

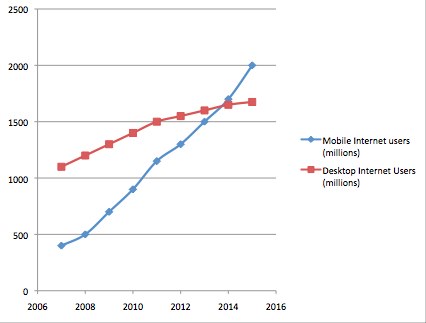
Свързвайки готвачи, любители на готвенето и хора без много опит в сферата, софтуерът се стреми да изгради социална мрежа, специално създадена за изкуството на приготвяне на храна. В нея всеки участник може да черпи от опита на другите и да споделя своя собствен, да получи обратна връзка за работата си и да оценява от своя страна тази на другите. Това основнo се изразява в създаване на рецепти и коментирането и оценяването на такива.

# ПЪРВА ЧАСТ

**Проучвателна част- Технологии и подобни приложения**

* 1. **Защо мобилно приложение**
     1. **Мобилни устройства**

Днешно време технологиите са навсякъде около нас и съответно приложение като „EverydayChef“ може да бъде разработено за всякакви устройства. Това повдига въпроса защо е избрана точно мобилната среда.

*Фиг. 1.1: Използваемост на мобилни и настолни устройства за достъп до интернет:*

Както се вижда на графиката, бройката на потребителите на мобилни устройства през 2016 е вече по-голяма от тази на хората, използващи настолни такива. Проучвания показват, че само една година след това, през 2017 вече 67% от глобалния трафик идва от джобната технология, което вече ясно изгражда тенденцията и за в бъдеще – използваемостта на мобилните устройства за връзка с интернет расте, за разлика от тази на десктоп компютрите. Много по-удобно е за един човек да използва телефона си, за да свърши каквато работа му е нужна, където и когато му е нужна. Това е главната причина да избера тази среда като основна за моето приложение.

* + 1. **Мобилно приложение или мобилен уеб**

[1]Връзката през мобилното устройство до интернет се осъществява чрез мобилно приложение или чрез мобилен уеб. Ако компании с по-стабилни финанси могат да си позволят разработката и на двете за техния продукт, то в повечето случаи, особено в началото на реализацията на проект, се налага да се избира една от двете опции. Изборът обикновено се основава на разликите в цената на изпълнение, целта на продукта и клиентите, за които е предназначен. Въпреки това изследвания показват, че хората предпочитат мобилните приложения пред мобилните уеб сайтове.

Ето някои от причините за избор на приложение пред уеб:

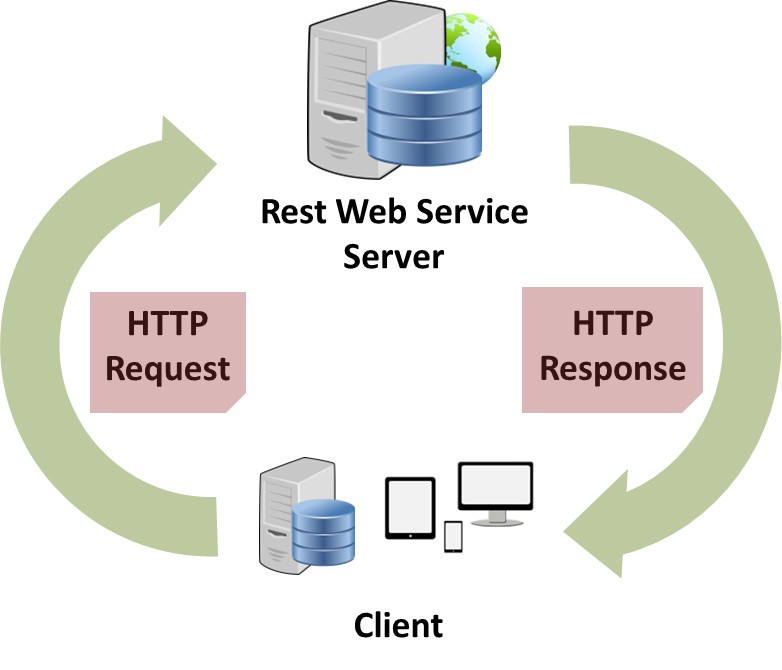
* **По-добра персонализация.** Когато се очаква продуктът да се използва ежедневно, мобилното приложение е най-добрият вариант поради възможнстта за настройването му по вкус на потребителя и поддръжката на важна информация под ръка.
* **Push notifications (Директни съобщения).** Вградена функция на мобилните приложения е изпращането на директни съобщения към потребителя при различни събития. Въпреки че такива съобщения могат да се изпращат и от уеб приложенията, тези от мобилните приложения предоставят доста по-богато оформление, както и интерактивност на съобщениетo. Поради това може да се каже, че в този аспект мобилните приложения превъзхождат тези в уеб.
* **Лесен достъп до устройството и неговите функционалности.** През мобилното приложение много лесно се използват камерата, контактите, историята на обажданията и подобни функции на телефона. По този начин може да се осигури много по-приятна и улеснена работа с продукта за потребителя, както и се намалява времето, за което той би свършил нещо, изискващо тези функции, чрез мобилния уеб.
* **Няма нужда от интернет връзка.** Много полезно предимство на мобилните приложения е това, че могат да се ползват и при липса на обхват. Това е, защото понеже приложението е инсталирано на самото устройстройство то запазва локално последното заредено съдържание и винаги показва него. При наличие на интернет връзка това съдържание се обновява.
* **По-висока скорост.** Едно добре направено приложение може да процедира много по-бързо от една уеб страница. Една от причините е именно локално запазената (кешираната) информация в приложението.
  1. **Използвани технологии**

За разработване на отделните части на приложенито са използвани някой от най-новите и разпостранени технологии, като се цели получаването на най-оптимално използване на предимствата, които предоставя всяка една от тях.

* + 1. **REST технология**

„EverydayChef“ е имплементирано на базата на REST. REST[1] (Representational state transfer) представлява софтуерна архитектура, предоставяща дефинирани стандарти за връзката между компютърните устройства в мрежата, за по-лесна комуникация помежду им. Основава се на HTTP – мрежов протокол, който е в основата на обмена на данни в глобалната мрежа, която познаваме днес.

Характерно за създадените по този начин системи (наричани RESTful системи) е, че при тях са разделени понятията клиент и сървър, както и на сървъра не му трябва да знае какво е състоянието на клиента във всеки един момент от време и обратното.

*:*

**Основни характеристики на REST технологията:**

* + - * **Разделение на клиент от съръвър**

При REST архитектурата клиентът и сървърът се изграждат напълно отделно без едното да се интересува от другото. Това значи, че могат да бъдат променяни по всяко време и това да не засяга другата страна.

По този начин интерфейсът по-лесно може да бъде интегриран на друга платформа и има много възможности за подобрение.

Чрез REST архитектурния стил различни клиенти могат да се закачат за същите пътища на интерфейса, да извършват еднакви операции, както и да получават еднакви данни.

* + - * **Не е необходимо да се поддържа информация за текущото състояние на отсрещната страна**

При всички приложения, следващи REST, на сървъра не му трябва да знае текущото състояние на клиента и обратното. По този начин и двете страни разбират всяко едно съобщение помежду си без да се налага да виждат предишните. Това се осъществява чрез използването на ресурси вместо на команди. Това са обекти, документи или информация, която трябва да бъде запазена или обработена по някакъв начин и/или изпратена към друго устройство.

Това спомага за ефективността и скоростта на RESTful приложенията, както и за тяхното лесно разрастване, тъй като отделните компоненти могат да бъдат променяни и преизползвани без това да се отразява на системата като цяло.

* + - * **Комуникация клиент - сървър**

В REST архитектурата клиентите изпращат заявки (requests), за да получат или да променят някакви ресурси, а сървърите връщат отговори (responses) на тези заявки. Цялата комуникация се извършва във вид на JSON(JavaScript Object Notation) – формат за обмен на данни, който е независим от езика, лесен за разбиране от хора и машини и пестелив откъм размер.

* + - * **Създаване на заявки**

REST изисква клиента да изпрати заявка, за да получи или да редактира информация на сървъра. Заявката обикновено се състои от:

* **HTTP Глагол** – заявява каква операция трябва да се извърши
* **HTTP header** – позволява на клиента да даде допълнителна инфорамция за заявката
* **Път (path) към ресурс**
* **Тяло на заявката** (само за някой глаголи) – не е задължително

**Основни елементи в REST:**

* + - * **Глаголи в HTTP**

Най-основните глаголи които се срещат в HTTP са:

* **GET** – получаване на специфичен ресурс (чрез неговото id) или на множество ресурси
* **POST** – създаване на нов ресурс
* **PUT** – редакция във вече съществуващ ресурс
* **DELETE** – изтриване на ресурс

Същестуват още глаголи, но изброените са основните и необходимите за създаването на RESTful приложение.

* + - * **Пътища (Path)**

Всяка заявка трябва да има път към ресурс, който трябва да се върне или промени по указан начин. В RESTful приложенията пътищата са така направени, че да е ясно накъде сочат.

По конвенция, първата част трябва да бъде множественото число на ресурса. Това прави вложените пътища лесни за четени и разбиране. Например за път като: *somesitedomain.com/books/12/pages/3*

ясно се разбира, че ще се извършва някаква операция с страницата с *id = 3* на книгата с *id = 12.* Пътищата трябва да съдържат толкова информация, колкото е необходима за да се локализира ресурсът. За операция *POST,* например, няма нужда да се записва *id*, тъй като сървърът автоматично ще сложи такова на новия създаден ресурс.

* + - * **Изпращане на отговори**

Както вече се спомена, в зависимост от получената от клиента заявка, сървърът връща отговор, в който се съдържа информация за това как е била обработена заявката, както и желаните данни ако има такива, в какъв формат са те и други.

* + - * **Кодове на състоянието (Status codes)**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

За да се извести клиента за това дали поисканата от него операция е извършена успешно или не, в отговора на сървъра се съдържа т. нар. status code (на български: „код на състояниетo“). Такива кодове има много на брой, но най-известните от тях са посочени в Таблица 1.2 по-долу:

***с****:*

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

За всеки HTTP глагол има очаквани статус кодове, които трябва да бъдат върнати от сървъра при успешно изпълнение:

* GET — връща 200 (OK)
* POST — връща 201 (CREATED)
* PUT — връща 200 (OK)
* DELETE — връща 204 (NO CONTENT)

Ако при обработка на HTTP заявката възникне грешка, сървърът трябва да върне най-специфичния възможен статус код за грешка, който да отговаря на изникналия проблем.

* + 1. **Android studio – за разработване на потребителския интерфейс на приложението**

**(Frontend)**

Частта от приложението, която потребителят ще вижда и с която ще работи(frontend), е писана чрез Android Studio [3]. Това е официалното IDE (integrated development environment - среда за разработка), специализирано за операционната система андроид. Излиза за първи път на 16.05.2013г., а първата стабилна версия е обявена декември 2014 година.

Поддържа се както новоизлезлият и обявен за предпочитан от Google език Kotlin, така и старите основни езици като Java и C++. Предоставени са множество функционалности, които улесняват разработката на приложения и увеличават удобството на използване като:

* Gradle build
* Специфични за андроид рефактори и quick fix опции
* Lint средства за засичане на различни проблеми
* ProGuard интеграция
* Темплейт-базирана обработка на дизайна и компонентите на приложенията
* Среда за обработка на дизайна
* Виртуален емулатор

* + 1. **Spring boot – за разработване на вътрешната част на приложението**

**(Backend)**

Другата част от приложението – сървъра (backend), е написана чрез Spring Boot. Това e open-source micro framework, поддържан от компания на име Pivotal. Разработен е върху framework на име Spring. Чрез него се осигурява автоматична конфиргурация на Spring проект, с което се спестява много време и усилия на програмистите.

Spring излиза през 2002 година и много бързо се утвърждава като доминиращ начин, по които се пише Java код. Разработен е open-source (целият код е достъпен) и има за цел да направи писането на сървърната част на Java по-лесен и удобен. През 2012 година започват дискусии по начин за подобрение на framework-a. 06.07.2013 Фил Уеб пише, че вместо да се променят вече съществуващия Spring, са започнали да работят по нов проект, наречен Spring Boot, който да оптимизира стария. По тази причина може да се каже, че Spring Boot излиза за да се направи разработката на уеб приложения с Spring по-достъпна и разбираема.

* 1. **Анализ на приложения с подобно предназначение**

Приложения, които да са от полза на човек при ежедневните му проблеми съвсем не са рядкост. Такива има много и най-различни. Честата нужда от помощ при готвене вече е зародила различни идеи в главите на различни хора за това как може така разпространената технология да помогне в такива ситуации.

* **Pepperplate**

Това приложение е много добър пример за мобилен помощник в кухнята, какъвто представлява и “EverydayChef” Потребителят може да създаде свои рецепти, да ги организира в менюта, които в последствие да включи в хранителния си план за седмицата или за месеца. При липса на желаните за съответната рецепта продукти той може да ги сложи в списъка с нужни покупки, за да е наясно какво му е необходимо при следващото му пазаруване.

Като негови добри особености могат да се посочат:

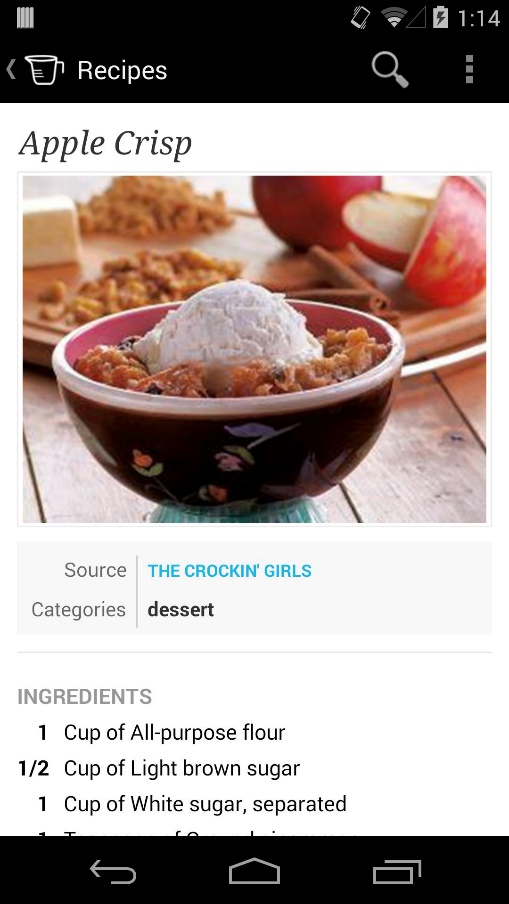
* Удобна за потребителя
* Възможност за вкарване ()готови

*Фиг. 1.3 – “Pepperplate“ - страница за рецепти*

Като негови недостатъщи може да се посочат следните такива:

* о.
* Липса на свързаност с други хора.
* .

*Фиг. 1.4 – “Pepperplate“ - страница с детайли за една рецепта*



# ВТОРА ЧАСТ

**Изисквания.**

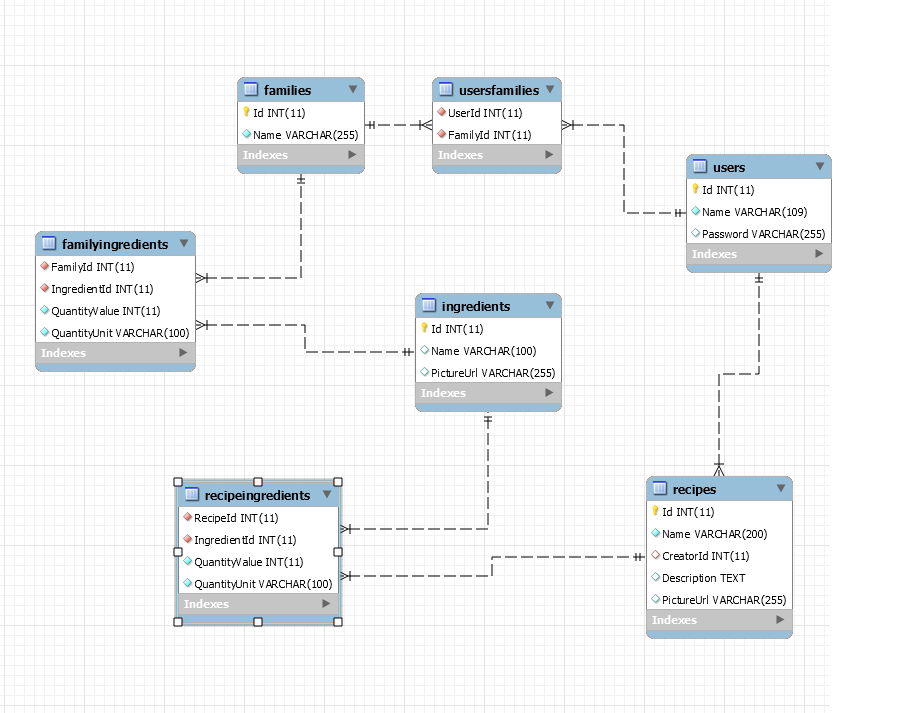
**Структура на базата и описание на алгоритъма.**

* 1. **Изисквания към програмния продукт**

Приложението ‘EverydayChef’ трябва да поддържа:

* Регистрация и логин чрез имейл и чрез фейсбук
* Създаване и менажиране на семейства или взимане на съществуващи такива от фейсбук
* Списък за пазаруване за всяко семейство
* Налични продукти за всяко семейство
* Създаване и менажиране на рецепти
* Оценяване на рецепти
* Любими рецепти на всеки един потребител
* Търсачка на рецепти по налични продукти
* Нотификации при различни събития
  1. **Структура на базата данни**

*Фиг. 2.1 – модел на база данни “everydaychef”*



* 1. **Описание на алгоритъма**
     1. **Логин и регистрация**

При влизане в приложението се показва страница, на която потребителят трябва да избере една от опциите за логин (чрез gmail или чрез facebook) или съответно да се регистира, ако все още няма акаунт за EverydayChef. – LoginActivity

След успешен логин се извлича информацията за този акаунт чрез заявки до сървъра и в различните страници (фрагменти) на основната част от приложението (MainActivity) се показва семейството, към което принадлежи, харесани и създадени рецепти, както и списъците с продукти за пазаруване.

* + 1. **Семейство**

Всеки потребител на EverydayChef може да участва в едно семейство. Участниците в семейството споделят общи налични продукти, харесани рецепти, списъци с нужни покупки и списък с рецепти за скоро готвене.

Чрез отварящото се отдясно меню (NavigationDrawer) потребителят може да избере страницата за профил (ProfileFragment). В нея той може да редактира информацията си, както и да управлява семейството, към което принадлежи в случай, че то е създадено от него. Ако това семейство е създадено от някой друг, потребителят може само да преглежда информацията свързана с него или да го напусне.

* + 1. **Рецепти**

Могат да се създават рецепти от всеки потребител, да се променят, премахват и харесват чрез съответните заявки към бекенда. Рецептите съдържат нужните за тях продукти.

От NavigationDrawer потребителят може да избере страницата за профил, където да види собствените си рецепти, или страницата за готвене (CookFragment), откъдето може да намира други рецепти според това какви продукти има или това какво е търсил в търсачката. При влзиането в страницата за специфична рецепта (RecipeFragment) се извличат от бекенда нужните продукти за нея, описанието на приготвянето и, снимката на рецептата, ако има такава, и други. В горната част на страницата има бутон добавя нейните съставки в списък за покупки в случай, че семейството няма тези продукти в наличност.

* + 1. **Списъци с нужни покупки**

Всяко семейство има възможност да си създава списъци с продукти, които са споделени за всеки един член на семейството.

Отново от NavigationDrawer потребителят може да достъпи страницата със списъците (ShoppingListFragment). На нея той трябва да избере кой от наличните списъци да използва чрез падащото меню в горната част на страницата. След това всички продукти, които присъстват в избрания списък се показват на страницата. нея могат да се добавят продукти за купуване както ръчно - чрез бутона на страницата, така и динамично чрез добавяне на рецепта в списъка за готвене на семейството. Отбелязаните продукти за купени се добавят в наличните продукти на семейството.

* + 1. **Търсачка на рецепти**

Това е най-важната функционалност на приложението. Всеки потребител може да търси рецепти, които да отговарят възможно най-много на неговите продукти.

През NavigationDrawer потребителят достъпва страницата с търсачката (CookFragment). На нея веднага се прилага алгоритъмът за филтрация на рецепти според наличността - сортират се получените рецепти по това до каква степен потребителят има нужните за приготвянето им продукти. Освен чрез налични продукти, потребителят може да търси рецепти и просто по името им – чрез текството поле най-горе на страницата.

* 1. **Аргументация за използвани технологии** 
     1. **Android Studio**

Освен Android Studio (АS), хората разработващи приложения за Android света познават и друго популярно, но по-старо IDE – Eclipse. Причините, поради които изборът за разработката на EverydayChef е това на Google (AS), са няколко. На първо място, продуктът предлага вградена Gradle build система – удобен помощник в управлението на начина, по който приложението се сглобява (build). Освен това интерфейсът е много интуитивен и лесно се използват отделните компоненти. За разлика от Eclipse, AS предлага и графичен интерфейс, чрез който процесът на разработка и стилизация на приложения е много улесенен. Друга важна причина е, че Android Studio е създадено специално за Android, докато Eclipse се използва за всяка платформа и всякакъв език. Студиото предлага много по-стабилно представяне от своя опонент, както и за работа по желания начин се изисква значително по-малко оперативна памет и процесорна скорост. И двете IDE-та се славят с оптимална code auto completion функционалност – автоматични предложения за допълване или поправяне на грешка в кода.

Всички изброени причини са в основната на моят избор да разработвам EverydayChef на Android Studio. Това също са и причините за това, че от неговото обявяване през 2013 година IDE-то е най-използваният продукт за разработка на Android приложения.

* + 1. **Spring boot**

Що се отнася до бекенд технологиите, въпросът какво да бъде използвано е малко по-труден заради по-големия брой варианти като NodeJS - Express, RubyOnRails, Laravel, Flask и много други. Spring Boot е избрано за този проект по следните причини:

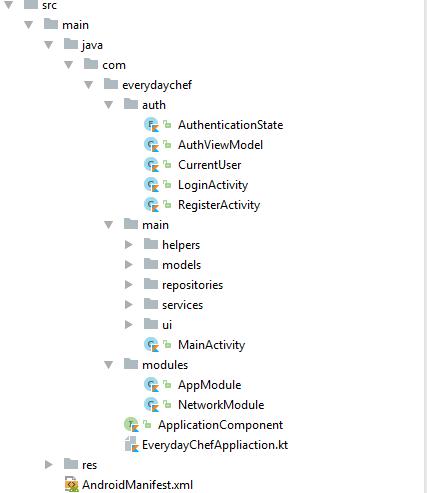
* Разработван от много време – Spring Boot е базиран на Spring Framework, който сам по себе си е разработван над 15 години като алтернатива на JEE стакът.
* Стабилен – Основните модули на Spring екосистемите са стабилни за дълго време и повечето промени лесно могат да бъдат отменени
* JVM - Spring Boot е базиран на Виртуалната машина на Java, което осигурява по-голяма защита и независимост от използваната платформа.
* Приятен за разработка – Spring Boot предоставя доста удобства за да улесни разработката, с което привлича доста софтуеристи.

# ТРЕТА ЧАСТ:

**Начин на разрабока на алгоритъма**

* 1. **Част приложение – фронтенд**

Частта от приложението, която потребителят вижда и с която комуникира. За основна цел беше поставено да се разработи интуитивен дизайн, който да не затруднява никого.

* + 1. **Структура**

*Фиг. 2.1 – Структура на проекта*

В приложението има 3 activity компонента:

* LoginActivity – за предоставяне на данни за аутентикация
* RegisterActiviy – за създаване на потребител
* MainActivity – за основната интеракция с потребителя

Първите две имат само по един основен файл за оформление („*layout file“*). В MainActivity се съдържат множество фрагменти, които се сменят на екрана на потребителя по време на неговата навигация в приложението. Тези 3 activity компонента са разделени в 2 основни подпакета – auth и main.

* В подпакет “auth” е частта от приложението, която отговаря за аутентикацията на потребителя и извличането на неговите данни. В него са имплементирани LoginActivity и RegisterActivity
* В подпакет „main” е основната част от приложението. Имплементиран е класът MainActivity, който управлява работата на цялото приложението. Дъщерни компоненти в него се явяват множество фрагменти, които са описани в подпакета ui. В другите поддиректории са разпределени отделните компоненти от архитектурата на приложението. Имлементирани са множеството класове, които да улесняват разрабоката, и са поставени в папка “helpers”.

В приложението е имплементирана архитектура подобна на *фиг. 3.1*, с която се постига енкапсулация на отделните методи и функционалности в отделни компоненти. Тези компоненти са отделени като подпакети на “main” директорията.

*Фиг. 3.1 – Архитектура на фронтенд*



Връзката между компонентите се осъществява чрез “dependency injection” (осигуряване на необходими ресурси за даден обект от друг обект), с помощта на библиотеката “dagger”.

За целта се създава интерфейс, който носи анотацията @Component и в който се описват всички методи, които са нужни:

@Singleton  
@Component(modules = [NetworkModule::**class**, AppModule::**class**])  
**interface** ApplicationComponent {  
 **fun** inject(activity: MainActivity)  
 **fun** inject(activity: LoginActivity)  
 **fun** inject(activity: RegisterActivity)  
 **fun** inject(fragment: HomeFragment)  
 **fun** inject(fragment: ProfileFragment)  
 **fun** inject(fragment: NotificationsFragment)  
 **fun** inject(fragment: FridgeFragment)  
 **fun** inject(fragment: CookFragment)  
}

Чрез този интерфейс Dagger създава имплементация на така наречения graph от обекти, свързани по своите зависимости. Инстанция на класът, описващ този graph, се взима в друг клас EverydayChefApplication, който да наследява класът Application, за да е достъпен за всички компоненти:

**val** appComponent = DaggerApplicationComponent.builder()  
 .appModule(AppModule(**this**))  
 .build()!!

От тук всички Activity и Fragment компоненти могат да извикат съответните функции, с които да изпълнят зависимостите на класовете, които са анотирани с @Inject, което информира Dagger както това кои класове са нужни, така и как да ги създава. Създаването обаче на компоненти, при които само тази анотация не е достатъчна, се осъществява чрез модули с анотиция @Module, в които се описват нужните стъпки при създаването на компонентите чрез анотацията @Provides:

@Provides  
**fun** provideUserService(): UserService {  
 **return** retrofit.create(UserService::**class**.java)  
}

* + 1. **Навигация**

За да може всички страници в EverydayChef да са лесно достъпни, основният компонент в приложението (MainActivity) е с имплементация на навигационно меню („navigation drawer“) – това е панел, който показва линкове към отделните страници. Отваря се, когато се натисне бутон с изображение, подобно на *фиг. 3.1*.

*Фиг. 3.2 – бутон за навигационно меню*



По този начин потребителят има ясна представа за това какво има в приложенито и как да го достъпи.

Целият процес по навигацията се управлява от MainActivity класа. Навигацията представлява връзката между дестинациите (отделните части на интерфейса, до които потребителят може да достигне) в приложението. Връзката между тези дестинации се описва в *navigation graph* – това е ресурс, който репрезентира всички пътища, по които може да поеме потребителят. Дестинациите представляват фрагменти - части от интерфейса на потребителя. Началният фрагмент е този екран, който се показва на потребителя при първоначално зареждане и този, който той вижда преди излизане от приложението. За начален фрагмент е зададен *HomeFragment* чрез *app:startDestination* атрибутът на <*navigation>* компонента, описващ *navigation graph*.

За начало, в главния файл за оформление („*layout file“*) за основен компонент е сложен *DrawerLayout*, a за дъщерен на него *NavigationView* компонентът:

<DrawerLayout>

<include/> - *toolbar* на приложението и *NavHost(\*)*

<NavigationView/>

</DrawerLayout>

(\*) – това е важна част от един *navigation graph*, представлява празен контейнер, в който се сменят отделните дестинации при навигирането на потребителя през тях.

Удобният за изпозване в андроид компонент – *NavigationUI*, който отговаря за навигацията в приложението предоставя множество функции, чрез които да се създаде лесно и удобно за първи път *NavigationDrawer*.

Извикват се методите:

setSupportActionBar(toolbar)

appBarConfiguration = AppBarConfiguration(

setOf(

R.id.nav\_home,

R.id.nav\_fridge,

R.id.nav\_cook,

R.id.nav\_shopping\_list,

R.id.nav\_share,

R.id.nav\_send,

R.id.nav\_profile

), drawer\_layout

)

setupActionBarWithNavController(navController, appBarConfiguration)

nav\_view.setupWithNavController(navController)

По този начин се осъществява връзката между създадените дестинации от *navigation graph* и елементите на менюто NavigationDrawer.

* + 1. **Логин и регистрация**

При влизане в приложението се винаги проверява дали данните на потребителя са записани на устройството. Ако това не е така, се стартира класът LoginActivity, с който потребителят да се аутентикира и данните му да се кешират в устройството, за да не се налага да се прави това повече. Това става чрез функцията authenticateUser():

**private fun** authenticateUser(){  
 authViewModel.currentUserLd.observe(**this**, Observer**{  
 if** (!authViewModel.isUserSignedIn()) {  
 goToLogin()  
 } **else** {  
 popupUtility.displayLongDefault(**"Logged in as: "** + it.username)  
 updateHeaderUI()  
 }  
 **}**)  
}  
  
**private fun** goToLogin() {  
 startActivity(Intent(**this**, LoginActivity::**class**.java))  
}

Във ViewModel класa отговарящ за аутентикацията се държи обект от тип MutableLiveData, който позволява да се следи за промени в съдържанието на променливата чрез функцията observe(). При всяка промяна на стойността на currentUserLd, където се държи текущият потребител, се проверява дали има такъв. Ако не, се използва функцията startActivity() с аргумент Intent, на който е подаден класът на LoginActivity и по този начин потребителят е прехвърлен към компонентът за аутентикация. Ако се установи, че има записан текущ потребител, неговите данни се въвеждат на страницата и се изписва съобщение чрез помощник класа PopupUtility.

Поради това, че приложението е осъществено по архитектурата *REST*, се използва *Jwt Authentication* – Идентификация на клиента чрез предварително даден от сървъра ключ(т.нар. *JSON Web Token или JWT*). В EverydayChef има три начина на идентификация и на получаване на съответния ключ:

* Чрез *Google;*
* Чрез *Facebook;*
* Чрез име и парола.

И трите метода са имплементирани чрез функцията:

**fun** setCurrentUser(username: String, email: String = **""**, accessToken: String = **""**, photoUrl: String = **""**, method: String = **"manual"**){  
 userService.getUserByUsername(username).enqueue(**object** : Callback<User> {  
 **override fun** onFailure(call: Call<User>?, t: Throwable?) {  
 Log.println(Log.ERROR, **"Process Authenticate"**, t.toString())  
 }  
  
 **override fun** onResponse(call: Call<User>?, response: Response<User>?) {  
 **val** body = response?.body()  
 **when** {  
 response?.code() == 200 -> {  
 fillUserFromBody(body!!, photoUrl, accessToken)  
 setUserAuthenticationState(method)  
 }  
 response?.code() == 404 -> {  
 **if** (method != **"manual"**) {  
 registerWithMethod(username, email, method, photoUrl, accessToken)  
 } **else** {  
 setUserAuthenticationState(**"invalid"**)  
 }  
 }  
 **else** -> {  
 setUserAuthenticationState(**"invalid"**)  
 }  
 }  
 }}}}}

Заявката към сървъра се извършва, чрез съответния Service клас, който използва библиотеката Retrofit. За горната заявка се използва:

**interface** UserService {

...

@GET(**"user/name/{username}"**)  
**fun** getUserByUsername(@Path(**"username"**) username: String): Call<User>

...

}

* **Аутентикация чрез Google**

Ключ(JWT) се взима чрез свързване с “Google API Console“, където се създава и конфигурира проект за връзката с Google.

В AuthViewModel са предоставени отделните функции за запис в системата чрез Google акаунт. Първоначално се създава инстанция на GoogleSignInOptions и се описва начинът, по който да се осъществи процесът и каква информация се изисква:

**val** gso: GoogleSignInOptions = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT\_SIGN\_IN)

.requestIdToken(context.getString(R.string.google\_server\_client\_id)).requestEmail().build()  
mGoogleSignInClient = GoogleSignIn.getClient(context, gso)

Също така се взима инстанция и на класа GoogleSignInClient с така зададените настройки. След това се описва финкционалността на бутона, който задвижва целия процес:

listener.setOnClickListener**{**googleSignIn(context **as** Activity)**}**

Където googleSignIn() e:

**fun** googleSignIn(activity: Activity) {  
 Log.println(Log.INFO, **"Action"**, **"Started process: google sign in"**)  
 **val** signInIntent: Intent = mGoogleSignInClient.signInIntent  
 activity.startActivityForResult(signInIntent, RC\_GOOGLE\_SIGN\_IN)  
}

На текущия activity компонент му се вика функцията startActicityForResult(), която, както името показва, стартира компонент от тип activity, който след това се очаква да върне резултат. Този резултат ще бъде получен в метода onActivityResult() на съответния компонент, от който е стартиран процесът:

**override fun** onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
 **super**.onActivityResult(requestCode, resultCode, data)  
 **when**(requestCode){  
 authViewModel.RC\_GOOGLE\_SIGN\_IN -> {  
 **val** task =  
 GoogleSignIn.getSignedInAccountFromIntent(data)  
 handleGoogleSignInResult(task)  
 }

...

}

**private fun** handleGoogleSignInResult(completedTask: Task<GoogleSignInAccount>?) {   
authViewModel.googleSignInSuccessful(completedTask!!.result!!)  
}

При успех се извиква методът на AuthViewModel googleSignInSuccessful(), където потребителят бива записан в устройството.

userRepository.setCurrentUser(  
 username = currentGoogleUser?.displayName.toString(),  
 accessToken = currentGoogleUser?.idToken.toString(),  
 email = currentGoogleUser?.email.toString(),  
 method = **"google"**,  
 photoUrl = currentGoogleUser?.photoUrl.toString()  
)

* **Аутентикация чрез Facebook**

Взимането на ключ(JWT) oт Facebook става по подобен начин. Връзка се осъществява с „Facebook SDK“, където отново се създава и конфигурира проект, който да се използва от Everydaychef.

След това е нужна регистрация на процеса в бутона. Това става чрез CallbackManager:

**val** facebookCallbackManager = CallbackManager.Factory.create()

Тази инстанция се подава при самата регистрация на процеса:

button.registerCallback(facebookCallbackManager, **object** : FacebookCallback<LoginResult?> {  
 **override fun** onSuccess(loginResult: LoginResult?) { loginResult?.accessToken?.let **{** fillDataFromFacebookAccount(it) **}** }

...

Последната функция, подобно на тази при методът с Google, записва данните на потребителя в устройството чрез върнатия ключ при успешни минали операции.

**val** request = GraphRequest.newMeRequest(accessToken) **{** `object`, \_ **->  
 try** {  
 **val** name = `object`.getString(**"name"**)  
 **val** email = `object`.getString(**"email"**)  
 **val** image = `object`.getJSONObject(**"picture"**).getJSONObject(**"data"**).getString(**"url"**)  
   
 userRepository.setCurrentUser(  
 username = name,  
 accessToken = accessToken.token,  
 email = email,  
 method = **"facebook"**,  
 photoUrl = image  
 )

...

**val** parameters = Bundle()  
parameters.putString(**"fields"**, **"id,name,email,picture.width(600).height(600)"**)  
request.parameters = parameters  
request.executeAsync()

* **Aутентикация чрез име и парола**

За хората, които не искат да свързват акаунта си в ЕverydayChef с други свои акаунти, е предложена алтернативата да се регистрират и след това идентифицират чрез име и парола. В този случай тези данни директно се изпращат на сървъра, който да върне цялостната информация за потребеителя, ако той е записан, а ако не е, да върне, че аутентукацията е неуспешна. Това става отново чрез функцията setCurrentUser(), на която се подава само име.

authViewModel.manuallySignIn(username, password)

**fun** manuallySignIn(username: String, password: String) {  
 userRepository.setCurrentUser(username = username)  
}

Успешна аутентикация се следи чрез променливата

**val** authenticationState: MutableLiveData<AuthenticationState> = userRepository.authenticationState

,където AuthenticationState e enum (тип данни, с който променлива може да има стойност една от няколко предварително дефинирани константи):

**enum class** AuthenticationState {  
 UNAUTHENTICATED, *// потребителят не е аутентикиран* AUTHENTICATED, *// потребителят е аутентикиран* INVALID\_AUTHENTICATION *// грешна аутентикация*}

И в LoginActivity се използва функцията observe(), за да засечем, когато променливата се промени:

authViewModel.authenticationState.observe(**this**, Observer**{** authenticationState **->  
 when**(authenticationState){  
 AuthenticationState.AUTHENTICATED -> returnToMain()  
 AuthenticationState.INVALID\_AUTHENTICATION -> showError(**"Invalid Authentication!"**)  
 **else** -> {}  
 }  
**}**)

**private fun** returnToMain() {  
 findNavController(R.id.login\_host).popBackStack()  
 finish()  
}

Регистрацията е имплементирана в RegisterActivity, до където се стига с бутон в LoginActivity. Процесът е подобен на този в LoginActivity, с тази разлика, че полетата са повече на брой и с тях се изисква допълнителна информация от потребителя. След това се изпълнява заявка

@POST(**"user"**)  
**fun** create(@Body body: HashMap<String, String>): Call<User>,

с която се създава нов потребител. При успех, потребителят е върнат на LoginActivity, за да въведе вече създадените си данни и да влезе в приложението.

* + 1. **Семейства**

Поради факта, че това приложение има за основна цел да помага на организацията на готвенето в семейството, всеки потребител има автоматично създадено за него семейство в началото, което носи неговото име. Моделът User съдържа в себе си обект от модела Family, за да е по-лесна връзката между двете.

В ProfileFragment – фрагмент, поддържан от MainActivity, е имплементирана основната функционалност на семействата. Още след създаването на елементите на страницата се попълват данните на текущия потребител, като се следи за промени по време на работа приложението:

**override fun** onViewCreated(view: View, savedInstanceState: Bundle?) {  
 **super**.onViewCreated(view, savedInstanceState)  
 profileViewModel.getCurrentUser().observe(viewLifecycleOwner, Observer**{** it?.let**{** updateUI()  
 **}  
 }**)

...  
По този начин това, което потребителят вижда, ще отразява това, което действително присъства в информацията. Функцията updateUI() е тази, която зарежда данните на потребителя като използва ProfileViewModel, който чрез UserRepository достъпва записаната информация в приложението.

**private fun** updateUI(){  
 user = profileViewModel.getCurrentUser().value  
 user?.let**{  
 val** photoUrl = it.photoUrl  
 **if**(photoUrl != **""**){  
 Glide.with(**this**).load(photoUrl).into(profile\_pic)  
 }**else**{  
 profile\_pic.setImageDrawable(ContextCompat  
 .getDrawable(activity!!.applicationContext, R.drawable.ic\_person))  
 }

...

За зареждането на снимки по техния URL е използван framework(структура на програмна система) на име Glide. При налична снимка на профила, се зарежда тя, а при липса на такава се използва локален ресурс, който да замести снимката.

Ако потребителят е със семейство, което не е автоматично създаденото за него, той може да го напусне, но не може да създава нови такива. Ако семейството е специално генерираното, бутонът за напускане се скрива, а този за създаване на семейства се показва:

**if**(!profileViewModel.isUserWithDefaultFamily()) {  
 btn\_create\_family.visibility = View.GONE  
 btn\_leave\_family.visibility = View.VISIBLE  
}**else** {  
 btn\_leave\_family.visibility = View.GONE  
 btn\_create\_family.visibility = View.VISIBLE  
}

* **Напускане на семейство**

При натискане на бутонът за напускане на семейството(„Leave family”) се показва съобщение, което иска потвърждение от потребителя, че той действително иска да напусне това семейство. Това се случва чрез класът AlertBuilder, който предоставя множество възможности за разработка на съобщения.

**if** (user != **null**) {  
 AlertDialog.Builder(activity)  
 .setTitle(**"Are you sure?"**)  
 .setMessage(**"Do you really want to leave this family? "** +  
 **"You will not be able to return unless they invite you!"**)  
 .setPositiveButton(**"YES"**) **{** dialog, which **->** popupUtility.displayShortDefault(**"Leaving family..."**)  
 leaveFamily()  
 **}** .setNegativeButton(**"NO"**) **{** dialog, which **->** popupUtility.displayShortDefault(**"Canceling..."**)  
 **}**.create().show()  
  
}

Когато се получи потвърждение от потребителя, се извиква функцията в profileViewModel класът leaveFamily(), която извиква тази от UserService:

@DELETE(**"user/{userId}/family"**)  
**fun** leaveFamily(@Path(**"userId"**) userId: Int): Call<Family>

Тя връща стандартното семейство на потребителя, с което е обновяват данните на потребителя в приложението. При успех на заявката:

**val** newCurrentUser = currentUserLd.value!!  
newCurrentUser.user.family = response.body()  
currentUserLd.value = newCurrentUser

По този начин се засича промяна в currentUser и се обновяват и данните на страницата.

* **Добавяне на участници в семейство**

Всеки потребител може да добавя хора към семейството, към което принадлежи. Това става чрез бутона “Add Members“. Отново чрез класа AlertDialog се създава прозорец, на който потребителят трябва да си избере тези, които да покани.

AlertDialog.Builder(*activity*)  
 .setTitle(**"Invite members to family"**)  
 .setMultiChoiceItems(nonMembers, previouslyCheckedUsers )  
 **{** dialog, which, isChecked **->  
 if**(isChecked) checkedUsers.add(which)  
 **else** checkedUsers.remove(which)  
 **}**.setPositiveButton(**"Add Members"**)**{**dialog, which **->  
 profileViewModel**.inviteUsers(checkedUsers)  
 **}**.setNeutralButton(**"Cancel"**)**{**dialog, which **-> }** .create().show()

С функцията setMultiChoiceItems() се пълни прозорецът с имената на всички потребители. Подават й се всички потребители, които не участват в семейството, както и всички, които вече са били поканени. След натискане на бутона “Add Members” на прозореца се изпълнява функцията inviteUsers в ProfileViewModel, където за всеки един избран потребител се вика userRepository.inviteUserToFamily:

**fun** inviteUsers(checkedUsers: ArrayList<Int>) {  
 **for** (checkedUser **in** checkedUsers){  
 **val** userId = **familyNonMembers**[checkedUser].**id  
 userRepository**.inviteUserToFamily(userId, getCurrentUser().*value*!!.**user**.**family**.**id**, **message**)...

При изпращане на покани всички поканени получават известие за това. Когато отидат на страницата с нотификациите(NotificationsFragment), се зареждат всички известия, които са получили:

**notificationsViewModel**.getUserInvitations()  
**notificationsViewModel**.**invitations**.observe(*viewLifecycleOwner*, *Observer***{**  
 **if**(context != **null**){  
 **if**(notifications\_lv.*adapter* == **null**){  
 notifications\_lv.*adapter* = NotificationsDataAdapter(context!!, -1, **it**,  
 **notificationsViewModel**)  
 fragment\_info\_text.*visibility* = View.*GONE* }**else**{  
 (notifications\_lv.*adapter* **as** NotificationsDataAdapter).notifyDataSetChanged()  
 }  
 }  
**}**)

В NotificationsViewModel getUserInvitations, която от своя страна извиква userRepository,userInvitations и се стига до UserService:

@GET(**"user/{userId}/invitations"**)  
**fun** getInvitations(@Path(**"userId"**) userId: Int): Call<ArrayList<Family>

Нотификациите се зареждат и се показват чрез NotificationsDataAdapter – адаптер клас. Адаптер класът служи за връзка между AdapterView (суперклас на ListView, GridView, Spinner и Gallery) и данните, с които то да се напълни. В NotificationsDataAdapter се описва какво да се направи спрям всеки един елемент от подадената колекция от обекти. Това се случва във функцията

**override fun** getView(position: Int, convertView: View?, parent: ViewGroup): View {

…

}

При повторна промяна на списъка с покани, на адаптера на AdapterView обекта му се извиква функцията notifyDataSetChanged(), която съобщава за промяната и се обновява интерфейсът на потребителя.

За отговор на покана се изпраща заявка чрез UserService

@PUT(**"user/{userId}/family/{familyId}/answer\_invitation"**)  
**fun** answerInvitation(@Path(**"userId"**) userId: Int, @Path(**"familyId"**) familyId: Int,  
 @Body body: HashMap<String, Boolean>): Call<User>

Подава и се тяло, в което се записва дали поканата е приета или отказана, в зависимост от това какво е натиснал потребителя.

* **Създаване на семейство**

При натискане на бутонът “Create family” се показва трети прозорец, на който потребителят трябва да въведе име на новото семейство. При натискане на бутона “Create” се извиква profileViewModel.createFamily():

btn\_create\_family.*id* -> {  
 **val** dialogViewRoot = LayoutInflater.from(*activity*)  
 .inflate( R.layout.*dialog\_create\_family*, **null**)  
 AlertDialog.Builder(*activity*)  
 .setView(dialogViewRoot)  
 .setPositiveButton(**"Create"**) **{** dialog, which **->  
 popupUtility**.displayShortDefault(**"Creating new family"**)  
 **profileViewModel**.createFamily(dialogViewRoot.*new\_family\_name*.*text*.toString())  
 **}** .setNeutralButton(**"Cancel"**) **{** dialog, which **->  
 popupUtility**.displayShortDefault(**"Canceled creating family!"**)  
 **}**.create().show()  
}

Този път начина, по който да изглежда прозореца е зададен с допълнителен layout file („файл на оформлението”). Заявката за създаване на семейства се намира този път в FamilyService:

@PUT(**"user/{userId}/family/{familyName}"**)  
**fun** changeUserFamily(@Path(**"userId"**) userId: Int,  
 @Path(**"familyName"**) familyName: String) : Call<Family>

Заявката е от тип Put, защото тя се използва на още едно място за смяна на семейството, но върши същата работа като заявка за създаване на семейство, ако семейстото, с което искаме да сменим не съществува.

При успех, сървърът връща семейството, което е било създадено и с него се обновяват отново данните в currentUser, при което се отразява промяната в интерфейса на потребителя.

* + 1. **Продукти**

Текущата наличност се управлява чрез FridgeFragment. Използва се адаптер клас IngredientDataAdapter, който да пълни продуктите върнати при заявката със FamilyService.

@GET(**"family/{id}/ingredients"**)  
**fun** getFamilyIngredients(@Path(**"id"**) familyId: Int): Call<ArrayList<Ingredient>>

На страницата има още бутони за изтриването на наличните продукти, както и бутон за добавяне на продукти.

При натискане на бутон за изтриване се изпълнява заявката от FamilyService.

@DELETE(**"family/{familyId}/ingredients/{ingredientId}"**)  
**fun** deleteIngredient(@Path(**"familyId"**) familyId: Int,  
 @Path(**"ingredientId"**) ingredientId: Int): Call<ArrayList<Ingredient>>

При успех, сървърът връща променения лист от продукти, с които се обновяват записаните данни в приложението.

За добавяне на продукти към ресурсите на семейството, потребителят може да използва бутона за добавяне. При натискане се използва AlertDialog.Builder, за да се създаде прозорец, в който потребителят да избере наличните продуки. Всички продукти се записват в прозореца чрез функцията setMultiChoiceItems(). Когато се потвърди добавянето на продуктите, за всеки един избран продукт се извиква функцията от FamilyService.

@POST(**"family/{familyId}/ingredients/{ingredientId}"**)  
**fun** addIngredient(@Path(**"familyId"**) familyId: Int,  
 @Path(**"ingredientId"**) ingredientId: Int): Call<ArrayList<Ingredient>>

При успех отново се връща от сървъра обновения списък с продукти, които се зареждат в приложението.

* + 1. **Рецепти**

В CookFragment е имплементирана цялата функционалност на рецептите.

* **Създаване на рецепти**

Върху прозореца за навигация (NavigationDrawer) има бутон, който препраща чрез NavController.navigate() към NewRecipeFragment – фрагмента, отговарящ за създаването на нова рецепта:

btn\_go\_to\_create\_recipe.setOnClickListener**{** *findNavController*().navigate(R.id.*fragment\_new\_recipe*)  
**}**

На страницата на NewRecipeFragment има текстови полета за име на рецепта и описание на стъпките за приготвянето й. Освен тях на страницата има три бутона – за избор на снимка, за избор на нужни продукти и за създаване на рецепта от зададените данни.

При избиране на снимка първо се проверява дали са налични нужните права за достъп до паметта на устройството и ако те липсват се отправя запитване към потребителя да ги даде:

**fun** checkForStoragePermission(activity: Activity): Boolean{  
 **return if** (ContextCompat.checkSelfPermission(activity,  
 Manifest.*permission*.*READ\_EXTERNAL\_STORAGE*)  
 != PackageManager.*PERMISSION\_GRANTED*) {  
 **if** (ActivityCompat.shouldShowRequestPermissionRationale(activity,  
 Manifest.*permission*.*READ\_EXTERNAL\_STORAGE*)) {  
 **messageUtility**.setMessage(**"Permission denied!"**)  
 } **else** {  
 ActivityCompat.requestPermissions(activity,  
 *arrayOf*(Manifest.*permission*.*READ\_CONTACTS*),  
 **RC\_STORAGE\_PERMISSION**)  
 }  
 **false** } **else** {  
 **true** }  
}

След като правата са предоставени се създава Intent с параметър Intent.ACTION\_PICK, за да знае каква операция ще се извършва, задава се тип на файловете, които ще се избират, както и се задават и форматите, в които да се приемат изборите. След това се използва функцията startActivityForResult(), за да се стартира процесът по избор на снимка.

**val** intent = Intent(Intent.*ACTION\_PICK*)  
intent.*type* = **"image/\*"  
val** mimeTypes = *arrayOf*(**"image/jpeg"**, **"image/png"**)  
intent.putExtra(Intent.*EXTRA\_MIME\_TYPES*, mimeTypes)  
startActivityForResult(intent, **newRecipeViewModel**.**GALLERY\_REQUEST\_CODE**)

Резултатът от този процес се връща във функцията onActivityResult, където при успех се получава URI (Уникале идентификатор на ресурс) на избраната снимка от потребителя. След това, от този URI се съставя обект от тип MultipartBody.Part, който се записва в newRecipeViewModel.

**val** file: File = File(getRealPathFromURI(uri))  
**val** requestFile: RequestBody = RequestBody.create(  
 MediaType.parse(*activity*!!.contentResolver.getType(uri)),  
 file  
)  
**newRecipeViewModel**.**selectedImage** = MultipartBody.Part.createFormData(**"picture"**, file.*name*, requestFile)

При натискане на бутона за добавяне на продукти отново чрез AlertDialog.Builder се създава прозорец и чрез функцията setMultiChoiseItems() се попълват всички продукти налични в системата, за да може потребителят да си избере кои да включи в рецептата.

При натискане на бутона “Create” се проверява дали всички полета са запълнени. Ако това е така, чрез RecipeService се изпраща заявка към сървъра

@Multipart  
@POST(**"recipe"**)  
**fun** create(@Part(**"creatorId"**) creatorId: Int, @Part(**"recipeName"**) recipeName: RequestBody,  
 @Part(**"recipeDescription"**) recipeDescription: RequestBody,  
 @Part file: MultipartBody.Part): Call<Recipe>

Тъй като ще се изпраща файл и съпровождаща го информация, се използва анотацията @Multipart. Така се знае, че тялото на заявката е от много различни данни. След като тази заявка е изпълнена успешно се изпращат и въведените продукти като Map oбект чрез заявката:

@PUT(**"recipe/{id}/ingredients"**)  
**fun** updateIngredients(@Path(**"id"**) id: Int, @Body ingredientsBody: Map<String, Array<Int>>) : Call<Recipe>

* **Показване на списъка с рецепти**

Използван е същият подход за събиране и показване на извлечените рецепти както при нотификациите за покани (*т. 3.1.4 – Добавяне на участници в семейство*). За извличането на рецептите се използва FamilyService:

@GET(**"family/{id}/recommended\_recipes"**)  
**fun** getRecommendedRecipes(@Path(**"id"**) familyId: Int): Call<ArrayList<Recipe>>

CookDataAdapter e адаптер класът използван в този случай. За всяка една рецепта се показва нейната снимка, която се зарежда чрез Glide (*като в т. 3.1.4*), името на нейния автор и броя харесвания. При натискане на снимката на рецептата чрез NavController.navigate() се отваря страницата за единична рецепта. Заедно с това обаче се прехвърля и номера на рецептата, за да се знае, коя рецепта се преглежда. Това става чрез поставянето на аргумент при дефиницията на дестинацията в navigation graph компонента:

<fragment ...>

<argument  
 android:name="recipeIndex"  
 app:argType="integer" />

</fragment>

По този начин Android Studio генерира клас, чрез който може този аргумент му се даде стойност преди навигацията:

**val** action = CookFragmentDirections.actionNavCookToRecipeFragment(position)

(context **as** Activity).findNavController(R.id.*nav\_host\_fragment*).navigate(action)

В страницата за единична рецепта чрез адаптер клас (RecipeDataAdapter) се показват и нужните продукти за приготвянето на рецептата, както и стъпките като текст:

recipe\_description.*text* = **recipeViewModel**.**recipe**.**description**recipe\_ingredients\_list.*adapter* = RecipeDataAdapter(context!!, R.layout.*row\_item*,  
 **recipeViewModel**.**recipe**.**ingredients as** MutableList<Ingredient>, **recipeViewModel**)

* **Харесване на рецепти**

В CookFragment може да се харесват и отхаресват рецепти чрез бутонът във формата на сърце в долния десен ъгъл на всяка една рецепта. При създаване на View за всяка рецепта се проверява дали текущият потребител е харесал съответната рецепта и ако да се сърцето се оцветява в червено.

**val** hasUserLiked = **viewModel**.hasUserLiked(recipe)  
**if**(hasUserLiked) {  
 rowView.*recipe\_like\_button*.setImageResource(R.drawable.*heart\_on*)  
}**else**{ rowView.*recipe\_like\_button*.setImageResource(R.drawable.*heart\_off*)  
}  
rowView.*recipe\_like\_button*.setOnClickListener **{  
 viewModel**.rateRecipe(recipe, rowView.*recipe\_like\_button,* position)  
**}**

При натискане на бутона за харесване, той се заключва, за да не може да се хареса рецепта повече от веднъж. След това се изпраща заявка чрез RecipeService за записване на харесването.

@PUT(**"recipe/{recipeId}/user/{userId}/{rate}"**)  
**fun** rateRecipe(@Path(**"recipeId"**) recipeId: Int,  
 @Path(**"userId"**) userId: Int, @Path(**"rate"**) rate: String): Call<Recipe>

Променливата ratе означава дали се записва харесване или се премахва такова. При успешно изпълняване на заявката се сменя снимката на бутона, самият той се отключва, и се обновява MutableLiveData обектът, следящ рецептите.

**if**(isLike) likeButton.setImageResource(R.drawable.*heart\_on*)  
**else** likeButton.setImageResource(R.drawable.*heart\_off*)  
likeButton.*isClickable* = **true  
recipes**.*value*?.set(recipePosition, **it**)  
**recipes**.*value* = **recipes**.*value***recipeRepository**.**recipes** = **recipes**.*value*!!

* + 1. **Списъци с нужни продукти**

Функционалнстта свързана с списъци с нужни покупки е имплементирана в ShoppingListFragment. Основната страница съдържа падащо меню, от където потребителят да избере с кой списък ще работи, самия списък с продукти, бутон за ръчно добавяне на продукти в избрания списък, както и бутон за изтриване на текущия списък.

Със зареждането на страницата се изпраща заявка към сървърва, с която да се вземат всички налични списъци за съответното семейсвто на потребителя.

@GET(**"family/{id}/shopping\_lists"**)  
**fun** getShoppingLists(@Path(**"id"**) familyId: Int): Call<List<ShoppingList>>

Резултатът се записва в променлива от тип MutableLiveData в ShoppingListViewModel класът, за да може да се следи от фрагмента и при наличие на промяна да се изпълнява функцията populateDropDown(), която пълни падащото меню със записаните данни. Това става чрез инстанция на адаптер клас от тип ArrayAdapter, който приема за колекция имената на всички получени списъци, към които е прибавена и стойността “New…”. Toзи адаптер клас се прсвоява на падащото меню.

Допълнителен клас за това падащо меню е и класът ShoppingListItemSelected, където има един метод onItemSelected(), в който се описва поведението на програмата, при натискане на различните обекти в менюто. Ако е избран някой списък, то се извиква функцията changeCurrentShoppingList(), на която й се подава избраният списък, за който вече се взимат всички продукти чрез заявка от ShoppingListService:

@GET(**"shopping\_list/{id}/ingredients"**)  
**fun** getIngredients(@Path(**"id"**) shoppingListId: Int): Call<List<Ingredient>>

Резултатът представлява колекция от всичките продукти, която в последствие се записва в променлива отново от тип MutableLiveData на име currentShoppingListIngredients, от която се пълни списъка с продукти. По този начин веднага щом в променливата се сложи резултатът от заявката, потребителят ще види всичките продукти, които присъстват в списъка.

При избиране от падащото меню на опцията “New…” чрез AlertDialog.Builder се създава прозорец, в който потребителят трябва да въведе име на нов списък с продукти за пазаруване. След като натисне бутона “Create”, се изпраща заявка към сървъра да се създаде новият списък.

@POST(**"shopping\_list/family/{id}"**)  
**fun** create(@Path(**"id"**) familyId: Int, @Body body: Map<String, String>): Call<ShoppingList>

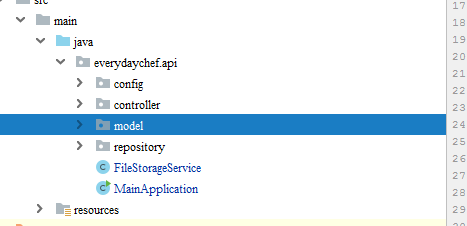
Премахването и добавянето на продукти са имплементирани по метода, използван за същите операции в т. 3.1.5.

* 1. **Част сървър – бекенд**

Частта от приложението, която извършва голяма част от логиката и която се занимава със съхранение и обновяване на данните.

* + 1. **Структура**

Структурата на бекенда е доста подобна на тази на фронтенда:



*Фиг. 3.3 – структура на бекенд*

В папка config се описват конфигурационните файлове, с който се настройва желаното поведение на Spring Boot.

В папка controller са имплементирани всички контролери – класовете, които отговарят за обработването на съответните заявки. Анотират се с **@RestController**, а самите методи - с **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, и като параметър получават URL (Uniform Resource Locator, универсален указател на ресурс), за да се знае коя заявка от кой метод трябва да бъде обработена. За връщане на отговори се използват обекти от класа ResponseEntity, с който отговорът се структурира.

В папка model стоят Entity класовете или класовете, които репрезентират структурата на базата и с които тя може да се манипулира. Класовете се анотират с

**@Entity  
@Table**(name = ...)

Където за параметър на Table се дава името на създадената таблица в действителната база данни. Вътре се описват всички полета, които съществуват в базата, за да е възможно преобразуването от запис в действителната таблица до обект от Entity класът. Релациите са описани чрез няколко различни анотации:

**@ManyToOne – много към едно  
@JoinColumn**(name= **име на колоната, с която се описва релацията**)

**@OneToMany**(mappedBy = **име на колоната от другия клас, с което е описана релацията**) **– едно към много**

**@ManyToMany – много към много  
@JoinTable**(name= **име на свързващата таблица**,

joinColumns = **@JoinColumn**(name= **име на колоната, която описва връзката с тази таблица**),

inverseJoinColumns = **@JoinColumn**(name= **име на колоната, което описва връзката с другата таблица**))

Допълнителни анотации са

**@JsonBackReference**

**@JsonManagedReference**

**@JsonIgnoreProperties**(**списък на имената на изключените колони**),

с които се управлява репрезентацията на обект от този клас в JSON.

В папка repository стоят всички класове, които се използват за директната връзка с базата. Те са отговорни за превеждане на желаните операции до SQL код, с който да се извърши нужната промяна в базата. Това става чрез наследяване на JPARepository, в който са имплементирани най-базовите операции за модулирне на базата(CRUD – create, read, update, delete). За имплементация на допълнителните специфични операции се използва модулът Spring Data JPA, с които заявките се създават на базата на имената на методите. Например:

Boolean existsByName(String name)

Метод, с който се намира дали в базата съществува запис с име name.

Файлът на име StorageService съдържа клас, който е отговорен за записване на файлове на сървъра. Това става чрез метода storeFile(), който приема два аргумента – самия файл и името, под което да бъде записан. Създава се адресът до новия файл чрез функцията resolve(), след което подадения файл се копира на този адрес.

**public void** storeFile(MultipartFile file, String newFileName) {  
 **try**{  
 Path targetLocation = **this**.**fileStorageLocation**.resolve(newFileName);  
 Files.*copy*(file.getInputStream(), targetLocation, StandardCopyOption.***REPLACE\_EXISTING***);  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
}

* + 1. **Логин и регистрация**
    2. **Семейства**

Заявките за семейства са:

**@GetMapping**(**"family"**) – взимане на всички семейства

Използва се предоставеният за това метод от Repository класът (FamilyRepository) –findAll()

**@GetMapping**(**"family/{id}"**) – взимане на едно специфично семейство по id

Използва се методът от FamilyRepository – findAllById(), който обаче враща обект от тип Optional, тъй като може такъв запис в базата да няма. Ако такова семейство съществува се връща то, а ако не се връща статус код 404(Not found – обектът не е намерен)

**@GetMapping**(**"family/{id}/nonmembers"**) – взимане на всички потребители, които не участват в семейството.

Взимат се всички потребители чрез findAll на UserRepository и се премахват участниците в това семейство, за да останат само тези, които имат различни семейства.

Set<User> nonMembers = **new** HashSet(**userRepository**.findAll());  
nonMembers.remove(family.getUsers());  
result = ResponseEntity.*ok*().body(nonMembers);

**@GetMapping**(**"family/{id}/members"**) – взимане на всички участници

Aко е намерено семейство му се извиква методът getUsers().

**@GetMapping**(**"family/{id}/ingredients"**) – взимане на текущата наличност на съответното семейство.

Aко е намерено семейство му се извиква методът getIngredients().

**@GetMapping**(**"family/{id}/recommended\_recipes"**) – препоръчване на рецепти специално за това семейство според наличността.

Имплементиран е Map-Reduce алгоритъм, с който се избират възможните за приготвяне от съответното семейство рецепти и се соритрат според това каква част от продуктите за приготвянето им са налични за семейството. За целта всички рецепти се записват в Map колекция, при която за всяка рецепта се записва колко процента от продуктите присъстват в наличността на семейството. Итерира се през всички рецепти и за всяка рецепта се взима всеки нейн продукт и се проверява дали той присъства в списъка с продукти на семейството. Ако да, променливата с процентите съвпадения се увеличава с 100/(броя на продуктите в рецпетата). След като се проверят всички продукти, рецептата се записва като ключ, а числото получено за процентите съвпадение се записва като стойност в Map елемента, за тази рецепта. След това, тази Map колекция се сортира спрямо процентите съвпадение (стойността) в низходящ ред, като се премахват всички елементи с 0 процента съвпадения.

**public** ResponseEntity<List<Recipe>> getRecommendedRecipes(**@PathVariable** String id){  
 List<Recipe> allRecipes = **recipeRepository**.findAll();  
 **return** findFamilyById(id).map(family -> {  
 List<Recipe> recommendedRecipes = allRecipes;  
 List<Ingredient> familyIngredients = family.getIngredients();  
 **if**(!familyIngredients.isEmpty()) {  
 Map<Recipe, Float> recipeSuitabilityPercentages = **new** HashMap<>();  
 allRecipes.forEach(recipe -> {  
 **float** percentage = 0;  
 **int** numIngredients = recipe.getIngredients().size();  
 **for** (Ingredient ingredient : recipe.getIngredients()) {  
 **if** (familyIngredients.contains(ingredient)) {  
 percentage += 100.0 / numIngredients;  
 }  
 }  
 recipeSuitabilityPercentages.put(recipe, percentage);  
 });  
 System.***out***.println(**"Recipe suitabilty percentages: "** + recipeSuitabilityPercentages.toString());  
 recommendedRecipes = recipeSuitabilityPercentages.entrySet()  
 .stream()  
 .filter(entry -> entry.getValue() > 0)  
 .sorted(Map.Entry.<Recipe, Float>*comparingByValue*().reversed())  
 .map(Map.Entry::getKey)  
 .collect(Collectors.*toList*());  
 }  
 **return** ResponseEntity.*ok*().body(recommendedRecipes);  
 }).orElse(ResponseEntity.*notFound*().build());  
}

**@GetMapping**(**"family/{id}/shopping\_lists"**) – взимат се всички списъци с продукти на съответното семейство

Ако семейството е намерено му се извиква методът getShoppingLists().

**@PostMapping**(**"family/{familyId}/ingredients/{ingredientId}"**) – добавяне на продукти в наличността на семейството

Aко са намерени семейството и продуктът, който трябва да бъде добавен, то в списъка на семейството с налични продукти се добавя продуктът и чрез методът save(), предоставен от FamilyRepository, се записват промените в базата данни.

**@PutMapping**(**"/family/{id}"**) – промяна на семейството

Ако семейството не е намерено се създава ново с подаденото име. Ако е намерено, то му се сменя зададеното име и промяната се записва в базата данни.

**@DeleteMapping**(**"family/{familyId}/ingredients/{ingredientId}"**) – премахване на продукти.

Aко и семейството и продуктът са намерени, то от списъка с налични продукти на семейсвтото се премахва съответния продукт и промените се записват в базата.

* + 1. **Потребители**

**@GetMapping**(**"/user"**) – взимане на всички потребители

Това става по подобен начин на т. 3.2.3

**@GetMapping**(**"/user/{userId}"**) – взимане на специфичен потребител по id

Подобен начин на т. 3.2.3

**@GetMapping**(**"/user/name/{username}"**) – взимане на специфичен потребител по име(username)

Това става с допълнително дефиниран метод в UserRepository, който връща първия срещнат запис в базата с подаденото име. Тъй като всяко потребителско име е уникално, върнатият запис със сигурност ще търсеният от нас потребител.

Optional<User> findByName(String username);

**@GetMapping**(**"/user/{userId}/family"**) – взимане на семейството на потребителя

Ако е намерен съответния потребител се използва методът му getFamily(), който връща семейството, към което принадлежи

**@GetMapping**(**"/user/{userId}/invitations"**) – взимане на поканите на потребителя

Ако е намерен потребителя се използва методът му getInvitations(), който връща всички покани, които са му изпратени.

**@PostMapping**(**"/user"**) – създаване на потребител

Първо се проверява дали подаденото име в тялото на заявката не съществува –дали няма вече такъв акаунт и ако има, се връща грешка. Поради това, че потребител може да се създаде по 3 различни начина (чрез Google, чрез Facebook и локален – чрез име и парола) се пази поле с метода – authenticationMethod, което може да има стойност ‘l’, ‘g’, ‘f’, съответно за локален, Google и Facebook акаунт. Ако паролата не е null (няма стойност), се използва класът BCryptPasswordEncoder, за да се криптира, и така се записва в базата. Ако акаунта не е локален паролата се записва като null.

**@PutMapping**(**"/user/{userId}/family/{familyName}"**) – смяна на семейството на дадения потребител

Ако са намерени съответните семейство и потребител, на потребителя му се извиква функцията setFamily(), а на семейството му се добавя съответния потребител към списъка с участници. Промените се записват чрез UserRepository и чрез FamilyRepository.

**@PutMapping**(**"/user/{userId}/family/{familyId}/invite"**)

Aко са намерени потребителят и семейството, на потребителя му се извиква методът addInvitation(), a на семейството му се добавя този потребител към списъка с поканени потребители.  
  
**@DeleteMapping**(**"/user/{userId}/family"**) – напускане на семейство

Ако е намерен потребителят, му се извиква функцията setDefaultFamily(), която намира автоматично генерираното семейство за този потребител и го присвоява на променливата за семейството на потребителя.

* + 1. **Рецепти**

Голяма част от заявките за рецепти са преместени в FamilyController, тъй като са обвързани със семейството.

Тези, за които се грижи RecipeController са:

**@PostMapping**(**"recipe"**) – създаване на рецепта

Заявка от тип Multipart. Като параметри се получават id на създател, име на рецепта, описание на рецепта, както и снимка на рецепта с тип MultipartFile. Ако съществува съответния потребител и подаденото име на рецептата не е заето вече, се записва файлът на сървъра чрез FileStorageService, с име във формат „{име\_на\_рецепта}\_image.{формат\_на\_файл}“. След това се създава и записва(чрез RecipeRepository) нова рецепта с подадените данни.

String originalImageName = picture.getOriginalFilename();  
**if**(originalImageName != **null**){  
 String newRecipeImageName = recipeName.replace(**" "**, **"\_"**).toLowerCase() +**"\_image"** + originalImageName.substring(originalImageName.indexOf(**"."**));  
 **storageService**.storeFile(picture, newRecipeImageName);  
  
 Recipe recipe = **new** Recipe(recipeName, recipeDescription, **"local:img/"** + newRecipeImageName, user);  
 recipe = **recipeRepository**.save(recipe);  
 **return** ResponseEntity.*ok*().body(recipe);  
}  
**return** ResponseEntity.*badRequest*().body(**"Image was null!"**);

**@PostMapping**(**"recipe/{recipeId}/shopping\_list/{shoppingListId}"**) – добавяне на продуктите на рецепта в списък с продукти за купуване

Ако са намерени съответните рецепта и списък с продукти, към списъка се добавят всички продукти, които участват в рецептата.

shoppingList.getIngredients().addAll(recipe.getIngredients());

**@PutMapping**(**"recipe/{id}/ingredients"**)

**@PutMapping**(**"recipe/{recipeId}/user/{userId}/{rate}"**)

* + 1. **Списъци с продукти**

# ЧЕТВЪРТА ЧАСТ

**Ръководство на потребителя**

**От инсталация до приготвяне на първото ястие**

* 1. **Инсталация**
  2. **Логин и регистрация**
     1. **С EverydayChef акаунт**
     2. **С Google акаунт**
     3. **С Facebook акаунт**
  3. **Семейства**
     1. **Създаване на собствено семейство**
     2. **Добавяне на членове в собствено семейство**
     3. **Присъединяване към семейство**
     4. **Налични продукти на семейство**
  4. **Рецепти**
     1. **Създаване на нова рецепта**
     2. **Редактиране на стара рецепта**
     3. **Оценка и коментари на рецепти**
     4. **Любими рецепти**
  5. **Списъци с нужни продукти**
     1. **Създаване на нов списък**
     2. **Добавяне на продукти в списък**
     3. **Динамично добавяне на продукти в списък//**
  6. **Търсачка на рецепти**
     1. **Търсене по налични продукти**
     2. **Динамично добавяне на продукти в списък**

# ЗАКЛЮЧЕНИЕ

Съдържание

[ПЪРВА ЧАСТ 6](#_Toc34172927)

**[1.1.](#_Toc34172928)****[Защо мобилно приложение](#_Toc34172928)** [6](#_Toc34172928)

**[1.1.1.](#_Toc34172929)****[Мобилни устройства](#_Toc34172929)** [6](#_Toc34172929)

**[1.1.2.](#_Toc34172930)****[Мобилно приложение или мобилен уеб](#_Toc34172930)** [7](#_Toc34172930)

**[1.2.](#_Toc34172931)****[Използвани технологии](#_Toc34172931)** [9](#_Toc34172931)

**[REST технология](#_Toc34172932)** [9](#_Toc34172932)

**[1.2.1.](#_Toc34172933)** [9](#_Toc34172933)

**[1.2.2.](#_Toc34172934)****[Android studio – за разработване на потребителския интерфейс на приложението](#_Toc34172934)** [16](#_Toc34172934)

**[(Frontend)](#_Toc34172935)** [16](#_Toc34172935)

**[1.2.3.](#_Toc34172936)****[Spring boot – за разработване на вътрешната част на приложението](#_Toc34172936)** [17](#_Toc34172936)

**[(Backend)](#_Toc34172937)** [17](#_Toc34172937)

**[1.3.](#_Toc34172938)****[Анализ на приложения с подобно предназначение](#_Toc34172938)** [18](#_Toc34172938)

[](#_Toc34172939) **[Pepperplate](#_Toc34172939)** [18](#_Toc34172939)

[ВТОРА ЧАСТ 21](#_Toc34172940)

**[2.1](#_Toc34172941)****[Изисквания към програмния продукт](#_Toc34172941)** [21](#_Toc34172941)

**[2.2](#_Toc34172943)****[Структура на базата данни](#_Toc34172943)** [22](#_Toc34172943)

[22](#_Toc34172944)

[22](#_Toc34172945)

**[Описание на алгоритъма](#_Toc34172946)** [22](#_Toc34172946)

**[2.3](#_Toc34172947)** [22](#_Toc34172947)

**[2.3.1](#_Toc34172948)****[Логин и регистрация](#_Toc34172948)** [22](#_Toc34172948)

**[2.3.2](#_Toc34172949)****[Семейство](#_Toc34172949)** [23](#_Toc34172949)

**[2.3.3](#_Toc34172950)****[Рецепти](#_Toc34172950)** [23](#_Toc34172950)

**[2.3.4](#_Toc34172951)****[Списъци с нужни покупки](#_Toc34172951)** [24](#_Toc34172951)

**[2.3.5](#_Toc34172952)****[Търсачка на рецепти](#_Toc34172952)** [25](#_Toc34172952)

**[2.4](#_Toc34172953)****[Аргументация за използвани технологии](#_Toc34172953)** [25](#_Toc34172953)

**[2.4.1](#_Toc34172954)****[Android Studio](#_Toc34172954)** [25](#_Toc34172954)

**[2.4.2](#_Toc34172955)****[Spring boot](#_Toc34172955)** [26](#_Toc34172955)

[ТРЕТА ЧАСТ: 28](#_Toc34172956)

**[3.1](#_Toc34172957)****[Част приложение – фронтенд](#_Toc34172957)** [28](#_Toc34172957)

**[3.1.1](#_Toc34172958)****[Структура](#_Toc34172958)** [28](#_Toc34172958)

[30](#_Toc34172959)

[Връзката между компонентите се осъществява чрез “dependency injection”(осигуряване на необходими ресурси за даден обект от друг обект), с помощта на библиотеката “dagger”. 30](#_Toc34172960)

**[3.1.2](#_Toc34172961)****[Навигация](#_Toc34172961)** [30](#_Toc34172961)

**[3.1.3](#_Toc34172962)****[Логин и регистрация](#_Toc34172962)** [33](#_Toc34172962)

**[3.1.4](#_Toc34172963)****[Семейства](#_Toc34172963)** [34](#_Toc34172963)

**[3.1.5](#_Toc34172964)****[Рецепти](#_Toc34172964)** [34](#_Toc34172964)

**[3.1.6](#_Toc34172965)****[Списъци с нужни продукти](#_Toc34172965)** [34](#_Toc34172965)

**[3.1.7](#_Toc34172966)****[Търсачка за рецепти](#_Toc34172966)** [34](#_Toc34172966)

**[3.2](#_Toc34172967)****[Част сървър – бекенд](#_Toc34172967)** [34](#_Toc34172967)

**[3.2.1](#_Toc34172968)****[Логин и регистрация](#_Toc34172968)** [34](#_Toc34172968)

**[3.2.2](#_Toc34172969)****[Семейства](#_Toc34172969)** [34](#_Toc34172969)

**[3.2.3](#_Toc34172970)****[Рецепти](#_Toc34172970)** [34](#_Toc34172970)

**[3.2.4](#_Toc34172971)****[Списъци с продукти](#_Toc34172971)** [34](#_Toc34172971)

**[3.2.5](#_Toc34172972)****[Търсачка на рецепти](#_Toc34172972)** [34](#_Toc34172972)

[ЧЕТВЪРТА ЧАСТ 35](#_Toc34172973)

**[4.1](#_Toc34172974)****[Инсталация](#_Toc34172974)** [35](#_Toc34172974)

**[4.2](#_Toc34172975)****[Логин и регистрация](#_Toc34172975)** [35](#_Toc34172975)

**[4.2.1](#_Toc34172976)****[С EverydayChef акаунт](#_Toc34172976)** [35](#_Toc34172976)

**[4.2.2](#_Toc34172977)****[С Google акаунт](#_Toc34172977)** [35](#_Toc34172977)

**[4.2.3](#_Toc34172978)****[С Facebook акаунт](#_Toc34172978)** [35](#_Toc34172978)

**[4.3](#_Toc34172979)****[Семейства](#_Toc34172979)** [35](#_Toc34172979)

**[4.3.1](#_Toc34172980)****[Създаване на собствено семейство](#_Toc34172980)** [35](#_Toc34172980)

**[4.3.2](#_Toc34172981)****[Добавяне на членове в собствено семейство](#_Toc34172981)** [35](#_Toc34172981)

**[4.3.3](#_Toc34172982)****[Присъединяване към семейство](#_Toc34172982)** [35](#_Toc34172982)

**[4.3.4](#_Toc34172983)****[Налични продукти на семейство](#_Toc34172983)** [35](#_Toc34172983)

**[4.4](#_Toc34172984)****[Рецепти](#_Toc34172984)** [35](#_Toc34172984)

**[4.4.1](#_Toc34172985)****[Създаване на нова рецепта](#_Toc34172985)** [35](#_Toc34172985)

**[4.4.2](#_Toc34172986)****[Редактиране на стара рецепта](#_Toc34172986)** [35](#_Toc34172986)

**[4.4.3](#_Toc34172987)****[Оценка и коментари на рецепти](#_Toc34172987)** [36](#_Toc34172987)

**[4.4.4](#_Toc34172988)****[Любими рецепти](#_Toc34172988)** [36](#_Toc34172988)

**[4.5](#_Toc34172989)****[Списъци с нужни продукти](#_Toc34172989)** [36](#_Toc34172989)

**[4.5.1](#_Toc34172990)****[Създаване на нов списък](#_Toc34172990)** [36](#_Toc34172990)

**[4.5.2](#_Toc34172991)****[Добавяне на продукти в списък](#_Toc34172991)** [36](#_Toc34172991)

**[4.5.3](#_Toc34172992)****[Динамично добавяне на продукти в списък//](#_Toc34172992)** [36](#_Toc34172992)

**[4.6](#_Toc34172993)****[Търсачка на рецепти](#_Toc34172993)** [36](#_Toc34172993)

**[4.6.1](#_Toc34172994)****[Търсене по налични продукти](#_Toc34172994)** [36](#_Toc34172994)

***[4.6.2](#_Toc34172995)*****[Динамично добавяне на продукти в списък](#_Toc34172995)** [36](#_Toc34172995)

[ЗАКЛЮЧЕНИЕ 36](#_Toc34172996)

**ИЗТОЧНИЦИ**

1. Приложение или уеб - <https://www.whizsolutions.co.uk/7-reasons-mobile-apps-better-website/>
2. REST - <https://www.codecademy.com/articles/what-is-rest>
3. “Android Studio” - <http://developer.android.com/tools/studio/index.html>
4. „Spring Boot“ - <https://dzone.com/articles/history-of-spring-framework-spring-boot-framework>