МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет физико-технический

Кафедра теоретической физики и компьютерных технологий

Допустить к защите
Заведующий кафедрой
д-р физ-мат. наук, доцент

__________ В.А. Исаев

________________ 2022 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)

ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИЙ РАСПОЗНАВАНИЯ РЕЧИ В ПРИЛОЖЕНИИ НА ПЛАТФОРМЕ ANDROID

Работу выполнил ______________________Колойтанов Валерий Игоревич

Направление подготовки 09.03.02 Информационные системы и технологии

Направленность (профиль) Информационные системы и технологии

Научный руководитель
канд. техн. наук, доцент _____________________________И. А. Парфенова

Нормоконтролер
преподаватель _______________________________________А. М. Пурунова

Краснодар
2022

РЕФЕРАТ

Выпускная квалификационная работа 60 с., 20 лист., 19 рис., 11 источников.

МОБИЛЬНАЯ РАЗРАБОТКА, БИЗНЕС-ПРОЦЕССЫ, РАСПОЗНОВАНИЕ РЕЧИ, KOTLIN, FIREBASE

Объектом исследования является использование речевых технологии для автоматизации бизнес-процессов.

Предметом исследования является приложение «цифровой официант».

Целью данной выпускной квалификационной работы является разработка мобильного приложения для устного формирования заказа в предприятии общественного питания.

В результате выполнения данной выпускной квалификационной работа разработано мобильное приложение, автоматизирующее процесс оформления заказа на предприятии общественного питания.

СОДЕРЖАНИЕ

Введение………………………………………………………………………. 4

1 Обзор технологий распознавания речи……………………………… 5

1.1 Принцип работы голосового помощника……………………….. 5

1.2 Сравнение различных API для распознавания речи………………. 9

2 Разработка мобильного приложения…………………………………… 12

2.1 Постановка задачи………………………………………………………… 12

2.2 Создание скелета………………………………………………………….. 12

2.3 Модуль Data……………………………………………………………….. 20

2.4 Подключение базы данных Firebase к приложению………………. 24

2.5 Интеграция Speech - to -Text API……………………………………… 30

2.6 Создание формы заказа…………………………………………………. 42

2.7 Обеспечение безопасности внешней инфраструктуры…………… 46

3 Экономический анализ проекта………………………………………… 50

3.1 Предполагаемые затраты на обслуживание инфраструктуры приложения……………………………………………………………………... 50

3.2 Монетизация……………………………………………………………….. 51

3.3 Сравнение с аналогами…………………………………………………. 54

Заключение……………………………………………………………………. 57

Список использованных источников……………………………………... 59

ВВЕДЕНИЕ

Пандемия затормозила развитие многих бизнесов и целых индустрий, но только не сферу разговорного искусственного интеллекта. Глобальный рынок голосовых технологий ежегодно растет на 17,2%, утверждают аналитики Meticulous Research. Ожидается, что к 2025 году его объем достигнет 26,8 млрд долларов. По данным Just AI и Canalys, к концу 2020 года в мире используется 400 млн умных колонок и экранов, а в России – более 1 млн. Растущее покрытие населения смарт-устройствами увеличивает и аудиторию голосовых ассистентов – только у нас в стране, по оценке Just AI, это 52 млн пользователей. Данные технологии находят применение не только в виде голосовых ассистентов. С их помощью автоматизируют call-центры, а также используют в процессе обучения.

Объектом исследования является использование речевых технологии для автоматизации бизнес-процессов.

Предметом исследования является приложение «цифровой официант».

Целью данной выпускной квалификационной работы является разработка мобильного приложения для устного формирования заказа в предприятии общественного питания. Для реализации этой цели поставлены следующие задачи:

– выбор наилучшего API для перевода речи в текст;

– настройка сторонних служб для обеспечения инфраструктуры приложения;

– интеграция API в приложение.

1 Обзор технологий распознавания речи

1.1 Принцип работы голосового помощника

Рассмотрим, как устроены технологии распознавания речи на примере голосового помощника. Работу голосового помощника можно разбить на три этапа:

– автоматическое распознавание речи (ASR): задача транскрибирования аудио;

– обработка естественного языка (NLP): выделение смысла из речевых данных и последующего транскрибированного текста;

– преобразование текста в речь (TTS): перевод текста в человеческую речь для формирования ответа [1].

Некоторые системы ASR зависят от входных данных и должны быть обучены для того, чтобы распознавать определенные слова и речевые шаблоны. По сути, это системы распознавания голоса, используемые в интеллектуальных устройствах. Необходимо произнести определенные слова и фразы прежде, чем голосовой помощник на основе ASR начнет работать, чтобы он научился идентифицировать голос. Другие системы ASR не зависят от данных. Эти системы не требуют никакой подготовки и обладают способностью распознавать произнесенные слова независимо от говорящего. Системы ASR обычно состоят из трех основных компонентов – лексикона, акустической модели и языковой модели – которые декодируют аудиосигнал и обеспечивают наиболее подходящую транскрипцию.

Лексикон – главный винтик механизма расшифровки речи. То насколько лексикон системы глубок определяет производительность и точность распознавателя речи, ведь необходимо учитывать, что некоторые слова могут произноситься несколькими способами. Создание комплексного лексического дизайна для системы ASR несет в себе включение фундаментальных

элементов как разговорного языка (для учета входных данных), так и письменной лексики (текст, который система отправляет). Например, в английском языке слово «read» произносится по-разному в зависимости от того, какое время используется – настоящее или прошлое. Полный лексикон учитывает все возможные фонетические варианты слова. Всесторонние лексиконы особенно важны для обеспечения точности систем распознавания речи, которые имеют большой словарный запас. Лексикон в свою очередь связан с акустической моделью. Акустическое моделирование включает в себя разделение аудиосигнала на небольшие таймфреймы, каждый кадр анализируется на предмет использования различных фонем в этом разделе аудио. Проще говоря, акустические модели направлены на то, чтобы предсказать, какой звук произносится в каждом кадре. Данная модель важна и потому, что разные люди произносят одну и ту же фразу несколькими способами. Такие факторы, как фоновый шум и акценты, могут заставить одно и то же предложение звучать по-разному.

Акустические модели используют алгоритмы глубокого обучения, для определения взаимосвязи между аудио кадрами и фонемами. Широко используемой акустической моделью в ASR является скрытая модель Маркова (HMM), которая основана на модели цепи Маркова – модели, которая предсказывает вероятность события, основываясь исключительно на текущем состоянии. HMM позволяет включать ненаблюдаемые речевые события, такие как часть речи, при определении вероятности того, какие фонемы используются в конкретном аудио кадре.

Современные системы ASR используют языковые модели (NLP), чтобы понять контекст того, что говорит человек. Языковые модели распознают смысл произносимых фраз и используют эти знания для составления последовательностей слов. Они работают аналогично акустическим моделям, используя глубокие нейронные сети, обученные на текстовых данных, чтобы оценить вероятность того, какое слово будет следующим во фразе. Общеязыковой моделью, которую программное обеспечение использует для перевода

устной речи в текстовые форматы для распознавания, является N-граммовая вероятность. N-грамм – это строка слов. Например, «настольная лампа» – это 2-граммовое словосочетание, а «сенсорная настольная лампа» – это 3-граммовое. N-граммовая вероятность, предсказывает следующее слово в последовательности, основываясь на известных предыдущих словах и стандартных правилах грамматики.

Вместе лексикон, акустическая модель и языковая модель позволяют системам ASR делать точные прогнозы о словах и предложениях, в которые произнесены [1]. Для определения точности распознавания речи в системе ASR требуется вычисление частоты ошибок (WER). Формула такова: WER = замены + вставки + удаления / количество сказанных слов=Хотя WER является полезной метрикой, важно отметить, что полезность программного обеспечения для распознавания речи не должна основываться только на этой метрике. Такие переменные, как произношение определенных слов говорящим, качество записи или микрофона говорящего, а также фоновые звуки могут влиять на процент ошибок при распознавании речи. Во многих случаях, даже при наличии упомянутых ошибок, декодированный результат может оказаться ценным для пользователя. Последний этап – формирование и «произнесение» ответа. Теоретически это простая проблема: все, что нужно компьютеру, это огромный алфавитный список слов и детали того, как произносить каждое из них. Для каждого слова понадобится список фонем, которые составляют его звук. Грубо говоря, фонемы для разговорной речи – это то же самое, что буквы для письменного языка: это атомы разговорного звука – звуковые компоненты, из которых можно составить любое произнесенное слово. Если у компьютера есть словарь слов и фонем, все, что ему нужно сделать, чтобы прочитать слово, это найти его в списке, а затем прочитать соответствующие фонемы, верно? На практике это сложнее, чем кажется. Как может продемонстрировать любой хороший актер, одно предложение может быть прочитано по-разному в зависимости от смысла текста, говорящего человека и эмоций, которые они хотят передать (в лингвистике эта идея известна как

просодия, и это одна из самых сложных проблем для синтезаторов речи). В предложении даже одно слово может быть прочитано несколькими способами, потому что оно имеет несколько значений. И даже в пределах одного слова данная фонема будет звучать по-разному в зависимости от фонем, которые приходят до и после нее [2]. Альтернативный подход включает в себя разбивку написанных слов на их графемы (письменные компонентные единицы, обычно состоящие из отдельных букв или слогов, составляющих слово), а затем генерацию фонем, соответствующих им, используя набор простых правил. Это немного похоже на то, как ребенок пытается прочитать слова, с которыми никогда ранее не сталкивался (метод чтения, называемый фонетикой, аналогичен). Преимущество этого заключается в том, что компьютер может сделать разумную попытку прочитать любое слово, будь то реальное слово, хранящееся в словаре, иностранное слово или необычное имя, или технический термин. Недостатком является то, что такие языки, как английский, имеют большое количество «неправильных» слов, которые произносятся совсем не так, как они написаны— именно те слова, которые вызывают проблемы у детей, обучающихся чтению, и людей с так называемой поверхностной дислексией (также называемой орфографической или визуальной дислексией). В итоге текст преобразован в список фонем (последовательность звуков, которые нужно говорить). Но откуда получить основные фонемы, которые компьютер должен читать вслух, когда превращает текст в речь [3]? Существует три различных подхода. Один из них заключается в использовании записей людей, произносящих фонемы, другой заключается в том, чтобы компьютер генерировал фонемы сам, генерируя основные звуковые частоты (немного похожие на музыкальный синтезатор), а третий подход заключается в имитации механизма человеческого голоса.

1.2 Сравнение различных API для распознавания речи

ASR является ядром речевых технологий, но для создания собственной ASR требуется большая вычислительная мощность. Поэтому со временем крупные корпорации начали предлагать системы ASR как сервисы. Каждый сервис имеет уникальную акустическую и языковую модель. Для оценки сервисов ASR в 2019 году тремя бразильскими инженерами Департамента прикладных исследований был проведен эксперимент с 15 мужчинами и 15 женщинами [4]. Для каждого участника обеспечили абсолютную тишину, а затем просили выполнить набор действий в приложении, используя только голосовые команды. Эксперимент состоял из сценария с 20 предложениями, содержащими как общие термины, так и иностранные слова, имена и аббревиатуры. Эксперимент управляется мобильным приложением, разработанным на платформе Android. Это приложение показывает последовательность голосовых команд участнику, а затем собирает речь участника. Голос записывается в формате PCM с одним каналом (моно), глубина звука 16 бит на выборку и частота дискретизации 16000 Гц. После захвата звука приложение отправляет данные на сервер для обработки тремя видами API: Google Cloud Speech-to-Text API, Bing Speech API, IBM Watson Speech to Text.

На рисунке 1 показано линейное изменение значения коэффициента корреляции из каждого предложения и каждого механизма ASR, где 1 означает, что API полностью распознал предложение для всех участников, а 0 означает полный сбой.

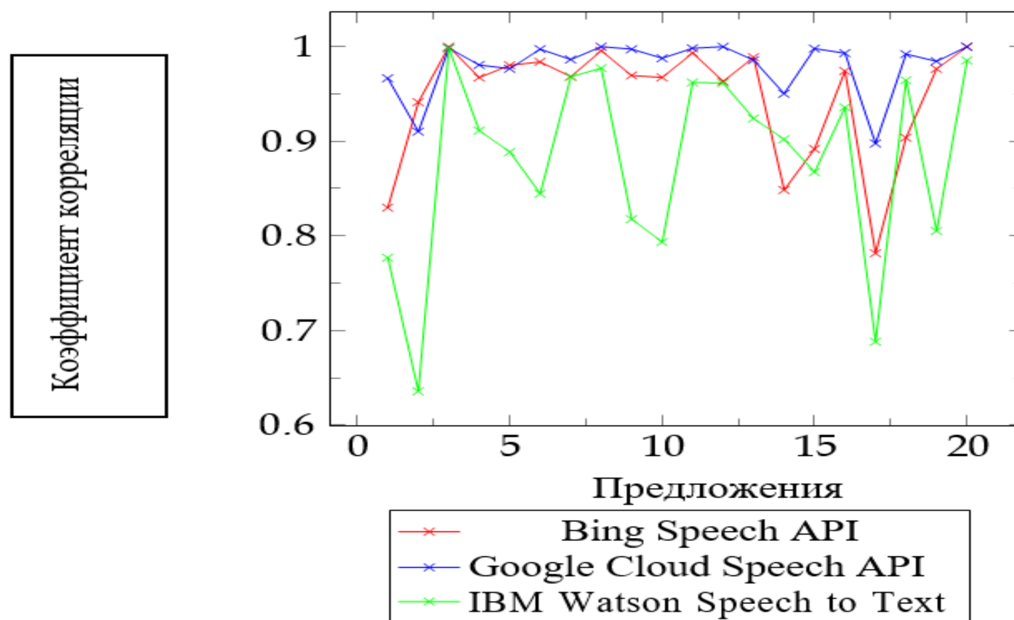

Рисунок 1 – Диаграмма точности распознавания

Оценка правильных ответов каждым механизмом распознавания речи показывает, что Google Cloud Speech-to-Text API имеет явное преимущество, с оценкой 487 правильных предложений из 600, что составляет почти 81%. Вторые лучшие результаты были получены от Bing Speech API с 376 из 600 или близко к 62%. Движок IBM появляется на последнем месте, с 186 из 600 правильных ответов, что составляет около 31%. Критическими факторами для предложений, в которых произошли ошибки распознавания, были наличие заимствованных слов, аббревиатур, названий и специфических терминов. В то время как Google Cloud Speech API распознал все эти факторы по крайней мере один раз, а API Bing Speech иногда не распознавал иностранные слова, IBM Watson Speech to Text не смог распознать различные словосочетания, термины, и названия известных компаний. Кроме того, результаты не показали существенной разницы между мужчинами и женщинами. Тем не менее, более высокая точность Google, несет за собой длительное среднее время распознавания, 7170,3 миллисекунды за предложение. Среднее время IBM составляло 3024,7 миллисекунды, в то время как самым быстрым был движок

Microsoft со средним значением 2659,3 миллисекунды. Другими словами, Google Cloud Speech API был примерно на 4,5 секунды или 269% и 4 секунды или 237% медленнее, чем Bing Speech API и IBM Watson Speech to Text соответственно.

В данном эксперименте были оценены движки ASR Bing Speech API, Google Cloud Speech API и IBM Watson Speech to Text на предмет точности распознавания речи и времени транскрипции. Google Cloud Speech API получил наилучшие результаты по точности и количеству полностью распознаваемых предложений, за ним следуют Bing Speech API и IBM Watson Speech to Text. Основным фактором были именно те аспекты, которые определяют сложность речи. В то время как движки Google и Microsoft получали отличные результаты с иностранными словами, аббревиатурами и именами, инструмент IBM испытывал трудности в распознавании данных речевых конструкций. Что касается производительности для синхронных запросов, то движок Microsoft был самым быстрым. Сразу за ним идет решение IBM и, наконец, более медленным движком стал Google Cloud Speech-to-Text API.

2 Разработка мобильного приложения

2.1 Постановка задачи

Данная задача была предложена компанией, которая занимается автоматизацией бизнес-процессов.

В предприятиях общественного питания сотрудник, принимающий заказ, традиционно записывает позиции из меню в блокнот в бумажном виде и передает заказ на кухню.

С целью оптимизации производственного процесса предлагается разработать функциональное мобильное приложение для устного формирования заказа в предприятии общественного питания (для платформы Android).

Результат работы: мобильное приложение для платформы Android, включающее в себя пользовательский интерфейс, функциональную часть, результативную часть с демонстрацией выполненного заказа. На оборудованной стойке установлен планшет с приложением. Сотрудник предприятия общественного питания, принимая заказ, проговаривает его голосом, приложение принимает и обрабатывает речь, переводит речь в текстовый вид для подтверждения заказа. По окончанию приема заказа данные отправляются на устройство «кухни» предприятия и автоматически встают в очередь заказов. При выполнении заказа он отображается в списке готовых на главном экране приложения.

2.2 Создание скелета

После постановки задачи следует понять структуру приложения и стек необходимых технологий. В качестве языка для написания приложения выбран Kotlin. Это во многом связано с позицией Google. В 2019 году на ежегодной конференции для разработчиков Google I/O, главный графический инженер компании сказал следующее: «Сегодня мы объявляем об еще одном

большом шаге: разработка под Android будет все больше ориентироваться на Kotlin. Многие новые API и функции Jetpack сначала будут предлагаться для Kotlin. Если вы начинаете новый проект, предпочтительнее написать его на Kotlin». Базой данных для приложения выбрана Firebase Realtime Database. Это эффективное решение с малой задержкой для мобильных приложений, которым требуется синхронизация состояний между клиентами в режиме реального времени. За обработку речи отвечает Google Cloud Speech-to-Text API. Эта технология является частью платформы облачных решений Google Cloud. В отличие от встроенной технологии голосового распознавания, Speech-to-Text API может автоматически распознать язык говорящего, воспринимает паузы, не боится шума.

Система сборки Android компилирует ресурсы приложений и исходный код, упаковывает их в APK-файлы или наборы приложений Android, которые можно тестировать, развертывать, подписывать и распространять Android Studio использует систему сборки Gradle, которая предоставляет расширенный инструментарий сборки, для автоматизации и управления процессом, позволяя при этом определять гибкие пользовательские конфигурации [5]. Каждая конфигурация может определять свой собственный набор кода и ресурсов, повторно используя части, общие для всех версий приложения.

Всякий раз, когда в Android Studio создается проект, система сборки автоматически генерирует все необходимые файлы сборки. Данные файлы используют доменный язык (DSL) для определения пользовательской логики сборки и взаимодействия с элементами, специфичными для Android.

Проекты Android Studio состоят из одного или нескольких модулей, являющихся компонентами, которые можно тестировать и отлаживать независимо друг от друга. Каждый модуль имеет свой собственный файл сборки, поэтому выделяют два типа файлов сборки файл сборки верхнего уровня и файл сборки на уровне модуля [5].

Файл сборки верхнего уровня – здесь хранятся параметры конфигурации, общие для всех модулей, составляющих проект. Каждый проект Android

Studio содержит один файл сборки Gradle верхнего уровня.

Файл сборки на уровне модуля – каждый модуль имеет свой собственный файл сборки Gradle, который содержит параметры сборки для конкретного модуля.

В файле build.gradle на уровне модуля можно определить следующие параметры:

– типы сборки определяют свойства, которые Gradle использует при построении и упаковке приложения, и обычно настраиваются для разных этапов жизненного цикла разработки. Например, тип сборки «отладка» включает параметры отладки и подписывает приложение временным ключом, в то время как тип сборки «выпуск» может сжиматься и подписывать приложение ключом для распространения. Для построения приложения необходимо определить хотя бы один тип сборки – Android Studio создает типы сборки отладки и выпуска по умолчанию;

– разновидности продуктов представляют различные версии приложения, которые можно выпустить для пользователей, например, бесплатные и платные версии приложения. Можно настроить разновидности для использования различного кода и ресурсов, а также совмещать ресурсы и повторно использовать части, общие для всех версий приложения. В defaultconfig указываем минимальную версию Android необходимую для запуска – API 23 (Android 6.0), компилируем проект относительно новейшей версии API 31 (Android 12);

– варианты сборки – это перекрестный продукт типа сборки и разновидности продукта, а также конфигурация, используемая Gradle для создания приложения. Используя варианты сборки, можно создать отладочную версию продукта во время разработки или подписанные версии продуктов для распространения. Хотя варианты сборки не настраиваются напрямую, они настраивают типы сборок и разновидности, которые их формируют. Создание дополнительных типов сборки или разновидностей также создает дополнительные варианты сборки;

– зависимости – система сборки управляет зависимостями проекта из локальной файловой системы и удаленных репозиториев. Это избавляет от необходимости вручную искать, загружать и копировать двоичные пакеты зависимостей в каталог проекта.

В данном проекте определены следующие зависимости.

Подписание APK – система сборки позволяет задавать параметры подписи в конфигурации сборки и может автоматически подписывать приложение в процессе сборки. Система сборки подписывает отладочную версию ключом и сертификатом по умолчанию, используя известные учетные данные, чтобы избежать запроса пароля во время построения. Система сборки не подписывает окончательную версию, если явно не определена конфигурацию подписи для этой сборки.

Сокращение кода и ресурсов – система сборки позволяет указать отдельный файл правил ProGuard для каждого варианта сборки. При построении приложения система сборки применяет соответствующий набор правил для сжатия кода и ресурсов с помощью встроенных средств сжатия, таких как R8.

Поддержка нескольких APK – система сборки позволяет автоматически создавать различные APK-файлы, каждый из которых содержит только код и ресурсы, необходимые для определенного размера экрана или двоичного интерфейса приложения (ABI). Однако на данный момент, вместо этого рекомендуется выпустить один AAB, поскольку он предоставляет возможности разделение по языку в дополнение к размеру, экрана и ABI, одновременно уменьшая сложность загрузки нескольких пакетов в Google Play.

Приложение состоит из следующих компонентов:

– Модуль Speech – отвечает за обработку речевых данных;

– Модуль Data – содержит код адаптера и дата-класс модели из базы данных;

– MainActivity – класс, представляющий окно для прорисовки пользовательского интерфейса, взаимодействия с системными компонентами и навигации между экранами приложения;

– Loginfragment – форма входа в приложение;

– Mainfragment – представляет собой стартовый экран, на котором расположены таблица готовых заказов и список сотрудников;

– ConnectivityBroadcastReceiver – проверяет соединение с интернетом;

– Voicefragment – экран восприятия и обработки речи говорящего;

– Orderfragment – экран для оформления заказа и корректировки содержимого.

К проекту может быть подключено неограниченное количество компаний. Для аутентификации компаний введена система аккаунтов. Вся серверная логика авторизации осуществляется с помощью firebase authentication. Пользователи добавляются в проект администратором через консоль Firebase, изображенную на рисунке 2. Аутентификация происходит через форму, содержащую поля «пароль» и «адрес электронной почты».

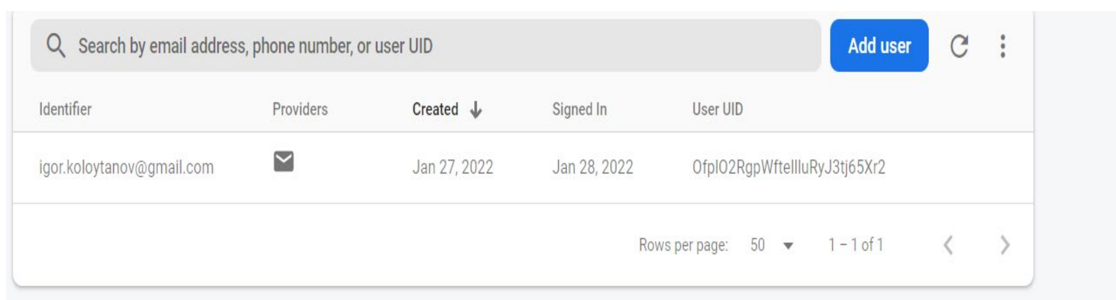

Рисунок 2 – Консоль firebase authentication

Если пользователь есть в списке он переходит к следующему фрагменту, иначе выводится сообщение об ошибке.

Класс Activity является важнейшим компонентом любого Android приложения [5]. Данный класс заменяет метод main который используется в обычных языках программирования. Система Android использует Activity для вызова определенных методов, которые соответствуют ее жизненному циклу. В данном случае класс Activity представляет MainActivity. Эта «активность» (FragmentActivity) является своеобразным холстом, на котором лежат экраны

приложения – фрагменты. Рассмотрим код MainActivity, представленный на листинге 1.

```
class MainActivity : FragmentActivity () {
    private fun hideui() {
        WindowCompat.setDecorFitsSystemWindows(window, false)
        WindowInsetsControllerCompat(window, binding.mainlayout).let {
controller ->
            controller.hide(WindowInsetsCompat.Type.systemBars())
            controller.systemBarsBehavior =

WindowInsetsControllerCompat.BEHAVIOR_SHOW_TRANSIENT_BARS_BY_SWIPE
        }    }
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        installSplashScreen()
        binding=ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        FirebaseApp.initializeApp(/*context=*/this)
        val firebaseAppCheck = FirebaseAppCheck.getInstance()
        firebaseAppCheck.installAppCheckProviderFactory(
            SafetyNetAppCheckProviderFactory.getInstance()
        )
        hideui()
        setEventListener(
            this,
            KeyboardVisibilityEventListener {isOpen ->  if (isOpen)
            {
                hideui()
            }}}
```

Листинг 1 – Класс MainActivity

Жизненный цикл активности начинается с метода onCreate который вызывается при ее создании или перезапуске [5]. Строчка installSplashScreen(), вызывает экран приветствия, созданный с помощью Splashscreens API. Экран представляет собой тему, определенную в ресурсе styles.xml. Поскольку приложение должно существовать самостоятельно, то есть без взаимодействия со шторкой уведомлений или программными кнопками навигации, был использован полноэкранный режим. Для этого в функции hideui было изменено поведение контроллера окна на скрытие системных элементов, а также расширение интерактивных частей экрана. Данная функция вызывается, когда элементы интерфейса берут фокус на себя (клавиатура или диалоговое окно). Функция навигации между экранами выполнена с помощью библиотеки androidx.navigation, которая включает в себя специальный навигационный компонент. Внутри xml файла активности создан специальный тип фрагмента – NavHostFragment который является контейнером для отображения запрашиваемого содержимого. Также следует создать навигационный граф – схему всех возможных путей пользователя при навигации по приложению [5]. Он показан на рисунке на рисунке 3. Когда навигационный граф раздувается, навигационные действия анализируются, и соответствующие объекты NavAction генерируются с конфигурациями, определенными в графе.

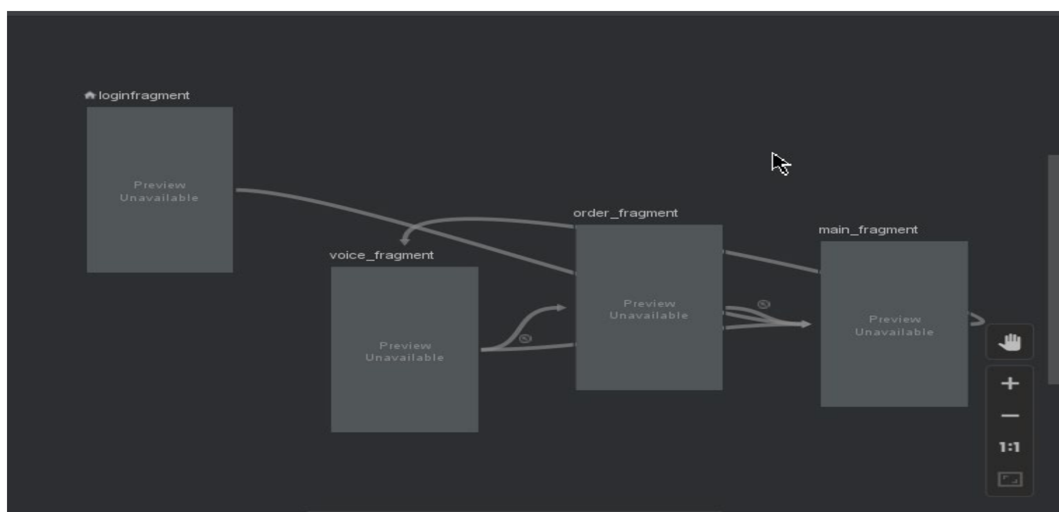

Рисунок 3 – Навигационный граф

Далее необходимо создать класс ConnectivityBroadcastReceiver. Это получатель широковещательных сообщений. Он прикрепляется (регистрируется) к фрагментам с помощью метода registerToFragmentAndAutoUnregister который изображен на листинге 2 и получает сообщения при определенных обстоятельствах, в данном случае, когда срабатывает ConnectivityManager. CONNECTIVITY_ACTION, то есть статус подключения к сети меняется.

```
    @JvmStatic
    fun registerToFragmentAndAutoUnregister(context: Context, fragment: Fragment,
connectionBroadcastReceiver: ConnectivityBroadcastReceiver) {
        val applicationContext = context.applicationContext
        registerWithoutAutoUnregister(applicationContext, connectionBroadcastReceiver)
        fragment.lifecycle.addObserver(object : LifecycleObserver {
            @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
            fun onPause() {
                applicationContext.unregisterReceiver(connectionBroadcastReceiver)
 }
        })
    }


  }
```

Листинг 2 – Метод registerToFragmentAndAutoUnregister

Для того чтобы проверить есть ли у сети доступ в Интернет используется функция isonline (листинг 3). Используя класс Runtime доступ, к которому есть у каждого приложения выполнением команду ping. Тестируем на отклик Dns-сервера Google доступные 99.99% времени. Если процедура завершилась с кодом 0, возвращается истинна, иначе ложь. В методе onReceive, (метод получения сообщений), результат данной функции присваивается переменной

hasConnection которая показывает состояние сети. В дальнейшем данная переменная будет использоваться в интерфейсе onConnectionChanged для определения поведения при определенном состоянии сети.

```
private fun isOnline(): Boolean {
    val runtime = Runtime.getRuntime()
    try {
        val ipProcess = runtime.exec("/system/bin/ping -c 1 8.8.8.8")
        val exitValue = ipProcess.waitFor()
        return exitValue == 0
    } catch (e: IOException) {
        e.printStackTrace()
    } catch (e: InterruptedException) {
        e.printStackTrace()
    }
    return false
}
```

Листинг 3 – Метод isonline

2.3 Модуль Data

Далее перейдём к рассмотрению модуля Data. Данный модуль предназначен для работы с данными, которые приложение получает из Firebase Realtime Database.

База данных для данного приложения состоит из коллекций компаний (обозначенных идентификаторами пользователей), у каждой из которых определены две таблицы – таблица заказов и таблица сотрудников. Фрагмент базы данных изображен на рисунке 4. Рассмотрим каждую из таблиц.

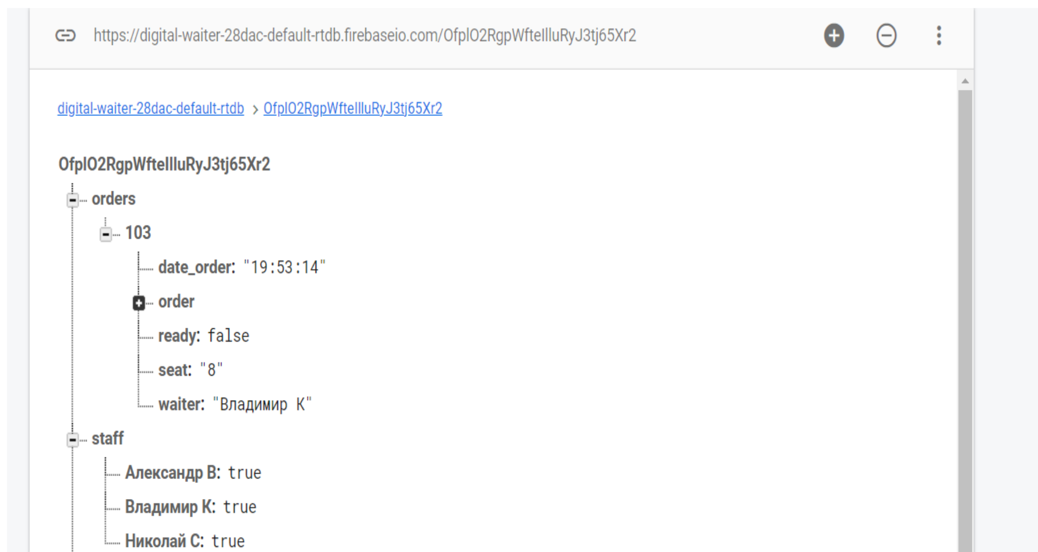

Рисунок 4 – База данных

Определим правила безопасности для базы данных, при которых пользователю доступна только определенная часть данных. Для это используем встроенную переменную $uid, с ее помощью определяем пользователя и даем доступ к чтению и записи для того участка базы данных, в котором определён данный идентификатор ($uid === auth.uid") [6].

На листинге 4 представлен дата-класс элемента в таблице заказов. Из него видно, что основными полями являются столик, для которого оформлен заказ, имя обслуживающего официанта, дата создания заказа и статус готовности. Тег @Exclude показывает игнорирование определенного свойства.

```
data class OrderInfo
  (var seat : String ?= null,
   var waiter : String ?= null,
   var date_order: String?=null,
   @Exclude @set:Exclude @get:Exclude
   var order:  Map<String, Order>?=null,
   var ready: Boolean?=null, )
```

Листинг 4 – Дата-класс OrderInfo

Далее рассмотрим таблицу сотрудники. Firebase Realtime Database не поддерживает массивы и списки в стандартном представлении. Чтобы хранить имена сотрудников используем структуру словарь с шаблоном «имя: true» [6].

Динамические данные в Android приложениях отображаются в специальном контейнере – RecycleView. Для работы RecycleView необходим специальный класс-адаптер [7]. Адаптер связывает данные, в данном случае единицы OrderInfo из списка заказов orderList с элементами RecycleView. Код адаптера списка заказов представлен на листинге 5.

```
class OrderAdapter(private val orderList : ArrayList<OrderInfo>) :
RecyclerView.Adapter<OrderAdapter.MyViewHolder>() {
private lateinit var  mlistner: OnOrderListener
fun setOrdersListener(listener:OnOrderListener)
  {
    mlistner=listener
  }
  interface OnOrderListener {
    fun onOrderClick(position: Int)
  }
  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
    val itemView = LayoutInflater.from(parent.context).inflate(R.layout.order_item,
      parent,false)
    return MyViewHolder(itemView,mlistner)
  }
  override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
    val currentitem = orderList[position]
    holder.seat.text = currentitem.seat
    holder.waiter.text = currentitem.waiter
  }
  override fun getItemCount(): Int {
    return orderList.size
  }
  fun getItem(position: Int): OrderInfo {
    return orderList[position]
  }
```

Листинг 5 – Класс OrderAdapter

Каждый отдельный элемент в Recycleview определяется объектом ViewHolder. Шаблонный элемент списка заказов, в котором находятся данные изображен на рисунке 5.

Рисунок 5 – Элемент Recycleview

Элемент представляет собой класс MyViewHolder, который изображен на листинге 6. В нем определена структура шаблонного элемента – в соответствующих textview отображаются данные. Внутри элемента списка также находится кнопка, поэтому для того, чтобы отслеживать позицию элемента, кнопка которого нажата, был создан «прослушивать» нажатий mlistner. Блок init содержит изначальные действия, которые производятся при инициализации элемента.

```
  class MyViewHolder(itemView : View,listner: OnOrderListener) :
RecyclerView.ViewHolder(itemView){

    val seat : TextView = itemView.findViewById(R.id.seat)
    val waiter : TextView = itemView.findViewById(R.id.waiter)
    val label : TextView = itemView.findViewById(R.id.waiter2)
    val label2 : TextView = itemView.findViewById(R.id.waiter3)

    private val button: MaterialButton=itemView.findViewById(R.id.button)
init {
  label.text="Официант"
  label2.text="Стол"
  button.setOnClickListener {
    listner.onOrderClick(bindingAdapterPosition)
  }}}
```

Листинг 6 – Класс MyViewHolder

Изначально MyViewHolder заполняется шаблонными элементами, не содержащими данных. В методе onBindViewHolder к TextView привязываются необходимо данные элементов из списка.

2.4 Подключение базы данных Firebase к приложению

Теперь перейдем к рассмотрению начального экрана – Mainfragment.

Фрагмент получает данные из базы данных Firebase и отображает их с помощью представлений Chipgroup и Recycleview. ChipGroup используется как контейнер для «чипов» (Chip), которые символизируют профили сотрудников. Зная какой чип активен на текущий момент, можно понять кто из сотрудников оформляет заказ. Чип – это компонент интерфейса, состоящий из иконки и текста, расположенный на прямоугольном фоне с закругленными углами [8].

Для того чтобы подключить Firebase Realtime Database необходимо выполнить следующие действия:

– подключение библиотек com.google.firebase:firebase-database-ktx и com.firebaseui:firebase-ui-database;

– регистрация приложения в консоли Firebase;

– добавление сгенерированного google-services.json в корень модуля проекта.

Также необходимо проверить доступ к чтению и записи.

Перед тем как наполнить Chipgroup и Recycleview данными следует рассмотреть ключевые понятия Firebase Realtime Database. Рассмотрим ключевые понятия Firebase. **Ссылка** представляет определенное место в базе данных и может использоваться для чтения или записи данных в это место. Данные, хранящиеся в базе данных Firebase Realtime, извлекаются путем присоединения асинхронного прослушивателя к ссылке [6]. **Прослушиватель** запускается один раз для начального состояния данных и снова при каждом изменении данных – при этом он может реагировать на несколько различных

типов событий. С помощью **запросов** к базе данных Firebase можно выборочно извлекать данные на основе различных факторов. Чтобы построить запрос в базе данных, надо указать то, как упорядочить данные, используя одну из функций упорядочивания: orderByChild(), orderByKey() или orderByValue(). Затем можно комбинировать их с пятью другими методами для выполнения сложных запросов: limitToFirst(), limitToLast(), startAt(), endAt() и equalTo().

На листинге 7 описан процесс заполнения Chipgroup. Поскольку данные из Firebase поступают динамически необходимо следить за изменениями. Для этого на ссылку query к таблице staff прикрепляется прослушиватель ValueEventListener. Если прослушиватель определяет изменения, сравниваем словари newstaff и staff. Учитывая эти изменения создаем «чипы», которыми заполняем представление Chipgroup.

```
val query = database.getReference(orgid)
          query.addValueEventListener(object :
            ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
              val newstaff = dataSnapshot.child("staff").value as HashMap<String, Boolean>
              val diff: MapDifference<String, Boolean> =
                Maps.difference(staff, newstaff)
              when {
                (diff.entriesOnlyOnRight()
                  .count() != 0) -> {
                  val difference =
                    diff.entriesOnlyOnRight()
                      .toMutableMap()
                  staff.putAll(difference)
                  for (person in difference.keys) {
                    val chip =
                      Chip(this@Mainfragment.requireActivity())
                    chip.isCheckable = true
                    chip.text = person
                    chip.setTextAppearance(R.style.chipText)
```

```
binding.chipGroup.addView(chip)  }
else -> { val difference =  diff.entriesOnlyOnLeft() .toMutableMap()
   difference.keys.forEach { text ->
      binding.chipGroup.children.forEachIndexed { _, view ->
         val chip = view as Chip
         if (chip.text == text) {
            binding.chipGroup.removeView(    view)
            staff.keys.removeIf { x -> difference.containsKey(x)      }


         }         }      }
        }   }    }   }
```

Листинг 7 – Логика наполнения Chipgroup

Для того чтобы определить кто из сотрудников делает заказ необходимо понять какой элемент из Chipgroup нажат. На листинге 8 приведен код описывающий данный процесс. Используем OnCheckedChangeListener. Этот метод возвращает номер активного чипа если такой имеется. После нажатия на «чип» кнопка оформления заказа становится видимой. Чтобы сохранить данные об имени сотрудника, который оформил заказ, для дальнейшего использования применяем структуру набор (bundle). При нажатии на кнопку с помощью менеджера фрагментов создаем набор как результат выполнение данного фрагмента.

```
binding.chipGroup.setOnCheckedChangeListener { _, checkedId ->
              if (binding.chipGroup.checkedChipIds.isNotEmpty()) {
                 val selectedchip: Chip =
                    binding.chipGroup.findViewById(checkedId)
                 val staffVal = selectedchip.text.toString()
```

```
            binding.textButton.visibility = View.VISIBLE
            binding.textButton.setOnClickListener { view ->
                PermissionX.init(activity)
                    .permissions(Manifest.permission.RECORD_AUDIO)
                    .request { allGranted, _, _ ->
                        if (allGranted) {
                            view.findNavController()
                                .navigate(MainfragmentDirections.actionMainFragmentToVoiceFragment())
                            setFragmentResult("waiterkey",
                                bundleOf("data" to staffVal))
                        }
                    }
            }


        } else
            binding.textButton.visibility = View.INVISIBLE

    }
```

Листинг 8 – Определение активного элемента Chip

Следующий элемент – список заказов. Создадим список orderList в котором будут храниться готовые заказ и привяжем его к адаптеру Recycleview. Для того чтобы понять какие из заказов готовы необходимо пройти по снимку данных получаемого с прослушивателя. Эта логика описана в листинге 9. Объекты типа OrderInfo являются детьми по отношению к уникальному номеру заказа, поэтому при итерации используется коллекция snapshot.children. Далее необходимо проверить значение флага готовности и добавить элемент

})

в RecycleView. Для того чтобы изменения отобразились используем adapter?.notifyDataSetChanged().

```
orderRef = FirebaseDatabase.getInstance().getReference("$orgid/orders")
    orderRef.addValueEventListener(object : ValueEventListener {

        override fun onDataChange(snapshot: DataSnapshot) {

            if (snapshot.exists()) {

                orderList.clear()
                for (ordersnapshot in snapshot.children) {

                    if (ordersnapshot.child("ready").value == true) {
                        val order = ordersnapshot.getValue(OrderInfo::class.java)
                        orderList.add(0, order!!)
                        adapter?.notifyDataSetChanged()

                    }

                }
            }
        }

        override fun onCancelled(error: DatabaseError) {
        }

    })
```

Листинг 9 – Логика наполнения RecycleView

Далее рассмотрим логику кнопки внутри элемента списка, которая представлена на листинге 10. Используя описанный в модуле data прослушиватель нажатий, извлекаем объект, который необходимо удалить из базы данных. Для того чтобы найти заказ для элемента, в котором нажата кнопка создаем firebase-запрос – ищем элемент в базе данных по свойству «время оформления заказа» из извлеченного объекта. Определяем ключ родителя по запросу (child.key), удаляем заказ и оповещаем Recycleview.

```
adapter?.setOrdersListener(object : OrderAdapter.OnOrderListener {
    override fun onOrderClick(position: Int) {
        val uid = orderRef.orderByChild("date_order")
            .equalTo(adapter!!.getItem(position).date_order)
        uid.addListenerForSingleValueEvent(object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                for (Order_info in dataSnapshot.children) {
                    val parent = Order_info.key
                    orderRef.child(parent!!).removeValue()
                    adapter?.notifyDataSetChanged()
                    if (adapter!!.itemCount == 1) {
                        orderview.visibility = View.INVISIBLE
                    }
                }
            }

            override fun onCancelled(databaseError: DatabaseError) {}
        })          }       })
```

Листинг 10 – Фрагмент кода удаления элемента из RecycleView

Итоговый вид главного экрана можно увидеть на рисунке 6. Все данные, отображённые в этих представлениях получены из базы данных Firebase описанной ранее.

Рисунок 6 – Экран фрагмента Mainfragment

2.5 Интеграция Speech-to-Text API

Следующий этап – внедрение Speech-to-Text API. API предоставляет три основных метода распознавания речи [9].

Синхронное распознавание (REST и gRPC) отправляет аудиоданные в API для преобразования речи в текст, выполняет распознавание этих данных и возвращает результаты после обработки всего аудио. Запросы на синхронное распознавание ограничены аудиоданными длительностью 1 минута или менее.

Асинхронное распознавание (REST и gRPC) отправляет аудиоданные в API для преобразования речи в текст и инициирует длительную операцию. Используя эту операцию, можно периодически спрашивать о результатах распознавания. Асинхронные запросы аудиоданных ограничены продолжительностью до 480 минут.

Потоковое распознавание (только gRPC) выполняет распознавание аудиоданных, предоставленных в двунаправленном потоке gRPC. Запросы потоковой передачи предназначены для целей распознавания в реальном времени, таких как запись живого звука с микрофона. Потоковое распознавание

предоставляет промежуточные результаты во время захвата звука, что позволяет отображать результат, например, пока пользователь еще говорит. Этот вариант соответствует специфике приложения. Стоит учитывать, что речь разделяется на фрагменты длительностью по 15 секунд каждый. Если длительность аудиоданных не кратна 15 секундам, то идет округление в сторону увеличения. Например, если пользователь проговорил 20 секунд, общее длительность обработанного аудио будет равна 30 секундам.

Перед тем как внедрять API в приложение следует создать учетную запись службы. Учетная запись службы – это особый тип учетной записи Google, предназначенный для представления пользователя, не являющегося человеком, которому необходимо пройти аутентификацию и авторизоваться для доступа к данным в API Google [9]. Форма создания учетной записи изображена на рисунке 7.

Create service account

1 Service account details

Service account name *

Display name for this service account

Service account ... @model-cirrus-335709.iam.gserviceaccount.com

Service account description

Describe what this service account will do

CREATE AND CONTINUE

2 Grant this service account access to project (optional)

3 Grant users access to this service account (optional)

DONE CANCEL

Рисунок 7 – Мастер создания сервисного аккаунта

В настройках созданного сервисного аккаунта, на вкладке ключи, необходимо нажать «создать новый ключ» (рисунок 8). Сгенерированный JSON файл понадобится для подтверждения прав приложения на доступ.

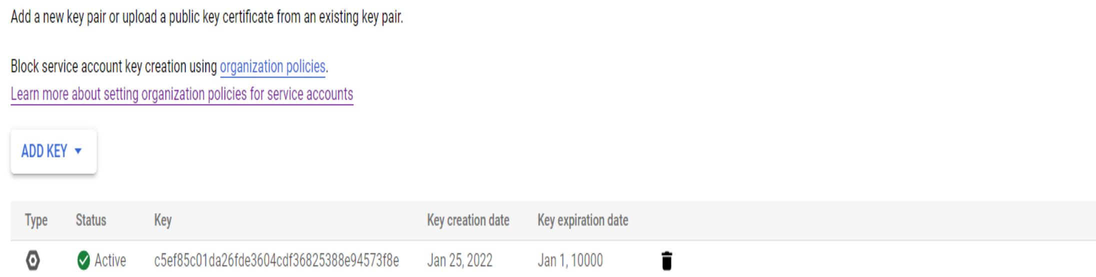

Рисунок 8 – Форма ключей сервисного аккаунта

Теперь перейдем к процессу интеграции. Для этого Google предоставляет клиентские библиотеки. За данные операции отвечает модуль Speech. Он разделен на несколько файлов:

SpeechCredentialsProvider – генерирует объект ServiceAccountCredentials необходимый клиенту из ключа сервисного аккаунта (листинг 11);

```
class SpeechCredentialsProvider(private val context: Context) : CredentialsProvider {
    private val tokenReference: AtomicReference<InputStream> = AtomicReference()
    private val semaphore: Semaphore = Semaphore(0)

    override fun getCredentials(): Credentials {
        val storage = Firebase.storage
        val storageRef = storage.reference
        storageRef.child("credentials.json").stream.addOnSuccessListener {
            tokenReference.set(it.stream)
            semaphore.release()
        }.addOnFailureListener {
        }
```

```
        }.addOnFailureListener {
        }
        semaphore.acquire()
        return ServiceAccountCredentials.fromStream(tokenReference.get())

    }
}
```

Листинг 11 – Класс SpeechCredentialsProvider

VoiceStreamer – отвечает за запись аудио (листинг 12). В данном классе определены объекты voiceRecorder, voiceStreamlistener (прослушиватель аудио данных). Переменная runnableAudioStream – это процесс, в котором в качестве источника данных выбран микрофон, для кодировки выбран алгоритм PCM_16BIT и настроен моноканал. Из микрофона считываются данные и сохраняются в виде массива байт.

```
class VoiceStreamer {
    val sampleRate: Int = 16000
    fun registerOnVoiceListener(voiceStreamListener: VoiceStreamListener?) {
        this.voiceStreamListener = voiceStreamListener
    }
    private val runnableAudioStream = Thread {
            try {
                val buffer = ShortArray(minBufferSize)
                if (voiceRecorder == null) {
                    AudioRecord(
                        MediaRecorder.AudioSource.MIC,
                        sampleRate,
```

```
                channelConfig,
                audioFormat,
                minBufferSize * 10
            ).also {
                voiceRecorder = it
            }
        }
        voiceRecorder?.apply {
            startRecording()
            while (isStreaming) {
                minBufferSize = read(buffer, 0, buffer.size)
                val byteArray = voiceStreamListener?.onVoiceDataAvailable(buffer)
                voiceStreamListener?.onVoiceStreaming(byteArray, byteArray!!.size)
                Log.i("MinBufferSize : ", "${buffer.size}")
            }

        }
    } catch (e: Exception) {
        e.printStackTrace()
```

Листинг 12 – Класс VoiceStreamer

Далее идут методы startVoiceStreaming() и stopVoiceStreaming которые запускает и останавливают процессы прослушивания (листинг 13).

```
fun stopVoiceStreaming() {
    isStreaming = false
    voiceRecorder?.release()
    voiceRecorder = null
```

```
    if (runnableAudioStream.isAlive)
        streamExecutorService.shutdownNow()
}

fun startVoiceStreaming() {
    isStreaming = true
    streamExecutorService.submit(runnableAudioStream)
}
```

Листинг 13 – Методы startVoiceStreaming() и stopVoiceStreaming

SpeechService – класс сервиса обработки речи. В функции initSpeechClient() , описанной на листинге 14, создается клиент(stub) который работает с API через фреймворк gRPC созданный Google. Задаются настройки для SpeechStub: точка входа API, реквизиты учетной записи службы.

```
private fun initSpeechClient() {
    if (speechClient == null) {
        SpeechStubSettings.newBuilder()?.apply {
            credentialsProvider = SpeechCredentialsProvider(this@SpeechService)
            endpoint = "$HOSTNAME:$PORT"
            grpcStub = GrpcSpeechStub.create(build())
        }
        speechClient = SpeechClient.create(grpcStub)
    }
}
```

Листинг 14 – Функция initSpeechClient

Функция createRecognizingRequest() как видно из листинга 15 создает запрос на распознавание речи. Создан поток клиента clientStream который выполняет двунаправленное потоковое распознавание речи и получает результаты при отправке аудио. Ответы от API сохраняем в объект класса ResponseObserver. Настроить параметры распознавания можно через объект StreamingRecognitionConfig который определяет такие параметры как язык распознаваемой речи,контекст,модель системы ASR и тд. В данном случае выбран русский язык, модель для быстрых поисковых команд, возможность прерывания речи (для разделения слов) и получение промежуточных результатов для анализа. Настройки клиента посылаются в поток clientStream. В методе recognize в clientStream посылаются извлеченные ранее аудиоданные. Методы addListener и removeListener управляют списком прослушивателей.

```
private fun createRecognizingRequest(sampleRate: Int) {
    if (speechClient == null) {
        initSpeechClient()
    }
        clientStream = speechClient?.streamingRecognizeCallable()?.splitCall(responseObserver)
        val speechBuilder = SpeechContext.newBuilder()
        speechBuilder.addPhrases("1").boost = 14F
        val streamRequest = StreamingRecognizeRequest.newBuilder()
            .setStreamingConfig(
                StreamingRecognitionConfig.newBuilder()
                    .setConfig(
                        RecognitionConfig.newBuilder()
                            .setLanguageCode("ru-RU")
                            .setEncoding(RecognitionConfig.AudioEncoding.LINEAR16)
                            .setSampleRateHertz(sampleRate)
                            .setModel("command_and_search")
```

```
                    .addSpeechContexts(speechBuilder)
                    .build())
                .setInterimResults(true)
                .setSingleUtterance(false)
                .build()
            )
            .build()

        clientStream?.send(streamRequest)

    }
```

Листинг 15 – Функция createRecognizingRequest

Далее реализуем объект responseObserver, согласно представленному листингу 16. У этого объекта есть несколько компонентов:

– onStart – реакция при начале работы;

– onResponse – реакция при получении ответа. Получаем транскрипт произнесённой речи из первого варианта (самого близкого), передаем значения переменной final и транскрипт в прослушиватель. Эти переменные будут использоваться в фрагменте VoiceFragment. Сохраняем ответ в список;

– onError – реакция при получении сообщения об ошибке;

– onComplete – реакция по завершению работы.

Когда сервис перестает работать (метод onDestroy) посылаем responseObserver сообщение о завершении и прекращаем передачу данных.

```
private val responseObserver: ResponseObserver<StreamingRecognizeResponse> =
    object : ResponseObserver<StreamingRecognizeResponse> {
        var responses: ArrayList<StreamingRecognizeResponse> = ArrayList()
        override fun onStart(controller: StreamController?) {}
```

```
override fun onResponse(response: StreamingRecognizeResponse?) {
    response?.let {
        responses.add(response)
        var text: String? = null
        var isFinal = false
        if (response.resultsCount > 0) {
            val result = response.getResults(0)
            isFinal = result.isFinal
            if (result.alternativesCount > 0) {
                val alternative = result.getAlternatives(0)
                text = alternative.transcript
            }
        }
        if (text != null) {
            for (listener in speechResultListeners) {
                listener.onSpeechRecognized(text, isFinal)
            }                }} }
override fun onError(t: Throwable?) {
    Log.e("errpr", t.toString())
}

override fun onComplete() {
   for( response in responses)
   { if (response.hasTotalBilledTime())
   {
       database.setValue(ServerValue.increment(-(response.totalBilledTime.seconds)))
   }  }}
```

Листинг 16 – Объект responseObserver

Перейдем к Voicefragment. В словаре ordermap хранится структура заказа, foodname – название блюда, foodcount – количество порций, responseCount – переменная в зависимости, от которой на экране меняется текст. Когда фрагмент начинает работу к нему прикрепляется speechservice (метод onStart), это отражено на листинге 17. Данный сервис запускает запись голосового потока mVoiceRecorder если это допустимо.

```
override fun onStart() {
    super.onStart()
    Intent(this.requireActivity(), SpeechService::class.java).also { intent ->
      activity?.bindService(
        intent,
        mServiceConnection,
        Context.BIND_AUTO_CREATE
      )      }
    database.addListenerForSingleValueEvent(object :
        ValueEventListener {
        override fun onDataChange(dataSnapshot: DataSnapshot) {
            time = dataSnapshot.value as Long
          if (time>0)
              startVoiceRecorder()
           else
               binding.response.text="Превышен лимит"
        }
        override fun onCancelled(error: DatabaseError) {
        }         })
```

Листинг 17 – Метод жизненного цикла onStart

Каждый раз при успешном распознавании сервисом аудиоданных во фрагменте вызывается метод прослушивателя – onSpeechRecognized.Его код можно увидеть на листинге 18. Если фраза распознана окончательно, то есть переменная final истина, определяется какой тип данных ожидаем сейчас (число или произвольная строка) – при необходимости уведомляем пользователя об ошибке. Если после двух срабатываний образовалась пара «название блюда - количества» добавляем ее в словарь ordermap. Повторяем алгоритм вплоть до нажатия кнопки «стоп». После нажатия приложение переходит к фрагменту Orderfragment, сохранив ordermap для дальнейшего использования.

```
val speechListener = object : SpeechListener() {
    override fun onSpeechRecognized(text: String, final: Boolean) {
        when (final) {
            true -> {
                when  {
                    (text.trim().toIntOrNull()== null)&&  ( responseCount % 2 == 0)->
                    { foodName = text
                        responseCount++
                    } (text.trim().toIntOrNull()!= null)&&  ( responseCount % 2 != 0)->
                    {  foodCount = text.trim().toIntOrNull()!!
                        responseCount++
                     }  else ->activity?.runOnUiThread {
                        binding.listenTextview.text = "Ошибка.Попробуйте еще раз" } }
                activity?.runOnUiThread { responseChange() }
                if ((foodName != "") && (foodCount != 0)) {
                    ordermap[foodName] = foodCount
                     activity?.runOnUiThread {
                        binding.listenTextview.text = "Добавлено"
                        binding.response.text = "Скажите название блюда"}                }
```

```
false -> {
activity?.runOnUiThread {
binding.listenTextview.text = "Слушаю }
 foodCount = 0
 foodName + ""              }          }}}
```

Листинг 17 – Объект speechListener

При нажатии на кнопку момент срабатывает метод onStop, код представлен на листинге 18, жизненного цикла фрагмента VoiceRecorder – прекращается запись голоса, поток останавливается, сервис отключается.

```
override fun onStop() {
    super.onStop()

    mSpeechService?.removeListener(speechListener)
    mSpeechService = null
    activity?.unbindService(mServiceConnection)
    stopVoiceRecorder()

}
```

Листинг 18 – Метод жизненного цикла onStop

В итоге получаем экран Voicefragment который реагирует на речь пользователя и дает ему подсказки на дальнейшие действия. Одно из состояний экрана, когда официант произносит названия и количества блюд представлен

на рисунке 9. Другие состояния уведомляют пользователя об ошибке или «слушают» речь.

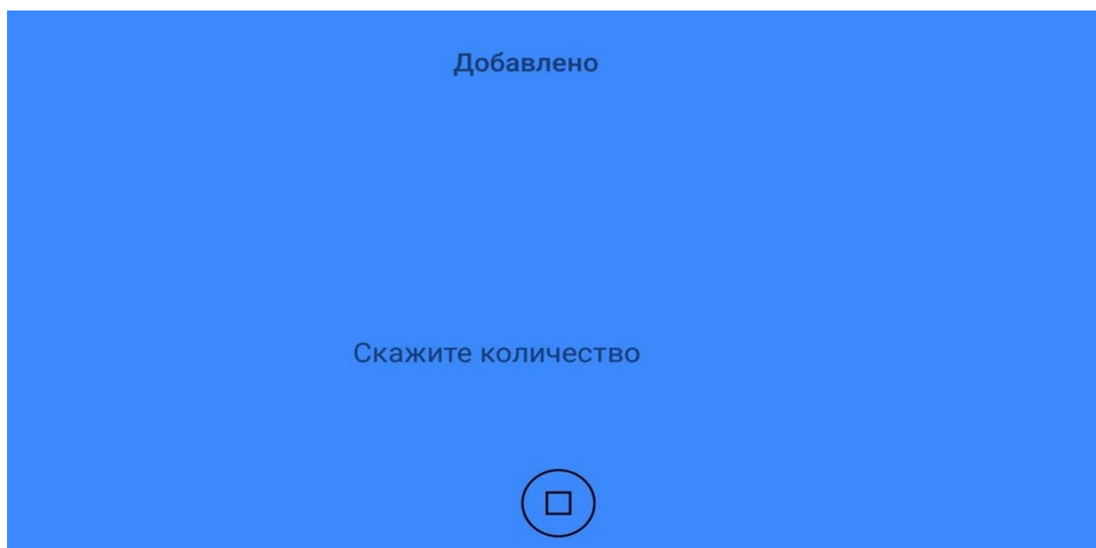

Рисунок 9 – Экран фрагмента Voicefragment

2.6 Создание формы заказа

Последний фрагмент, Ordefragment, имеет два ключевых элемента – тулбар-меню и таблицу tableLayout с содержимым заказа в виде формы. При создании фрагмента извлекаем сохранённые ранее переменные ordermap и waiterkey. Чтобы создать форму необходимо динамически добавлять строки в таблицу. Для этого используем layoutInflater. Попарно проходим по словарю и расширяем tableLayout новыми строками. Сохраняем количество полей в таблице в переменной validationSuccess. Данный процесс показан на листинге 19.

```
tableLayout = binding.orderForm
val ordersMap = arguments?.get("order") as Map<String, Int>
val tableRow = TableRow(activity)

for ((key, value) in ordersMap) {
   val row =
      activity?.layoutInflater?.inflate(R.layout.form_pair, tableRow, false) as TableRow
   (row.findViewById<View>(R.id.name) as TextInputEditText).setText(key)
   (row.findViewById<View>(R.id.count) as TextInputEditText).setText(value.toString())
   tableLayout.addView(row)
}
validationSuccess = tableLayout.childCount
```

Листинг 19 – Заполнение tableLayout

Меню состоит из кнопки принятия заказа которая перебрасывает пользователя на главный экран и отмены заказа – в этом случае пользователь видит предупредительный диалог. Логика меню находится setOnMenuItemClickListener и разделена по названию элементов – action_send (подтверждение заказа) и action_discard (отмена заказа). Её код представлен на листинге 20.

```
binding.myToolbar.setOnMenuItemClickListener {
   when (it.itemId) { R.id.action_send -> {
ConnectivityBroadcastReceiver.registerToFragmentAndAutoUnregister(
      this.requireContext(), this,object : ConnectivityBroadcastReceiver() {
         override fun onConnectionChanged(hasConnection: Boolean) =
```

```
if (!hasConnection) { Toast.makeText(
    activity, "Проблемы сокдинения с сервером. Проерьте соединение и попробуйте снова",Toast.LENGTH_SHORT
).show() } else {
val database = Firebase.database("https://digital-waiter-28dac-default-rtdb.firebaseio.com/")
ordersRef=database.reference.child( Firebase.auth.currentUser?.uid+"/orders")
val order = getFormContent()
val orderNumber = (0..10000).random()
when (validationSuccess) { 0 -> {ordersRef .addListenerForSingleValueEvent(object :ValueEventListener {override fun onDataChange(dataSnapshot: DataSnapshot) { ordersRef.child(orderNumber.toString()).setValue(mapOf(
        "order" to order, "seat" to position.editText!!.text.toString(),
        "ready" to false,
    "date_order" to LocalTime.now().truncatedTo(ChronoUnit.SECONDS).toString(),
        "waiter" to staffKey) ) }
      override fun onCancelled(error: DatabaseError) {
    }}) view.findNavController()
.navigate(OrderfragmentDirections.actionOrderFragmentToMainFragment()) }
  else -> {}}} }) true}
R.id.action_discard -> {val dialog = AlertDialog.Builder(this.activity)
    .setTitle("Отменить")
    .setMessage("Вы действительно хотите отменить заказ?")
    .setPositiveButton(android.R.string.yes) { _, _ ->
        view.findNavController()
          .navigate(OrderfragmentDirections.actionOrderFragmentToMainFragment())}
      true} else -> false }}
```

Листинг 20 – Логика меню

Поскольку у пользователя есть возможность корректировки данных, сгенерированных сервисом обработки аудио данных, необходимо внедрить механизм валидации перед отправкой заказа в базу данных. Она происходит

в методе getFormContent. Сначала создается валидатор с несколькими правилами: число не может быть дробным, отрицательным, оба поля не должны быть пустыми). Таблица разделяется на строки, из каждой строки извлекаются значения пары «название: количество» и проверяются на соответствие правилам. При каждой успешной валидации поля счётчик validationSuccess снижается на единицу. Если он равен 0 – можно составлять заказ и отправлять данные иначе сообщаем о некорректном заполнении. Каждому заказу присваивается уникальный номер orderNumber и присваиваются следущие поля:

– **order** – итоговое значение формы заказа (переменная order);

– **seat** – стол заказа;

– ready – флаг готовности (изначально ложь);

– **date_order** – время создания заказа- то есть текущее время. Определяется с помощью класса LocalTime;

– **waiter** – имя официанта, принимающего заказ (переменная staffkey).

Пример экрана при заказе из трех позиций показан на рисунке 10. Данная форма является полностью интерактивной, это означает что текстовые поля можно редактировать при необходимости если была допущена ошибка.

Рисунок 10 – Экран фрагмента Oderfragment

Связь кухни с приложением может быть реализована разными способами. Например, при помощи одностраничного веб-приложения (рисунок 11). Firebase предоставляет Firebase JS SDK, для интеграции с веб-приложениями. Принципы работы с базой данных не отличаются от тех, что были описаны ранее. На рисунке представлена динамическая таблица заказов, ожидающих исполнения. После подтверждения готовности нажатием галочки, заказ отображается в RecycleView главного экрана мобильного приложения.

Официант	Время	Заказ	Готовность
Владимир К	15:26:49	паста Карбонара 1 паста с лососем 2 стейк из свинины 2	▢

Рисунок 11 – Приложение кухни

2.7 Обеспечение безопасности внешней инфраструктуры

После окончания работы над логикой приложения необходимо позаботиться о безопасности внешних компонентов. Со стороны Firebase это можно сделать с помощью технологии AppCheck. AppCheck помогает защитить серверные ресурсы от злоупотреблений, таких как перегрузка базы данных и фишинг [6]. Данная технология работает как со службами Firebase, так и сторонними серверами, чтобы обеспечить безопасность ресурсов. С помощью App Check устройства, на которых выполняется ваше приложение, будут использовать приложение или поставщика аттестации устройств, который подтверждает одно или оба из следующих условий:

– запросы исходят от вашего аутентичного приложения;

– запросы исходят от аутентичного, нетронутого устройства.

Для Android провайдером AppCheck является Safety Net, в частности, SafetyNet Attestation API (API аттестации SafetyNet) – это API защиты от злоупотреблений, который позволяет разработчикам приложений оценивать устройство Android, на котором работает их приложение. API следует использовать как часть вашей системы обнаружения злоупотреблений, чтобы определить, взаимодействуют ли ваши серверы с вашим подлинным приложением, работающим на подлинном устройстве Android. Ядром API является криптографически подписанное свидетельство, оценивающую целостность устройства. Чтобы создать свидетельство, API исследует программную и аппаратную среду устройства, выискивая проблемы с целостностью и сравнивая ее со справочными данными для одобренных устройств Android. Созданное свидетельство привязывается к одноразовому номеру, предоставляемому вызывающим приложением, а также содержит метку времени генерации и метаданные о запрашивающем приложении. API аттестации SafetyNet работает по следующему принципу:

– API аттестации SafetyNet получает вызов от приложения. Этот вызов включает в себя одноразовый номер;

– SafetyNet оценивает среду выполнения и запрашивает подписанную аттестацию результатов оценки с серверов Google;

– Серверы Google отправляют подписанное свидетельство в службу аттестации SafetyNet на устройстве;

– Служба аттестации SafetyNet возвращает это подписанную свидетельство приложению;

– Приложение перенаправляет подписанную аттестацию на сервер;

Этот сервер проверяет ответ и использует его для принятия решений по борьбе со злоупотреблениями. сервер передает свои результаты приложению.

Далее нужно включить обязательную проверку токена App Check для компонента Realtime Database. Как видно из рисунка 12 запросы разделяются на несколько типов. Верифицированные запросы содержат в себе валидный токен AppCheck, им разрешен доступ к базе данных. Далее идут три типа не валидных запросов: запросы от устаревшего клиента (не обновлен Firebase SDK), некорректная форма запроса, запросы от неаутентичного приложения (эмулятор или root-права).

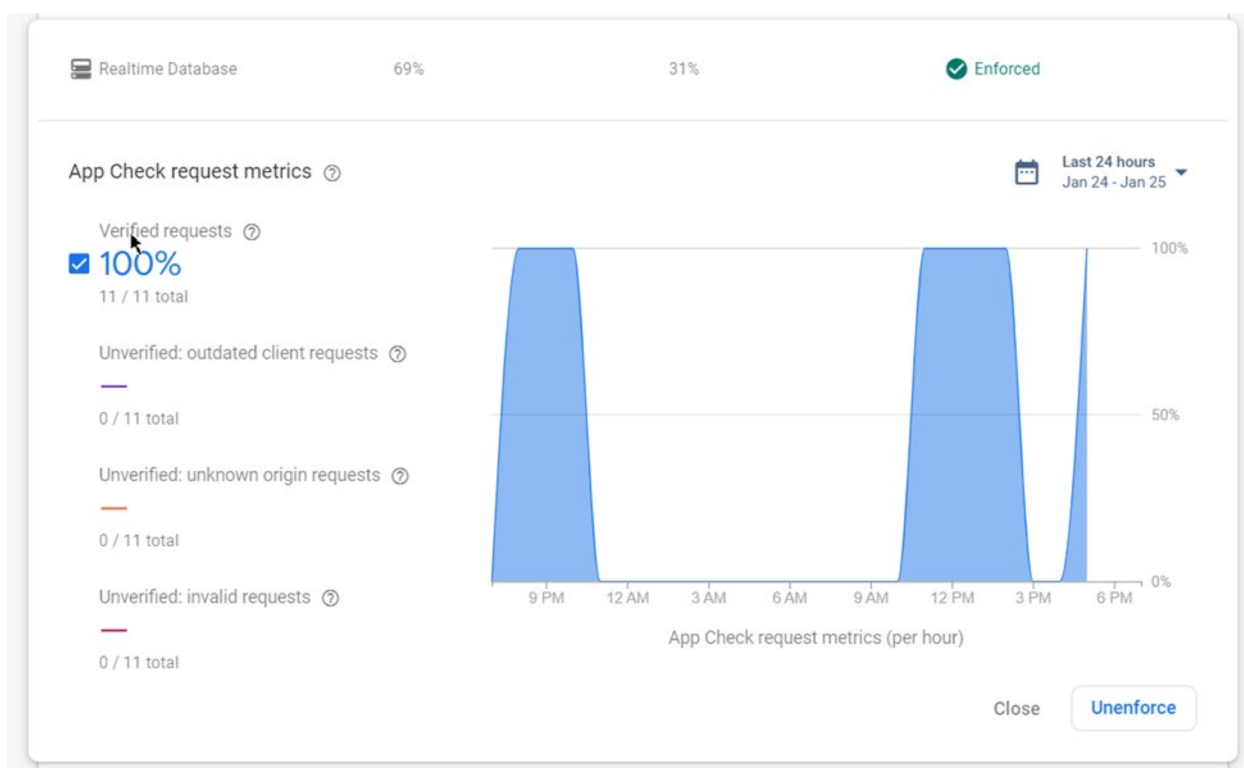

Рисунок 12 – Диаграмма запросов AppCheck

Уязвимым местом проекта Google Cloud является сервисный аккаунт [9]. Для начала необходимо удостовериться что при утечке ключевые ресурсы проекта не будут скомпрометированы. Для этого необходимо предоставить аккаунту службы минимально необходимые права и минимизировать количество пользователей, которые могут его использовать. Поскольку вызов API

выходит за пределы ролей Google Cloud, роль, данная сервисному аккаунту, не имеет значения- вызов все равно произойдет, поэтому назначаем минимальную роль «Viewer» не позволяющую редактировать какие-либо ресурсы проекта. Таблица пользователей проекта и их ролей изображена на рисунке 13.

Type	Principal ↑	Name	Role	Security insights	Inheritance
	igor.koloytanov@gmail.com	master fix	Owner	4799/5047 excess permissions	
	speechclient@model-cirrus-335709.iam.gserviceaccount.com	SpeechClient	Cloud Speech Client	1/1 excess permissions	

Рисунок 13 – Таблица ролей

Кроме того, ключ сервисного аккаунта хранится не в приложении, а в защищенном хранилище Firebase Cloud Storage, с которого файл считывается как поток данных. Если сервисный ключ все же станет общедоступным, то администратор сгенерирует новый и обновит файл в хранилище, без необходимости замены приложения.

3 Экономический анализ проекта

3.1 Предполагаемые затраты на обслуживание инфраструктуры приложения

В бизнесе очень важно рассчитать затраты. При интеграции данного приложения необходимо учитывать, что оно работает за счет нескольких сторонних сервисов, у которых есть своя ценовая модель.

Firebase – сервис, на серверах которого хранится, база данных проекта предоставляет тариф «Blaze», где ежемесячная плата рассчитывается по мере использования. Для упрощения расчетов Google предоставляет специальный калькулятор (рисунок 14). Из калькулятора видно, что бесплатно предоставляется 1 гигабайт хранилища и 10 гигабайт траффика между базой данных и приложением. Учитывая специфику проекта данных ресурсов, хватит на обслуживание нескольких компаний.

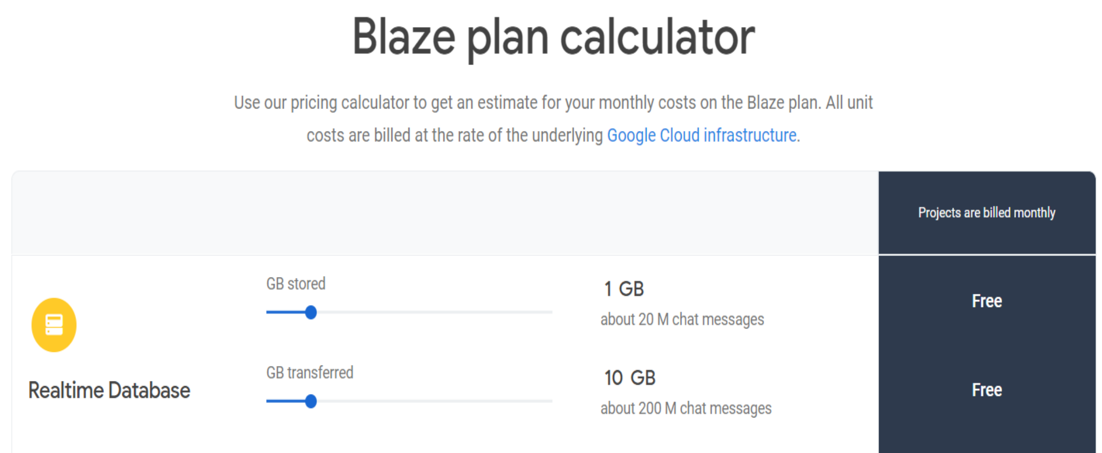

Рисунок 14 – Калькулятор тарифов Firebase

Следующий компонент – Speech-to-Text API отвечающий за обработку аудиоданных. За один календарный месяц можно обработать 1 час аудиоданных бесплатно. Далее необходимо доплачивать, ежемесячная плата за API зависит от типа использованной модели и количества минут использовании. На рисунке 15 отображает расценки для проекта. Если учесть конфигурацию, используемую в данном приложении, на данный момент плата за минуту обработанного аудио составляет 1.77 рублей.

Pricing for My Billing Account

Filter Enter property name or value

Google service	Service description	Service ID	SKU ID	SKU description	Product taxonomy	Unit description	Per unit quantity	Tiered usage start	List price (₽)
GCP	Cloud Speech API	63DE-82AB-F564	60AE-2FE3-C3D8	Cloud Speech-to-Text Audio Length Standard	GCP > Cloud AI and Industry Solutions > Cloud Speech API > Other	minute	1	0	0.00
GCP	Cloud Speech API	63DE-82AB-F564	60AE-2FE3-C3D8	Cloud Speech-to-Text Audio Length Standard	GCP > Cloud AI and Industry Solutions > Cloud Speech API > Other	minute	1	60	1.775956799

Рисунок 15 – Тарифы ресурсов проекта

3.2 Монетизация

Далее разберем стратегию монетизации приложения. Для каждого клиента будут созданы отдельные проекты Google Cloud и Firebase. Разработчик назначается администратором. Приложение будет монетизироваться, используя модель подписки на определенные тарифы. Тарифы зависят только от количества заказов, интеграция и техподдержка предоставляются в каждом тарифе. Если считать один заказ равен одной минуте, то можно сформировать следующие тарифы:

Тариф «Lite» – подходит для маленьких компании (30 заказов в день, 900 минут в месяц);

Тариф «Pro» – подходит для средней компании (60 заказов в день, 1800 минут в месяц);

Тариф «Pro+» – предоставляется компаниям с высоким потоком клиентов, которым недостаточно опции «Pro». Исходя из потребностей предприятие может запросить количество минут.

Для того чтобы отслеживать затраты используем следующий механизм, предоставляемый Google Cloud. Бюджет – позволяет отслеживать фактические расходы Google Cloud по сравнению с запланированными расходами. После установки суммы бюджета необходимо задать пороговые значения оповещений о бюджете, которые используются для запуска уведомлений по электронной почте. Электронные письма с оповещениями о бюджете помогут оставаться в курсе того, как расходы отслеживаются в соответствии с бюджетом. На рисунке 16 изображен мастер создания бюджета.

Рисунок 16 – Мастер создания бюджета

Если траты превысят обозначенные в настройках бюджета пороги, в данном случае 50%,90%,100%, пользователи получат сообщение на электронную почту, пример которого показан на рисунке 17.

Google Cloud ОТКРЫТЬ КОНСОЛЬ

Оповещение об использовании бюджета

Достигнуто пороговое значение в 50% от бюджета

Платежный аккаунт: My Billing Account 19.01.2022 11:19

Размер бюджета	Бюджетный период
₽500.00	01.01.2022 – 31.01.2022

Расходы в вашем платежном аккаунте "My Billing Account" достигли 50% от бюджета в ₽500.00 за бюджетный период с 01.01.2022 до 31.01.2022.

Название бюджета

Plan a

Идентификатор платежного аккаунта

012846-2989DB-DEBA05

Посмотреть данные о расходах можно на странице "Отчеты".

ПОСМОТРЕТЬ ПОДРОБНЫЕ СВЕДЕНИЯ О БЮДЖЕТЕ

Рисунок 17 – Сообщение о превышении порога бюджета

Но как же ограничить использование API при истощении бюджета? Для этого введем поле timeleft для каждого клиента. Установим определенное значение в зависимости от используемого тарифа. С помощью totalbilledtime которое показывает сколько секунд данных обработано за запрос, то есть с учетом округления определим на сколько единиц необходимо уменьшить значение поля timeleft. Если значение меньше или равно нулю отказываем в создании запроса на распознавание.

3.3 Сравнение с аналогами

На рынке существует несколько приложений, решающих задачу оптимизации коммуникации между официантами и кухней. Например, приложение «Мобильный официант» от фирмы «Группа компаний ККС». Внешний вид приложения архаичен и неинициативен, часто ключевые функции такие как выбор стола, навигация по плану заведения перестают работать. В отзывах, представленных на рисунке, упоминается огромное количество багов в работе системы. Последний раз приложение обновлялось в 2018 году, что говорит о нежелании разработчика поддерживать продукт и исправлять ошибки в работе приложения.

Кроме этого, данное приложение не может работать самостоятельно и зависит от ресторанной системы r_keeper [10]. В данном случае рассматривается облачная версия r_k Cloud. Как видно из сравнительной таблицы, изображённой на рисунке 18, модуль «Мобильный официант» предоставляется только в максимальной опции. Это сильно ограничивает предприятие, оно вынуждено пользоваться сервисами единой экосистемы, которые могут не отвечать нуждам компании.

	r_k Lite Фудтрак, кофепоинт	r_k Cloud Start Маленькие рестораны и сети (1–5)	r_k Cloud Pro Рестораны и сети	r_k Cloud Max Крупные рестораны и сети
Складские операции				
Управление складом				
Автоматизация документооборота				
Доставка	1 200 ₽ / мес.	3 000 ₽ / мес.	5 000 ₽ / мес.	7 200 ₽ / мес.
Прием, обработка заказов, контроль изменения статусов	−	−	+	+
Создание сайта доставки	−	−	−	+
Автоматизация приема заказов (телефон, касса, сайт)	−	−	−	+
Обработка интернет заказов	−	−	−	+
Отчеты по доставке	−	−	+	+

Рисунок 18 – Таблица тарифов r_keeper

Еще один пример подобного приложения – является «Мобильный официант» от «1С-Рарус». На рисунке 19 изображен главный экран данного приложения. Интерфейс и функционал приложения выглядит отлично. Однако весь бэкенд приложения - зависим от системы, «РестАрт» которая разработана «1С-Рарус» [11].

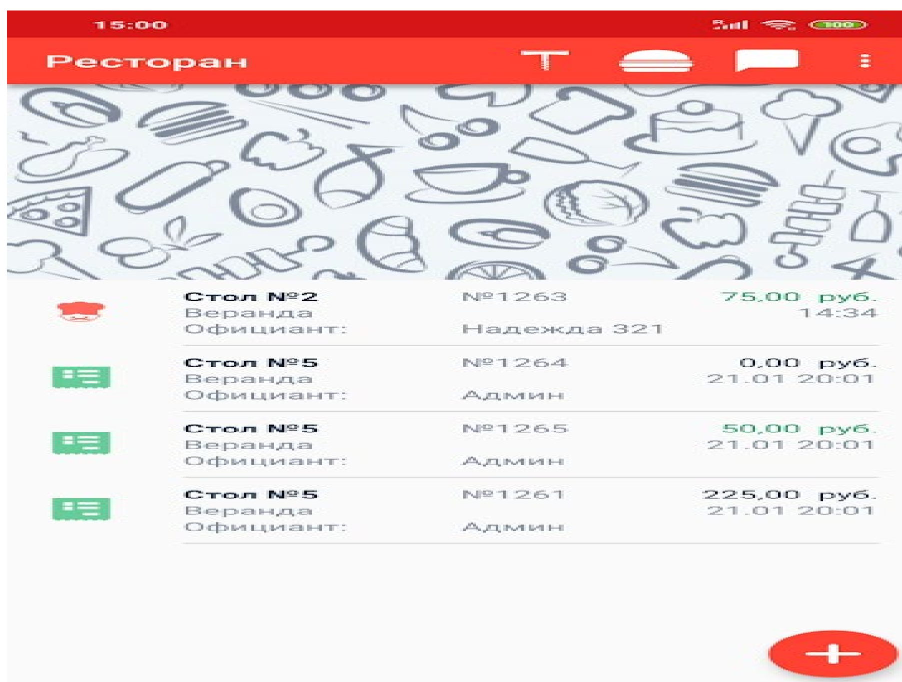

Рисунок 19 – Мобильный официант «РестАрт»

Интегрировать данную систему в предприятие достаточно сложно, поэтому часто необходимо сопровождение специалиста. Такая услуга предоставляется и плата за нее рассчитывается отдельно исходя из выбранного тарифа.

Сравнив данные продукты со своей разработкой, могу отметить следующие особенности:

– слишком много действий со стороны пользователя для достижения интеграции;

– оба приложения используют принцип «конструктора» для составления заказа, что менее оптимально чем голосовое управление;

– разработчики данных продуктов недостаточно оперативно отвечают на запросы пользователей.

Исходя из анализа приложений-аналогов в данной сфере считаю, что разработанное приложение исправляет недостатки и итерирует над функционалом предшественников.

ЗАКЛЮЧЕНИЕ

При выполнении выпускной квалификационной работы были поставлены и решены следующие задачи:

– выбран облачный API для перевода речи в текст;

– настроены службs для обеспечения инфраструктуры приложения;

– интеграция служб в приложение.

Для решения поставленных задач и достижения цели выпускной квалификационной работы были проанализированы специализированные источники литературы. Исходя из экспериментальных данных в качестве облачного API, отвечающего за обработку речи был выбран Google Cloud Speech-to-Text API. Данный сервис, показал превосходные результаты: гибкость в настройке, высокую скорость потокового распознавания, и невероятную точность в распознавании. API является частью сервиса Google Cloud предоставляющего множество облачных решений. Мощная инфраструктура созданная Google позволяет расширять возможности современных мобильных приложений. Такие задачи как сервер базы данных, защищенное файловое хранилище и аутентификация пользователей были возложены на платформу Firebase. Firebase Realtime Database выступила в роли NoSQL базы данных. Данный компонент хорошо обрабатывает большие объёмы данных и обладает повышенной безопасностью. Firebase Cloud Storage отвечала за хранение ключей сторонних API, ключевой особенностью являет поддержка большого количества форматов файлов, а также скачивание файла в виде потока без необходимости прописывать URL. Firebase authentication берет на себя логику форм входа и регистрации, учет пользователей и обеспечивает множество провайдеров (вариантов для предоставления данных). Платформа Firebase является идеальным решением для мобильных приложений, так как она находится в тесном сотрудничестве с Google. Данное сотрудничество позволяет Firebase предлагать все новые сервисы для решения задач, связанных с серверной логикой, тестированием и аналитикой. В ходе проектирования было создано

приложение «Мобильный официант», автоматизирующие процесс создание заказа и развернуто на устройстве под управлением Android. В процессе разработки были изучены и применены такие основные компоненты Android приложений как фрагменты, активности, стек навигации. Также были освоены следующие компетенции:

ОК-4 – понимание социальной значимости своей будущей профессии, обладание высокой мотивацией к выполнению профессиональной деятельности;

ОПК-1 – владение широкой общей подготовкой (базовыми знаниями) для решения практических задач в области информационных систем и технологий;

ОПК-5 – способность использовать современные компьютерные технологии поиска информации для решения поставленной задачи, критического анализа этой информации и обоснования принятых идей и подходов к решению;

ПК-12 – способность разрабатывать средства реализации информационных технологий (методические, информационные, математические, алгоритмические, технические и программные);

ПК-22 – способность проводить сбор, анализ научно-технической информации, отечественного и зарубежного опыта по тематике исследования;

ПК-26 – способность оформлять полученные рабочие результаты в виде презентаций, научно-технических отчетов, статей и докладов на научно-технических конференциях.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Колойтанов, В. И. Обзор речевых технологий / В. И. Колойтанов, А. Ю. Ткаченко // Современное состояние и приоритеты развития фундаментальных наук в регионах: материалы XVIII Всерос. науч. конф. молодых ученых и студентов: сборник научных статей / редакционная коллегия: В. А. Исаев, Н. Н. Куликова; Кубанский Государственный университет. – Краснодар Кубанский Государственный университет, 2021. – С. 26–31. – Библиогр.: с. 31.

2. Фролов, А.В. Синтез и распознавание речи. Современные решения/ А. В. Фролов, Г. В. Фролов – 2018. – URL: http://www.frolovlib.ru: (17.01.2022).

3. Воробьева, С. А. Методы распознавания речи / С. А. Воробьева. – Текст: непосредственный // Молодой ученый. – 2016. – № 26 (130). – С. 136 - 141. – URL: https://moluch.ru/archive/130/36213: (25.01.2022).

4. Andrey L. Herchonvicz A comparison of cloud-based speech recognition engines / Andrey L. Herchonvicz, Cristiano R. Franco, Marcio G. Jasinski // X Computer on the Beach. – 2019. – № 9. – P. 366–375.

5. Спинеллис, Д. Идеальная архитектура. Ведущие специалисты о красоте программных архитектур / Д. Спинеллис, Г. Гусиос, [перевод с английского Е. Матвеева]. – Санкт-Петербург: СимволПлюс, 2018. – 528 с. – ISBN 9785932861752.

6. Ashok Kumar S. Mastering Firebase for Android Development / S. Ashok Kumar. – Birmingham: Packt Publishing Ltd, 2018. – 372 p. – ISBN 978-1-78862-471-8.

7. Жемеров, Дмитрий. Kotlin в действии / Дмитрий Жемеров, Светлана Исакова, [перевод с английского А. Н. Киселев]. – Москва: ДМК Пресс, 2018. – 401 с. – ISBN 978-5-97060-497-7.

8. Клифтон Ян. Проектирование пользовательского интерфейса в Android / Ян Клифтон. – Москва: ДМК Пресс, 2017. – 451 с. – ISBN 978-5-97060-449-6.

9. Риз, Дж. Облачные вычисления / Дж. Риз. – Санкт-Петербург: БХВ-Петербург, 2019. – 288 с. – ISBN 978-5-9775-0630-4.

10. Официальный сайт системы автоматизации r_keeper – 2021. – URL: https://rkeeper.ru/tariffs/: (13.01.2022).

11. Официальный сайт программного продукта РестАрт – 2021. – URL: https://rarus.ru/1c-restoran/as-restart/: (01.02.2021).