Добрый день уважаемая комиссия!

Наша команда состоит из Горчаковой Софьи, Матвеевой Ульяны, Семеновой Вероники и Суриновой Ангелины, юридический факультет, группа ЮИ-21, наш проект мобильное приложение TastyFood.

Это результат нескольких месяцев изучения android разработки, обсуждений, проб и ошибок.

По реализации могут быть вопросы в некоторых местах мы до конца не разобрались, но старались делать максимально хорошо исходя из знаний, что получили на курсе.

### **0. Как мы работали в команде**

У нас команда из четырех человек, и мы сразу договорились о разделении ролей.

Первый и второй участники в основном отвечали за техническую часть: писал код, разбирался с архитектурой, интегрировал библиотеки, настраивал навигацию, работал с базой данных и backend-частью.

Третий участник больше занимался сбором информации, собирал требования, а также выступал как project manager — планировал задачи, следил за сроками.

Четвертый участник оформлял документацию, делал презентацию и отчет по проекту.

Когда мы застревали на сложных моментах (например, с DI или Room), другие участники помогали искать решения, вместе читали документацию, смотрели видеоуроки. А когда нужно было оформить презентацию или отчет, мы помогали с техническими скриншотами и объяснениями, чтобы всё выглядело понятно для комиссии.

Такое разделение помогло нам не только быстрее двигаться, но и не терять фокус: мы могли глубже погрузиться в технические детали и в организацию и коммуникацию.

Благодаря этому мы смогли учесть реальные потребности пользователей и сделать проект более живым и приближенным к реальной работе.

В итоге мы научились не только программировать и быстро изучать новые и сложные для нас темы, напомню мы по специальности не являемся программистами, но и работать в команде, слушать друг друга и договариваться.

### **1. Введение и мотивация**

Выбор тем был свободный, можно было выбрать из представленных тем преподавателем.

Идея нашего приложения возникла не случайно — современные люди больше вовлечены в использование гаджетов, но потребность в еде от этого не становится меньше.

Наше приложение - отличный помощник для людей, которые не хотят сильно заморачиваться над поиском и выбором рецепта или всегда забывают, куда сохранили понравившийся рецепт.

Мы решили, что можем сделать актуальное приложение, которое станет персональной кулинарной книгой для любого человека.

### **2. Основной функционал и пользовательский сценарий**

Мы старались, чтобы каждый экран приложения решал конкретную задачу пользователя. Реализованы экраны:

* **SplashScreen** — не просто заставка, а возможность задать настроение и бренд приложения.
* **Главый экран** — “лицо” приложения, на нем располагаются все основные функции.
* **Поиск рецепта по названию** — пользователь вводит первые буквы названия рецепта и быстро находит его в ленте.
* **Лента рецептов** — пользователь может выбрать уже готовый рецепт из ленты или поискать рецепты определенной категории.
* **Добавление своего рецепта** — кнопка, при нажатии на которую пользователь оказывается на экране, где пошагово может добавить свой собственный рецепт, прикрепить фото готового блюда или даже ссылку на видео.

Каждый экран мы продумывали и сравнивали с аналогичными существующими приложениями.

### **3. Демонстрация экранов**

(Показываем презентацию или приложение в эмуляторе)

* На SplashScreen — фирменная иллюстрация, слоган, плавная анимация.

Тут мы видим экран входа, с логотипом и названием салона.

Так же рассказываем и показываем про каждый экран, даже если пока не реализована функциональность или используются заглушки в эмуляторе, а тем более в презентации рабочую функциональность сложно понять.

* **Главый экран** — название приложения, меню, кнопки и картинки с рецептами.
* **Поиск рецепта по названию** — поиск, вывод картинок по названию.
* **Лента рецептов** — картинки, названия, краткое описание.
* **Добавление своего рецепта** — название, описание, сложность, время приготовления, продукты, фото.

### **4. Архитектура и структура проекта**

Мы выбрали архитектуру **MVVM + Clean Architecture** — честно говоря изначально мы думали, что можно реализовать через MVC/MVI подходы используя одну или несколько MainActivity, но по мере полученных знаний и реализации стало понятно, что приложение такого уровня куда сложнее поэтому пришлось разделять логику на компоненты.

В проекте:

* **Presentation Layer** — все экраны на Jetpack Compose, каждый экран — отдельный Composable, логика работы экранов вынесена во ViewModel.
* **Domain Layer** — use cases, которые инкапсулируют бизнес-логику.
* **Data Layer** — репозитории, которые общаются с Room и Supabase.
* Например, **TheMealDbRepository** реализует интерфейс, а внутри сам решает, брать данные из локальной БД или из облака.

**Пример кода:**

suspend fun getSomeMealsFromCategory(category: String, count: Int = 2): List<MealDto> {

val shortList = getMealsByCategory(category)

println("getSomeMealsFromCategory: shortList.size = ${shortList.size}")

Log.d("TheMealDbRepository", "Коротких рецептов по категории '$category': ${shortList.size}")

val selected = if (shortList.size > count) shortList.shuffled().take(count) else shortList

val result = mutableListOf<MealDto>()

for (shortMeal in selected) {

try {

println("lookupMealById: idMeal = ${shortMeal.idMeal}")

val detail = api.lookupMealById(shortMeal.idMeal ?: "").meals?.firstOrNull()

if (detail != null) result.add(detail)

} catch (e: Exception) {

Log.e("TheMealDbRepository", "Ошибка при получении подробного рецепта: ${shortMeal.idMeal}", e)

println("Ошибка при получении подробного рецепта: ${shortMeal.idMeal}, ${e.message}")

}

}

println("getSomeMealsFromCategory: result.size = ${result.size}")

Log.d("TheMealDbRepository", "Подробных рецептов получено: ${result.size}")

return result

}

### **5. Реализация: детали и этапы**

* Весь UI построен на Jetpack Compose c Material Design 3. Например, экран сетевых рецептов — это LazyColumn с карточками, каждая карточка — отдельный Composable элемент.
* Навигация реализована через Navigation Compose, все маршруты вынесены в отдельный файл NavGraph.kt.

Для хранения данных о пользователях, услугах, мастерах и бронированиях — отдельные сущности и DAO в Room. Пример:  
@Entity

data class RecipeEntity(

@PrimaryKey (autoGenerate = True),

val id: Int = 0,

val title: String,

val imageURL: String?,

val cookingTime: String,

val difficulty: Int,

val description: String,

val rating: Float,

val isFavorite: Boollean = false,

val ingredients: List<String> = emptyList(),

val descriptoinMedia: List<String> = emptyList(),

val category: String = “Все рецепты”,

val steps: List<com.students.tastyfood.model/RecipeStep> = emptyList()

)

* Для загрузки изображений — Coil, особенно важно для ленты с рецептами.

### **6. Почему именно такой стек**

Все выбранные технологии и архитектурные решения мы брали из уроков курса и примеров, которые нам показывали преподаватели. Мы считаем этот стек наиболее верным, потому что он современный, поддерживается Google, и реально работает в боевых проектах.

К тому же, все модули курса — архитектура, базы данных, асинхронность, DI, UI — мы применили на практике. Например, DI через Hilt мы впервые попробовали именно на этом проекте, и теперь понимаю, насколько это облегчает тестирование и поддержку кода. Но если честно, до конца не разобрались, как работает DI внутри — просто следовал примеру из урока: добавил аннотации @Inject, @HiltViewModel, прописал модули, и оно заработало.

Понимаем, что это про то, чтобы зависимости создавались автоматически, но если что-то пойдет не так — пока не уверены, что сможем быстро починить. Хотелось бы в дальнейшем глубже разобраться в этой теме.

### **7. Где и как использовали технологии**

* **Room** — для локального хранения всех сущностей . Например, при отсутствии интернета пользователь все равно видит свои рецепты. Но если нет данных с сервера, мы подгружаем мок-объекты, чтобы приложение не выглядело пустым.
* **DI (Hilt)** — для внедрения зависимостей, чтобы не было жестких связей между слоями. Пример: **RecipeRepository** внедряется во ViewModel через Hilt. Но, как уже сказали, делали по шаблону из курса, и если бы пришлось настраивать что-то сложнее, скорее всего, возникли бы вопросы и понадобилось много времени.
* **API ( используем Supabase)** — для синхронизации данных, регистрации и аутентификации пользователей. Но часть функций пока работают на мок-данных, потому что не успели реализовать полноценный backend и не разобрались с подключением Api.
* **Потоки и async (Coroutines, Flow)** — для загрузки данных, обновления UI в реальном времени.
* **Navigation Compose** — для переходов между экранами, передачи параметров. Все маршруты вынесены в отдельный файл.
* **Coil** — для загрузки фото рецептов. Если фото не загружается — показываем заглушку.
* **Material Design 3** — для современного UI и поддержки темной темы. Темная тема переключается автоматически или вручную.

### **8. Трудности и решения**

Честно скажу, не все темы дались легко. Особенно сложно было с:

* **Room** — сначала путались с миграциями и связями между сущностями. Для тестирования использовали мок-данные, чтобы не потерять данные при ошибках миграции.
* **DI (Hilt)** — не сразу поняли, как правильно настраивать модули и компоненты. В итоге часть зависимостей пока внедряется вручную, но хотим доработать. Пока больше следованием примерам из курса.
* **API и асинхронность** — пришлось разбираться с Coroutines, чтобы не нагружать основной поток для всего.
* **Кеширование и офлайн-режим** — было сложно сделать так, чтобы данные не терялись при переключении между онлайн и офлайн. Сейчас часть данных всегда берутся из мок-списков, чтобы приложение работало даже без интернета.

### **9. Перспективы развития**

В будущем хотим добавить:

* Список покупок — пользователь сможет записывать нужные продукты прямо в приложении или выбирать их из готового списка.
* Корзина и доставка - хотим добавить возможность заказа продуктов прямо в приложении из самых популярных магазинов (Перекресток, Пятерочка, Лента).
* Лента обсуждений - возможность опубликовать свой рецепт, писать комментарии, подписываться на пользователей с интересными рецептами.

### **10. Личный опыт и выводы**

В команде работало 4 человека: двое отвечал за анализ и UX, двое — за разработку. Иногда подключался друг к другу для тестирования и сбора обратной связи. За время работы мы научились не только программировать, но и работать в команде, планировать задачи, искать компромиссы.

Я лично понял, как важно продумывать архитектуру заранее, как сложно, но интересно реализовывать офлайн-режим и синхронизацию, и как приятно видеть, что твой продукт реально может быть настоящим приложением полезным людям.

Если бы было больше времени, мы бы обязательно доработали некоторые функции. Очень хочется продолжить изучать мобильную разработку.

**Спасибо за внимание! Готов ответить на ваши вопросы — по архитектуре, технологиям, реализации, тестированию или любым другим аспектам проекта.**

### **Возможные вопросы на защите и как на них ответить**

### 

### **Вопросы по проекту:**

* Как реализовано получение данных о клиентах, услугах, мастерах?
  + Ответ: Основные данные (услуги, мастера, акции) сейчас берутся из Supabase удаленной базы данных, на случай недостающих данных подготовлены массивы из мок-объектов — это заранее подготовленные списки в коде.
  + Мы сделали так, потому что не успели реализовать полноценный backend. Для профиля пользователя планировали использовать авторизацию supabase authentication, но пока работает только локально через Room.
* Как осуществляется кэширование?
  + Ответ: Для кэширования и офлайн-режима используем Room. Все основные сущности (услуги, мастера, бронирования) сохраняются в локальной базе. Если нет интернета — показываем данные из Room или из мок-объектов.
* Как пользователь может добавить запись или изменить данные?
  + Ответ: Добавление и изменение реализовано в ручную через редактирования таблицы supabase, авторизацию планировали supabase authentication но пока не успели подключить — данные сохраняются в Room, а при появлении интернета (в будущем) будет синхронизация с Supabase.
* Как реализована работа с тёмной темой?
  + Ответ: Используем Material 3 и Jetpack Compose, переключатель темы есть в настройках, состояние темы сохраняется в SharedPreferences.
* Как происходит взаимодействие с внешним API?
  + Ответ: В нашем проекте внешний API — только Supabase, и то только для синхронизации данных пользователя мастера, услуги и бронирований. Все остальные данные (отзывы, лента новостей, профиль пользователя) — это мок-данные.
* Почему не реализованы push-уведомления?
  + Ответ: Не хватило времени и опыта работы с Cloud Messaging. Сейчас уведомления реализованы как анимация, но планируем доработать.
* Как устроены мок-данные и почему вы их используете?
  + Ответ: Мок-данные — это обычные списки объектов, которые мы храним прямо в коде (например, список мастеров, услуг, акций).
  + Мы используем их, чтобы приложение работало и выглядело полноценно даже без полноценного backendа и внешнего API. Мы думали использовать внешние данные через API но не нашли подходящие данные с открытым доступом для салона красоты.
* Как тестировали офлайн-режим?
  + Ответ: Отключали интернет на устройстве/эмуляторе и проверяли, что данные из Room и мок-объектов отображаются корректно, а приложение не падает.
* Как бы реализовали push-уведомления?
  + Ответ: В интернете рекомендуют Cloud Messaging, думаю полученных знаний хватает разобраться в настройке.
* Как бы реализовали интеграцию с платежами?
  + Ответ: Рассматривали бы Яндекс.Кассу (через SDK) или Telegram Pay (через Bot API), реализовали бы отдельный экран оплаты и обработку статуса платежа.
* Как бы добавили аналитику?
  + Ответ: Интегрировали бы Amplitude или Яндекс.Метрику, отправляли бы события при бронировании, оплате, просмотре экрана.
* Как бы реализовали поддержку нескольких салонов?
  + Ответ: Самое простое решение в упор создать отдельный клон приложения с отдельной бд.
  + Если у нас салон относится к одной франшизе, тогда добавили бы сущность "Салон" в базу, реализовали бы выбор салона на первом экране или на новом экране, фильтрацию услуг и мастеров по выбранному салону.

**Вопросы по темам курса:**

* Что такое activity
  + Activity в Android — компонент приложения, который представляет собой отдельный экран или окно. Каждая Activity обычно соответствует одному экрану пользовательского интерфейса или фрагменту функциональности.
* Как устроен жизненный цикл Activity?
  + Ответ: Activity проходит стадии onCreate, onStart, onResume, onPause, onStop, onDestroy. Важно правильно управлять ресурсами и состоянием.
* Как устроен жизненный цикл Composable?
  + Ответ: Composable-функции пересоздаются при изменении состояния(рекомпозиции), важно использовать remember и rememberSaveable для сохранения данных.
* Как реализована навигация между экранами?
  + Ответ: Navigation Compose, NavController, маршруты вынесены в отдельный файл. Параметры передаются через аргументы.
* Как реализовано хранение данных?
  + Ответ: Supabase для удаленных данных,Room для сложных структур, SharedPreferences для настроек, мок-данные для демонстрации.
* Как работает Room и для чего он используется?
  + Ответ: Room — это библиотека для работы с SQLite, позволяет хранить объекты и получать их через DAO. В нашем проекте — для бронирований, профиля, истории.
* Как внедряются зависимости через Hilt?
  + Ответ: Используем аннотации @Inject, @HiltViewModel, модули для предоставления зависимостей. Честно: делал по примеру из курса, до конца не разобрался, насколько я понял это облегчает тестирование и поддержку кода.
* Как тестировали приложение?
  + Ответ: Вручную, на эмуляторе и устройстве, тесты есть но простые не сильно вникали в данную тему в первую очередь старались максимально реализовать функциональную часть приложения, но структура позволяет их добавить.
* Как бы реализовали поддержку нескольких языков?
  + Ответ: Через ресурсы strings.xml, локализацию, переключение языка в настройках.
* Какие трудности были с архитектурой?
  + Ответ: Сложно было понять, как правильно разделять слои и для чего это используется,

Почему не реализованы некоторые функции?

* + Ответ: Не хватило времени и опыта, часть функций (push, оплата, полноценный backend) оставили на будущее, чтобы не усложнять проект, попытаться реализовать большую часть и не делать неработающие заглушки.

**Вопросы по Android-разработке:**

* Чем отличается ViewModel от Activity?
  + Ответ: ViewModel хранит состояние и логику, переживает повороты экрана, Activity — только UI.
* Для чего нужен Hilt и как он работает?
  + Ответ: Для автоматического внедрения зависимостей, чтобы не создавать их вручную. Работает через аннотации и генерирует код для DI.
* Как реализовать асинхронную загрузку данных?
  + Ответ: Coroutines, suspend-функции, Flow.
* Как реализовать сохранение состояния при повороте экрана?
  + Ответ: ViewModel, rememberSaveable, SavedStateHandle.
* Какие паттерны проектирования использовали?
  + Ответ: MVVM, Repository, Singleton

**Совет:**Если не знаете точный ответ — честно скажите, что делали по примеру из курса, не до конца разобрались, но готовы изучить глубже. Попробуйте рассказать общую идею и ваше понимание.