Министерство образования и науки Нижегородской области

Государственное бюджетное профессиональное образовательное учреждение

**«Нижегородский Губернский колледж»**

Методическая комиссия Информатика и вычислительная техника

Допущен к защите:

преподаватель

\_\_\_\_\_\_\_\_\_\_Ю. С. Мамшева

« » 2023 г.

РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ

**ПРОЕКТНАЯ РАБОТА**

по МДК 01.03 «Разработка мобильных приложений»

Студент: \_\_\_\_\_\_\_\_\_\_\_\_ А. А. Климычева 27.11.2024

(подпись)

Специальность, группа: 09.02.07, 31П

Нижний Новгород

2024

СОДЕРЖАНИЕ

# ГЛАВА 1. ПОСТАНОВКА ЦЕЛЕЙ И ЗАДАЧ

## Разработка технического задания

Техническое задание представлено в [Приложении 1](#_Приложение_1).

## Выбор ПО и паттерна проектирования

Выбор ПО и паттерна представлен в [Приложении 2.](#_Приложение_2)

# ГЛАВА 2. РАЗРАБОТКА МАКЕТА

## Макет каждого экрана разрабатываемого приложения.

Макет был выполнен в Figma и представлен по ссылке: <https://www.figma.com/design/v5KJd1cucl5fac1s3jC2wG/Study-Buddy?node-id=0-1&t=B7dIjNDIy9C9wzZF-1>.

## Выбор цветовой палитры

Разработаны две цветовые темы: тёмная (Рис. 1 ниже) и светлая (Рис. 2 ниже).

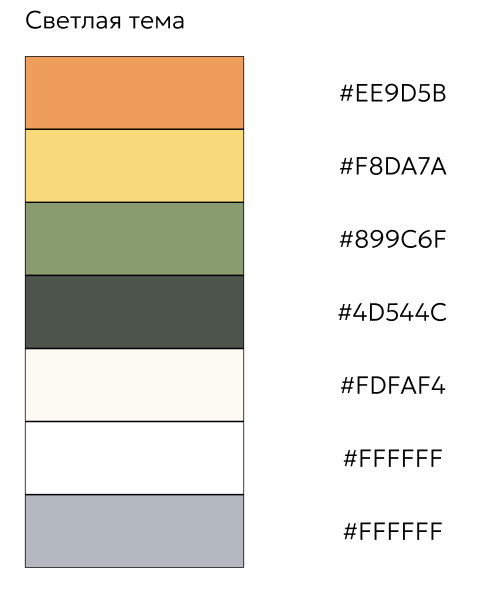


Рисунок 1. Цветовая схема светлой темы приложения

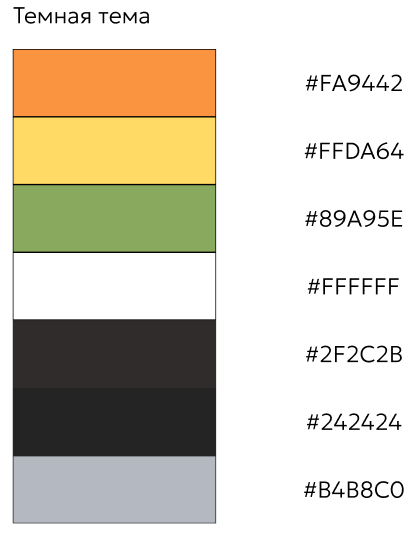


Рисунок 2. Цветовая схема темной темы приложения

## Выбор шрифта для заголовков и основного текста

Шрифт, используемый в макете приложения указан на Рисунке 3 ниже.



Рисунок 3. Шрифты приложения.

## Элементы формы

В Style Guide были вынесены все элементы формы. Их можно увидеть на Рис. 5 ниже

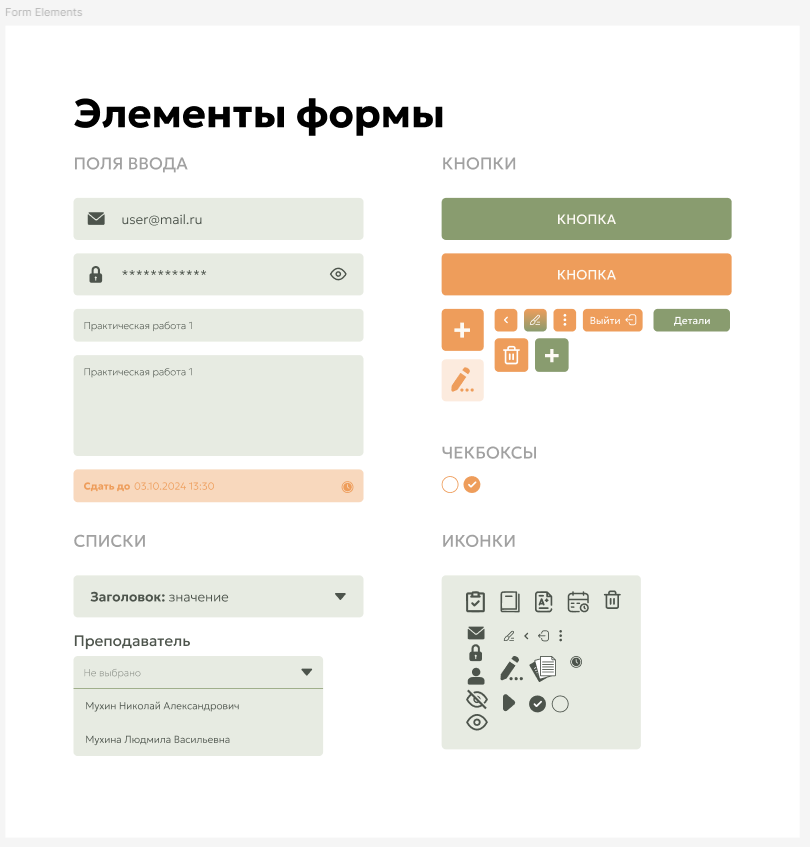


Рисунок 5. Элементы формы.

## Разработка логотипа

Разработан логотип приложения. Его можно увидеть на Рис. 6 ниже

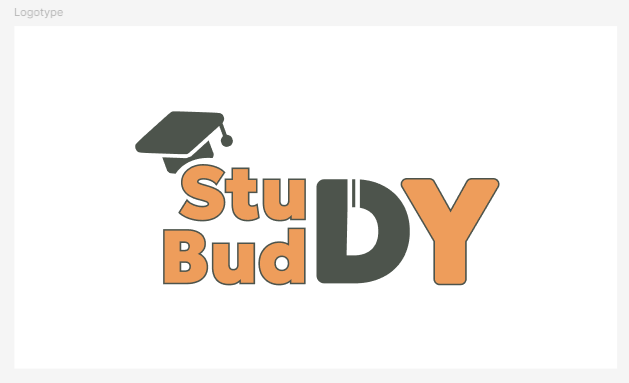


Рисунок 6. Логотип.

## Инструменты Figma.

Если перейти по [ссылке макета](https://www.figma.com/design/v5KJd1cucl5fac1s3jC2wG/Study-Buddy?node-id=0-1&t=B7dIjNDIy9C9wzZF-1), откроется главная страница макета с описанием стилей. Всего таких страниц 3.

1 страница: Style Guide – описание стилей, цвета, шрифты, элементы формы.

2 страница: Приложение – макет приложения, как будут выглядеть страницы в темной и светлой теме.

3 страница: Черновик стилей – страница для экспериментов в пользовательском интерфейсе (Подбор оптимальных шрифтов, цвета, все варианты иконок приложения и т.д.). Имеет вес только для разработчика.

Страницы выглядят следующим образом (см. Рис. 4 ниже)

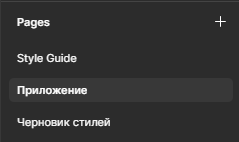


Рисунок 4. Страницы приложения.

# ГЛАВА 3. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

Серверная часть приложения StudyBuddy написана на платформе APS.NET с использованием Identity для авторизации пользователей через JWT token. После успешного тестирования, API была опубликована на сервере ГБПОУ НГК по адресу: <https://iis.ngknn.ru/ngknn/МамшеваЮС/10/swagger/index.html>

# ГЛАВА 4. ТЕСТИРОВАНИЕ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

Документация тестирования серверной части представлена в [Приложении 3](#_Приложение_3).

# ГЛАВА 5. РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ

## Используемые библиотеки

1. Библиотеки для работы с JWT (JSON Web Tokens) — для декодирования и верификации токенов.

|  |
| --- |
| *implementation* ("io.jsonwebtoken:jjwt-api:0.12.6") *runtimeOnly* ("io.jsonwebtoken:jjwt-impl:0.12.6") *implementation* ("com.auth0.android:jwtdecode:2.0.2") *runtimeOnly* ("io.jsonwebtoken:jjwt-gson:0.12.6") |

1. Библиотека Jetpack Compose для реализации навигации между экранами в вашем приложении.

|  |
| --- |
| val nav\_version = "2.7.7" *implementation*("androidx.navigation:navigation-compose:$nav\_version") |

1. Набор библиотек Ktor для создания HTTP-клиента. Используются различные модули для базовой функциональности, работы с Android, логирования, обработки контента и сериализации/десериализации JSON с помощью kotlinx.serialization.

|  |
| --- |
| val ktor\_version = "2.3.10" *implementation*("io.ktor:ktor-client-core:$ktor\_version") *implementation*("io.ktor:ktor-client-android:$ktor\_version") *implementation*("io.ktor:ktor-client-logging:$ktor\_version") *implementation*("io.ktor:ktor-client-content-negotiation:$ktor\_version") *implementation*("io.ktor:ktor-serialization-kotlinx-json:$ktor\_version") |

1. Библиотека для логирования, используемая в сочетании с Ktor для вывода логов.

|  |
| --- |
| *implementation*("ch.qos.logback:logback-classic:1.2.3") |

1. Библиотека Compose для реализации компонента “Pull to refresh” (обновление при прокрутке вниз).

|  |
| --- |
| *implementation*("eu.bambooapps:compose-material3-pullrefresh:1.0.0") |

1. Библиотеки Hilt (DI фреймворк) для внедрения зависимостей.

|  |
| --- |
| val hilt\_android\_version = "2.48" *implementation*("com.google.dagger:hilt-android:$hilt\_android\_version") *kapt*("com.google.dagger:hilt-compiler:$hilt\_android\_version") *implementation*("androidx.hilt:hilt-navigation-compose:1.0.0") |

1. Библиотека для написания UI-тестов для Jetpack Compose с использованием JUnit 4.

|  |
| --- |
| val compose\_version = "1.7.5" *androidTestImplementation*("androidx.compose.ui:ui-test-junit4:$compose\_version") |

1. Библиотеки Room для работы с локальной базой данных.

|  |
| --- |
| val room\_version = "2.4.0" *implementation*("androidx.room:room-runtime:$room\_version") *kapt*("androidx.room:room-compiler:$room\_version") *implementation* ("androidx.room:room-ktx:$room\_version") |

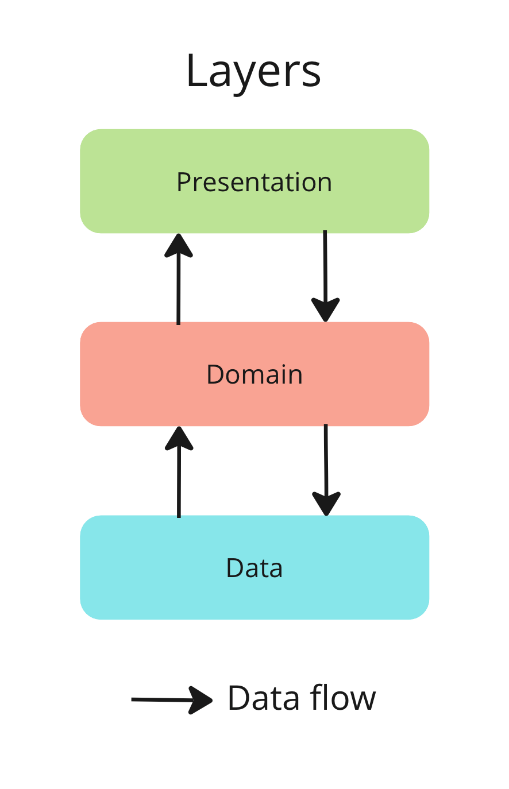
## Архитектура

Выбранная архитектура - Clean Architecture. Такой выбор был сделан исходя из следующих плюсов:

1. Разделение ответственности (масштабируемость, минимизация проблем с зависимостями)
2. Тестируемость
3. Устойчивость к изменениям
4. Популярность (большое кол-во гайдлайнов, шаблонов)

Принципы:

1. Единственная ответственность. Каждый слой (модуль), класс или функция выполняет только одну задачу. Данный принцип призван конкретизировать ответственность отдельных по смыслу классов или слоев, что в свою очередь придает ясность в код и снижает его связанность.
2. Разделение на слои. Приложение должно быть разбито на слои, у каждого слоя своя зона ответственности. Обычно выделяют следующие уровни: presentation, domain, data.
   1. Presentation: отвечает за отображение пользовательского интерфейса и реагирование на его события
   2. Domain: бизнес-логика, изолированная от деталей реализации, определяет правила и операции, как приложение должно взаимодействовать с данными
   3. Data: хранилище данных



1. Инверсия зависимостей. Один из важнейших принципов который гласит о том, что стоит использовать общий контракт, такой как интерфейс или абстрактный класс, вместо прямой зависимости слоя верхнего уровня от компонентов слоя нижнего. Таким образом, каждый слой использует этот контракт, что обеспечивает изоляцию изменений в верхнем слое.

Архитектура проекта представлена на Рисунке 7 ниже.

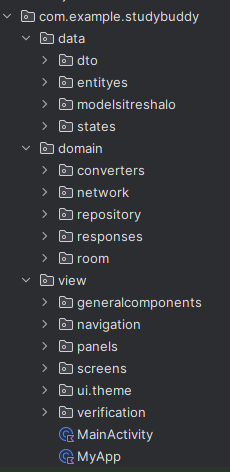


Рисунок 7. Архитектура проекта.

## Тема

После создания приложения на Jetpack Compose создаётся базовая тема Material Theme. Но данную тему не очень удобно использовать, поэтому была создана новая, custom тема StudyBuddyTheme.

|  |
| --- |
| @Composable fun StudyBuddyTheme(  themeMode: ThemeMode = ThemeMode.Dark,  typography: Typography = StudyBuddyTheme.typography,  content: @Composable () -> Unit ) {  val colors = when (themeMode) {  ThemeMode.Dark -> *darkPalette* ThemeMode.Light -> *baseLightPalette* }  CompositionLocalProvider(  *LocalTypography* provides typography,  *LocalColors* provides colors  )**{** content()  **}** } |

В эту тему можно передать аргументом цветовую режим темы (светлая, тёмная).

Также для удобства был создан класс темы, чтобы задавать свои названия переменным с цветом.

|  |
| --- |
| data class ColorPalette(  val background: Color,  val primary: Color,  val secondary: Color,  val third: Color,  val textTitle: Color,  val textDesc: Color,  val textButton: Color,  val containerPrimary: Color,  val containerSecondary: Color,  val containerThird: Color,  val containerDefault: Color,  val unselectItem: Color ) |

На основе этого класса были добавлены две темы: темная и светлая.

|  |
| --- |
| val *Cream* = *Color*(0xFFFDFAF4) val *DarkBack* = *Color*(0xFF2F2C2B) val *DarkCont* = *Color*(0xFF242424) val *White* = *Color*(0xFFFFFFFF) val *White40* = *Color*(0x66FFFFFF) val *Orange* = *Color*(0xFFEE9D5B) val *Orange2* = *Color*(0xFFFA9442) val *Gray* = *Color*(0xFFB4B8C0) val *Dark\_Green* = *Color*(0xFF4D544C) val *Green* = *Color*(0xFF899C6F) val *Green2* = *Color*(0xFF89A95E) val *Yellow* = *Color*(0xFFF8DA7A)  val *baseLightPalette* = ColorPalette(  background = *Cream*,  primary = *Green*,  secondary = *Orange*,  third = *Yellow*,  textTitle = *Dark\_Green*,  textDesc = *Green*,  textButton = *White*,  containerPrimary = *Green*.copy(alpha = 0.2f),  containerSecondary = *Orange*.copy(alpha = 0.2f),  containerThird = *Yellow*.copy(alpha = 0.8f),  containerDefault = *White*,  unselectItem = *Gray* )  val *darkPalette* = ColorPalette(  background = *DarkBack*,  primary = *Green2*,  secondary = *Orange2*,  third = *Yellow*,  textTitle = *White*,  textDesc = *Green2*,  textButton = *White*,  containerPrimary = *Green2*.copy(alpha = 0.2f),  containerSecondary = *Orange2*.copy(alpha = 0.2f),  containerDefault = *DarkCont*,  containerThird = *Yellow*.copy(alpha = 0.8f),  unselectItem = *White40* )  val *LocalColors* = *staticCompositionLocalOf* **{** *darkPalette* **}** |

Точно таким же способом создан класс для шрифтов, в котором добавлены шрифты и стили для текста

|  |
| --- |
| data class Typography(  val bold: TextStyle,  val regular: TextStyle,  val extralight: TextStyle,  val light: TextStyle )  val *Geologica* = *FontFamily*(  *Font*(R.font.*geologica\_black*, FontWeight.Black),  *Font*(R.font.*geologica\_bold*, FontWeight.Bold),  *Font*(R.font.*geologica\_extra\_bold*, FontWeight.ExtraBold),  *Font*(R.font.*geologica\_extra\_light*, FontWeight.ExtraLight),  *Font*(R.font.*geologica\_light*, FontWeight.Light),  *Font*(R.font.*geologica\_medium*, FontWeight.Medium),  *Font*(R.font.*geologica\_regular*, FontWeight.Normal),  *Font*(R.font.*geologica\_semi\_bold*, FontWeight.SemiBold),  *Font*(R.font.*geologica\_thin*, FontWeight.Thin) )  val *typography* = Typography(  bold = TextStyle(  fontWeight = FontWeight.Bold,  fontFamily = *Geologica*,  fontSize = 16.*sp* ),  regular = TextStyle(  fontWeight = FontWeight.Normal,  fontFamily = *Geologica* ),  extralight = TextStyle(  fontWeight = FontWeight.ExtraLight,  fontFamily = *Geologica* ),  light = TextStyle(  fontWeight = FontWeight.Light,  fontFamily = *Geologica* ) )  val *LocalTypography* = *staticCompositionLocalOf* **{** *typography* **}** |

В MainActicity установлена тема

|  |
| --- |
| val currentThemeMode = remember **{** *mutableStateOf*(UserRepository.themes.*first* **{ it**.title == UserRepository.theme **}**) **}** StudyBuddyTheme(themeMode = currentThemeMode.value) **{** Scaffold() **{** Navigation()  **} }** |

## Настройка DI (Hilt).

В процессе разработки мобильного приложения, многие классы (например, viewModel) имеют зависимости, которые необходимо передавать при создании объекта. Зависимостей может быть достаточно много и для того, чтобы каждый раз для создания объекта с зависимостями не нужно было прописывать и сами зависимости будет использована библиотека Hilt, которая будет автоматически генерировать нужные зависимости (для этого нужно только прописать модуль для генерации)

Все приложения, использующие Hilt, должны содержать класс Application, помеченный @HiltAndroidApp.

|  |
| --- |
| @HiltAndroidApp class MyApp : Application() {  } |

@HiltAndroidApp запускает генерацию кода Hilt, включая базовый класс для приложения, который служит контейнером зависимостей на уровне приложения.

Этот сгенерированный компонент Hilt прикрепляется к жизненному циклу объекта Application и предоставляет ему зависимости. Кроме того, это родительский компонент приложения, а это означает, что другие компоненты могут получить доступ к предоставляемым им зависимостям.

Также необходимо указать его в Manifest в атрибуте name

|  |
| --- |
| <application  android:name=".view.MyApp"  /// </application> |

Также, чтобы Hilt предоставлял зависимости другим классам Android, у MainActivity прописана аннотация @AndroidEntryPoint

|  |
| --- |
| @AndroidEntryPoint class MainActivity : ComponentActivity() {  ///  } |

## Локальная база данных Room

Для организации хэширования данных используется библиотека Room.

Room — это ORM (Object-Relational Mapping) библиотека от Android, упрощающая работу с базами данных SQLite. Он работает по принципу абстрагирования доступа к базе данных, предоставляя более удобный объектно-ориентированный интерфейс.

1. **Entity:** это классы, которые представляют таблицы в базе данных. Они аннотируются (@Entity), и поля класса соответствуют столбцам таблицы.
2. **DAO (Data Access Object):** это интерфейсы (или абстрактные классы), которые определяют методы доступа к базе данных. Эти методы аннотируются (@Insert, @Update, @Delete, @Query), и Room генерирует реализацию этих методов, которые выполняют запросы к базе данных.
3. **Database:** Это абстрактный класс (@Database), который объявляет все Entity, используемые в приложении, и содержит метод getDao(), возвращающий экземпляр DAO. Room генерирует реализацию этого класса, которая обрабатывает создание и управление базой данных.

В essence, вы определяете структуру данных (Entity), методы доступа к ним (DAO) и объединяете все это в единый объект базы данных (@Database). Room берет на себя всю работу по преобразованию объектно-ориентированного кода в SQL-запросы и обратно, что значительно упрощает разработку и повышает безопасность.

Для генерации аргумента типа RoomDatabase() была прописана в классе с аннотацией @Module функция, возвращающая объект типа StudyBuddyDatabase.

|  |
| --- |
| @Provides fun provideTaskDb(  @ApplicationContext  context: Context ) = Room.databaseBuilder(  context,  StudyBuddyDatabase::class.*java*,  "tasks" ).allowMainThreadQueries().build() |

## Подключение к API

Для подключения к API используется Ktor Client.

Создан модуль, в котором описана инструкция (функция), как генерировать аргумент типа HttpClient. В этой функции идёт настройка KtorClient.

|  |
| --- |
| @Module @InstallIn(SingletonComponent::class) class ApiServiceProvider {   @Provides  fun provideClient(): HttpClient {  return HttpClient(Android)**{** expectSuccess = true  install(Logging) **{** level = LogLevel.*ALL* **}** install(ContentNegotiation)**{** *json*(  *Json* **{** encodeDefaults = false  ignoreUnknownKeys = true  isLenient = true  useAlternativeNames = false  **}**)  register(  ContentType.Text.Html, KotlinxSerializationConverter(  *Json* **{** prettyPrint = true  isLenient = true  ignoreUnknownKeys = true  **}** )  )  **}  }** }  ///  } |

Был создан интерфейс, в котором описаны будущие методы для обращения к API и на его основе был реализован класс, в котором реализованы все методы подключения к API.

|  |
| --- |
| */\*\* Реализация интерфейса, в котором описаны все методы для запросов к API + их кэшировние в локальную базу данных Room\*/* class ApiServiceImpl(  private val client: HttpClient,  private val database: StudyBuddyDatabase,  ): ApiService {  ///Методы подключения к API  } |

Объект данного класса используется во viewModel, поэтому необходимо прописать провайдера в модуль Hilt.

|  |
| --- |
| @Provides fun provideService(client: HttpClient, database: StudyBuddyDatabase): ApiServiceImpl {  return ApiServiceImpl(client, database) } |

И теперь с помощью Hilt во ViewModel легко внедрять зависимости для подключения к API и для обращения к локальной БД, также легко передавать контекст приложения с помощью аннотации @ApplicationContext.

|  |
| --- |
| @HiltViewModel class ExamsViewModel @Inject constructor(  private val service: ApiServiceImpl,  @ApplicationContext private val context: Context,  private val database: StudyBuddyDatabase, ) : ViewModel() {  ///  } |

## Функции конвертации даты

API возвращает дату в том формате, в котором она хранится в БД. Пользователю удобнее показывать дату в словесном формате или другом, более удобном для распознавания. Для конвертации даты из формата БД в другие (а конкретно: DD.MM.YYYY и словесный формат) написаны следующие функции:

|  |
| --- |
| fun FormatDateDBToDDMMYYYY(inputDate: String): String {  return try {  val inputFormat = SimpleDateFormat("yyyy-MM-dd", Locale.getDefault())  val outputFormat = SimpleDateFormat("dd.MM.yyyy", Locale("ru"))  val date = inputFormat.parse(inputDate)  val formattedDate = outputFormat.format(date)  formattedDate  } catch (e: Exception) { inputDate } } |
| fun FormatDateDBToDMMMMYYYY(inputDate: String): String {  return try {  val inputFormat = SimpleDateFormat("yyyy-MM-dd", Locale.getDefault())  val outputFormat = SimpleDateFormat("d MMMM yyyy", Locale("ru"))  val date = inputFormat.parse(inputDate)  val formattedDate = outputFormat.format(date)  formattedDate  } catch (e: Exception) { inputDate } } |

## Функции валидации и регулярные выражения

На экранах регистрации и авторизации есть текстовые поля с почтой и никнеймом. Почта должна соответствовать паттерну «name@domenname.ru», а никнейм может состоять только из латиницы и кириллицы. Для того, чтобы проверять данные поля на соответствие паттерну написано два метода, содержащих регулярные выражения и встроенную в андроид функцию проверки для почты.

|  |
| --- |
| fun String.isEmailValid () : Boolean {  return !TextUtils.isEmpty(this)  && android.util.Patterns.*EMAIL\_ADDRESS*.matcher(this)  .matches() } |
| fun String.isNicknameValid () : Boolean {  val regex = Regex("^[a-zA-Zа-яА-ЯёЁ\\s]+\$");  return regex.matches(this) } |

## Screen и ViewModel.

У каждого экрана приложения есть своя ViewModel. У каждого ViewModel есть свой объект состояния (state), который создаётся из потока MutableStateFlow, который помечен как изменяемый. К потоку state в экране подключается ещё один state, который помечен как collectAsState() и при изменении состояния из ViewModel, объекты @Composable, привязанные к state, обновляются. ViewModel передаётся в экран как аргумент, но при помощи Hilt, генерируется с помощью Dagger и при вызове функции экрана из Navigation можно не передавать аргумент ViewModel.

Пример класса состояния:

|  |
| --- |
| data class RegisterSt(  var nickname: String = "",  var email: String = "",  var password: String = "",  var confirmPassword: String = "", ) |

Пример экрана:

|  |
| --- |
| */\*\* Экран регистрации \*/* @Composable fun Register(controller: NavHostController, viewModel: RegisterViewModel = hiltViewModel()) {   val state = viewModel.state.collectAsState()  //UI  } |

Пример ViewModel:

|  |
| --- |
| @HiltViewModel class RegisterViewModel @Inject constructor(  private val service: ApiServiceImpl,  @ApplicationContext private val context: Context ): ViewModel() {   private val \_state = *MutableStateFlow*(RegisterSt())  val state: StateFlow<RegisterSt> = \_state.*asStateFlow*()   var stateValue: RegisterSt  get() = \_state.value  set(value) {  \_state.value = value  }  //остальные функции  } |

# ГЛАВА 6. ТЕСТИРОВАНИЕ МОБИЛЬНОГО ПРИЛОЖЕНИЯ

Документация по тестированию мобильного приложения представлена в [Приложении 4](#_Приложение_4).

# ЗАКЛЮЧЕНИЕ

В заключение работы следует констатировать успешную реализацию поставленной цели – создание мобильного приложения, оптимизирующего доступ к информации, необходимой для эффективного функционирования учебного процесса. Проведенное исследование выявило актуальность проблемы фрагментации информационных ресурсов образовательных учреждений, что обусловило целесообразность разработки интегрированного решения. В рамках проектной работы были успешно выполнены все запланированные задачи: разработан интуитивно понятный пользовательский интерфейс приложения, обеспечивающий удобную навигацию и доступ к необходимым данным; создана надежная и масштабируемая серверная архитектура, гарантирующая стабильность работы приложения и защиту данных; проведено всестороннее тестирование как серверной части, так и клиентского приложения, что подтвердило соответствие разработанного продукта заявленным требованиям к функциональности, производительности и безопасности; подготовлены отчет и презентация, детально описывающие архитектуру, функциональные возможности и результаты тестирования разработанного приложения. Разработанное мобильное приложение представляет собой инновационное решение, способствующее повышению эффективности учебного процесса за счет централизации доступа к информации о расписании, учебных материалах и других важных данных. Перспективы дальнейшего развития проекта включают интеграцию с другими информационными системами образовательного учреждения и расширение функциональных возможностей приложения, в том числе, внедрение персонализированных уведомлений. Таким образом, результаты данной дипломной работы демонстрируют успешное применение современных технологий для решения актуальной проблемы в сфере образования, внося вклад в оптимизацию учебного процесса и повышение его эффективности.

# Приложение 1

Техническое задание

**ОСНОВНАЯ ЧАСТЬ**

1. **Общие сведения.**
   1. **Наименование системы**

Полное наименование системы – «Study Buddy».

Краткое наименование системы – «SB».

1. **Цели и задачи.**

В настоящее время многие люди в мире получают образование. У многих образовательных учреждений есть свои сайты с расписанием звонков, предметов и другой информации. Но не у всех есть свой собственных сайт для получения заданий по предметам, а если и есть, то это уже другая ссылка. Также не у всех сайтов учебных учреждений поиск информации удобен и понятен, иногда приходится долгое время искать по всему сайту нужную информацию. Поэтому было бы удобнее, если бы всё необходимое находилось в одном месте. Мобильное приложение, с помощью которого можно будет реализовать данную потребность может стать решением данной проблемы.

Цель разработки: создать мобильное приложение, которое будет упрощать процесс поиска информации, связанной с учебными процессами.

Для выполнения поставленной цели необходимо выполнение следующих задач:

* Разработать макет приложения;
* Разработать серверную часть приложения;
* Провести тестирование серверной части приложения;
* Разработать мобильное приложение;
* Провести тестирование мобильного приложения;
* Подготовить отчёт и презентацию о разработанном продукте;
* Презентовать разработанный продукт перед заказчиком.

1. **Требования к серверной части.**
   1. **Функциональные требования.**

Перечень функциональных требований к серверной части:

* Реализация на Framework ASP.NET;
* Аутентификация пользователей: регистрация, авторизация;
* Хранение информации об экзаменах:
* Хранение информации о предметах: добавление в БД, изменение, получение из БД, удаление из БД
* Хранение заданий: добавление в БД, изменение, получение из БД, удаление из БД
* Хранение расписания: добавление в БД, изменение, получение из БД, удаление из БД
* Наличие API для получения расписания
* Наличие API для авторизации в системе и работы с данными основной БД.
  1. **Нефункциональные требования.**

Перечень нефункциональных требований к серверной части:

* Хэширование паролей и безопасное хранение данных для авторизации пользователей.
* Расширяемость системы (возможность работать с большим количеством пользователей).

1. **Требования к мобильному приложению.**
   1. **Функциональные требования**

* Навигация по приложению (переход между экранами).
* Регистрация пользователей: возможность пользователям регистрироваться, вводя свои данные, такие как имя, адрес электронной почты и пароль. Это должно позволить создать учетную запись и получить доступ к персонализированным функциям системы.
* Авторизация и аутентификация: проверка учетных данных пользователей при входе и доступ только для авторизованных пользователей. Это будет обеспечивать безопасность и конфиденциальность данных пользователей.
* Управление данными: просмотр, создание, изменение, удаление и редактирование данных через мобильное приложение.
* Поиск и фильтрация: возможность поиска задач, предметов и экзаменов по названию, сортировать по дате добавления по убыванию или возрастанию, сортировать по алфавиту. Это поможет пользователям быстро находить нужную информацию.
  1. **Нефункциональные требования:**
     1. **Основные требования.**
* Пользовательский интерфейс: простота использования, интуитивно понятный дизайн;
* Быстрая загрузка приложения, быстрое выполнение операций.
  + 1. **Требования в вёрстке страниц.**

Приложение должно подстраиваться под разные размеры экранов, иметь прокрутку на тех экранах, где это необходимо, позволять пользователю через системные настройки изменять размер шрифта и при этом на экранах не должно ничего съезжать или обрезаться.

* + 1. **Цветовая схема и шрифты:**

Цветовая схема и шрифты приложения представлены на Рисунке 1 и 2 ниже.

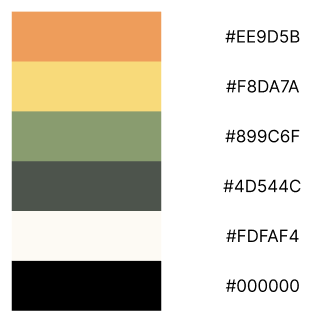


Рисунок 1. Цветовая схема приложения



Рисунок 2. Шрифты приложения

# Приложение 2

Выбор ПО и паттерна проектирования

1. **ВЫБОР ПО ДЛЯ РЕАЛИЗАЦИИ.**
   1. **Мобильное приложение.**

Android Studio будет использоваться в качестве среды разработки для мобильного приложения. Android Studio – это официальная среда разработки от OpenAI для Android, которая предлагает множество преимуществ для разработки приложений:

* Интеграция с Android SDK: Android Studio тесно интегрирована с Android SDK, что значительно упрощает процесс разработки и отладки.
* Удобные инструменты: Android Studio предоставляет богатый набор инструментов, таких как визуальный редактор макетов, инструменты профилирования и отладки, а также эмулятор Android.
* Система сборки Gradle: Android Studio использует систему сборки Gradle, которая обеспечивает гибкость и контроль над процессом сборки приложения.
* Инструменты для тестирования: Android Studio предоставляет инструменты для автоматического тестирования (Android Test и Espresso), что позволяет разработчикам убедиться в качестве кода.
* Система управления версиями: Android Studio интегрирована с системами управления версиями, такими как Git, что облегчает совместную разработку и отслеживание изменений.

Для написания кода приложения будет использоваться Jetpack Compose с использованием языка Kotlin. Такой выбор был сделан по ряду причин:

* Современный декларативный UI-фреймворк: Jetpack Compose – это современный UI-фреймворк от OpenAI, который предлагает декларативный подход к созданию пользовательского интерфейса, что делает код более лаконичным и читаемым.
* Упрощение UI: Jetpack Compose упрощает разработку UI, так как позволяет описывать UI в виде композиции функций.
* Интеграция с другими компонентами Jetpack: Jetpack Compose хорошо интегрируется с другими компонентами Jetpack, такими как ViewModel и LiveData, что делает разработку более структурированной.
* Поддержка Material Design: Jetpack Compose включает в себя поддержку Material Design, что позволяет создавать приложения с современным и привлекательным интерфейсом.
* Простота изучения: Jetpack Compose относительно прост в изучении, что сокращает время разработки и позволяет быстро приступить к работе.
* Минимальный код: Jetpack Compose позволяет создавать UI с меньшим количеством кода по сравнению с традиционными методами разработки UI на Android.
  1. **Серверная часть.**

Для реализации серверной части необходима база данных и API, с помощью которого можно будет управлять данными базы при помощи телефона.

Данные будут храниться на сервере НГК в СУБД Postgre SQL, а API будет разработана с помощью фреймворка ASP.NET.

Такой выбор был сделан из-за ряда преимуществ:

* Надежность и стабильность: PostgreSQL известна своей высокой стабильностью и надежностью, что делает ее отличным выбором для приложений.
* Транзакционная целостность: PostgreSQL поддерживает ACID-свойства (атомарность, согласованность, изоляция, долговечность), гарантируя целостность данных при одновременном доступе.
* Широкие возможности моделирования данных: PostgreSQL поддерживает сложные типы данных, в том числе JSON, геопространственные данные, массивы и пользовательские типы.
* Высокая производительность: PostgreSQL обеспечивает высокую производительность и масштабируемость, оптимизирована для работы с большими объемами данных.
* Открытый исходный код: PostgreSQL бесплатна для использования и распространения, что делает ее доступной для разработчиков всех уровней.
* Гибкость: ASP.NET Core предоставляет широкие возможности для кастомизации, чтобы создать API, идеально соответствующее требованиям.
* Масштабируемость: возможность легко масштабировать приложение, добавляя серверы и ресурсы по мере роста нагрузки.
* Опыт: ASP.NET - устоявшийся фреймворк с большим сообществом и множеством ресурсов.
* Открытый исходный код: PostgreSQL и ASP.NET Core бесплатны и открыты, что дает свободу использования и модификации.

1. **ВЫБОР ПАТТЕРНА ПРОЕКТИРОВАНИЯ.**

Выбранный паттерн проектирования: MVVM (Model-View-ViewModel)

MVVM (Model-View-ViewModel) – это архитектурный паттерн, который широко применяется в разработке мобильных приложений. Он разделяет приложение на три основных компонента:

1. Модель (Model): представляет данные приложения, логику доступа к данным (например, работа с базой данных) и правила валидации данных.
2. Представление (View): отображает данные и взаимодействует с пользователем. Это то, что видит пользователь (например, экраны, элементы UI).
3. Модель представления (ViewModel): является посредником между моделью и представлением. Он обрабатывает запросы от представления, получает данные от модели, форматирует их и передает обратно в представление. ViewModel не должен иметь зависимостей от View, поэтому его можно использовать для тестирования и повторного использования.

Почему выбор пал на MVVM для разработки мобильного приложения:

* Разделение ответственности: MVVM помогает разделить ответственность между разными компонентами приложения, что делает код более структурированным и легко поддерживаемым.
* Тестируемость: MVVM делает приложение более тестируемым, так как ViewModel не зависит от View и может быть тестирован отдельно.
* Повторное использование: ViewModel можно повторно использовать в разных частях приложения или в других приложениях.
* Обновление UI: MVVM делает обновление UI более простым, так как ViewModel уведомляет View об изменениях данных с помощью механизмов наблюдения.
* Совместимость с Jetpack Compose: MVVM хорошо интегрируется с Jetpack Compose, позволяя создавать более структурированные и тестируемые UI.

# Приложение 3

Документация тестирования серверной части приложения

# 1. Тест план.

# Перечень работ.

Перечень функциональных областей API, которые будут подвергаться тестированию:

1. Ввод данных: проверка корректности ввода данных, включая проверку на допустимые значения (например, при некорректном вводе даты или не заполненном обязательном поле должна вернуться соответствующая ошибка).
2. Авторизация в системе: возможность создать аккаунт, авторизоваться и получать данные об авторизованном пользователе.
3. Интерфейс API: можно ли делать запросы прямо на странице API.
4. Совместимость: проверка совместимости с другими приложениями, например, какие данные при подключении к API можно получить через Postman.
5. Безопасность: доступ к закрытым методам только для авторизованных пользователей, получение только информации авторизованного пользователя.
6. Документация: проверка наличия и качества документации, включая понятное описание API и всех методов.

# Критерии качества.

1. Надежность: способность программного продукта выполнять свои функции без сбоев в течение определенного времени или количества операций.
2. Безопасность: обеспечение защиты информации и ресурсов от несанкционированного доступа, изменений.
3. Производительность: способность программного продукта эффективно выполнять свои функции в разумные сроки и с минимальным потреблением ресурсов.
4. Масштабируемость: способность программного продукта адаптироваться к увеличению объема данных или количества пользователей без значительного снижения производительности.
5. Удобство использования: уровень удобства и простоты использования программного продукта для пользователя, включая понятный интерфейс, интуитивную навигацию и доступность функций.
6. Сопровождаемость: способность программного продукта быть легко модифицируемым, исправляемым и обновляемым без необходимости переписывания большого количества кода.
7. Поддерживаемость: возможность программного продукта быть поддержанным и обслуживаемым в течение всего его жизненного цикла, включая обновления, исправления ошибок и обучение пользователей.
8. Эффективность: способность программного продукта выполнять свои функции с минимальным использованием ресурсов, таких как память, процессорное время и сетевой трафик.

# Оценка рисков.

Таблица 2. Оценка рисков

|  |  |  |  |
| --- | --- | --- | --- |
| Риск | Оценка риска | Описание риска | Выход из ситуации |
| Некорректный ввод данных | Высокий | Неправильный ввод данных может привести к ошибочным результатам и недовольству пользователей. | Провести тщательную проверку ввода данных, проверяя возможность ввода пустой строки, null, отрицательных значений (для числовых полей), специальных символов, отсутствие поля в json запросе |
| Нестабильная работа приложения | Средняя | Нестабильная работа приложения может привести к потере данных и недовольству пользователей. | Провести тщательное тестирование на различных устройствах и в различных условиях использования - использовать механизмы отладки и мониторинга для выявления и исправления ошибок |
| Низкая производительность | Средняя | Низкая производительность приложения может привести к недовольству пользователей. | Оптимизировать код, использовав эффективные алгоритмы - произвести тестирование производительности и мониторинг нагрузки на приложение. |

# Стратегия тестирования. (GUI & Ux-testing, Positive, Negative, White-box, Black-box).

Стратегия тестирования – это план действий, который определяет, как будет проводиться тестирование приложения. Она включает в себя выбор методов и инструментов тестирования, а также определение приоритетов и целей тестирования.

GUI & UX-testing – это тестирование интерфейса пользователя и его удобства использования. Оно включает проверку навигации, доступности элементов управления и понятности сообщений об ошибках.

Positive testing – это тестирование, при котором приложение проверяется на корректную работу при вводе правильных данных и выполнении ожидаемых функций.

Negative testing – это тестирование, при котором приложение проверяется на работу при вводе неправильных или некорректных данных.

White-box testing – это тестирование, при котором проверяются внутренние компоненты и функциональность приложения. Оно включает проверку алгоритмов, обработку ошибок и безопасность.

Black-box testing – это тестирование, при котором проверяется работа приложения без знания его внутренних компонентов и функциональности. Оно включает проверку интерфейса пользователя, совместимости с различными операционными системами и браузерами, а также проверку производительности и надежности.

# Ресурсы.

Для разработки серверной части приложения StudyBuddy могут быть задействованы следующие ресурсы:

Человеческие ресурсы: человек, который будут заниматься разработкой API.

Механические ресурсы: это оборудование, которое будет использоваться для разработки приложения. Это рабочее место студента (АРМ) в колледже, его рабочее место дома.

Технические ресурсы: это программное обеспечение и инструменты, которые будут использоваться для разработки приложения. Это язык программирования C#, свободно распространяемая платформа для разработки динамических сайтов и веб-приложений ASP.NET, являющаяся частью NET.Framework, Nu Get пакеты для авторизации через JWT токен, для подключения к БД, для генерации кода (специальных таблиц для авторизации), и др., среда разработки Visual Studio, инструменты тестирования, наличие системы удаленных версий.

Информационные ресурсы: это данные, которые будут использоваться для разработки приложения. Видеоуроки и статьи в интернете, гайды по созданию API на ASP.NET с авторизацией через JWT токен, гайды по Entity Framework и как создавать БД через код, как документировать Swagger и др.

Организационные ресурсы: это структура и процессы, которые будут использоваться для разработки приложения. Это включает в себя управление проектом и планирование.

# Метрики

Точность: Программа должна правильно регистрировать пользователя в системе и выводить только данные авторизованного пользователя. Это может быть измерено путем ручного тестирования авторизации пользователя в системе и проверке получаемых данных через методы в API, требующие авторизации.

Скорость: Программа должна быть быстрой и эффективной в обработке данных.

Надежность: Программа должна быть надежной и стабильной, то есть она не должна вылетать или вызывать ошибки при обработке данных. Это может быть измерено путем тестирования программы на различных наборах данных и проверки ее стабильности.

Использование ресурсов: Программа должна потреблять минимальное количество ресурсов, таких как память и процессорное время.

# Расписание и ключевые точки.

Таблица 3. Расписание и ключевые точки.

|  |  |
| --- | --- |
| **Этап** | **Количество пар** |
| Разработка тестового набора и тест-кейсов | 1 |
| Реализация 10 unit-test к API | 1 |

# Тестовый набор

Тестовый набор – это набор тест-кейсов, в которых результат описывается предисловием, то есть очередность проводимых тестов.

* 1. **Выбор фреймворка для тестирования**

Для тестирования API будет использоваться инструмент для модульного тестирования xUnit. xUnit — это пакет с открытым исходным кодом для .NET Framework и .NET Core. Кроме фреймворков для создания и проведения юнит-тестов при тестировании часто бывают полезны такие фреймворки, которые позволяют имитировать или эмулировать какую-то функциональность или создавать мок-объекты. Подобных фреймворков существует множество, и одним из самых популярных является Moq, он и будет использоваться. Также будет использоваться модель тестов Arrange-Act-Assert, которая представляет целую парадигму тестирования, которая используется многими фреймворками юнит-тестов.

* 1. **Проверка аутентификации и авторизации при помощи модульного тестирования.**

В API присутствует система авторизации, поэтому необходимо проверить, корректно ли она работает. Проверка реализована в виде unit тестов, где для изоляции компонентов друг от друга и имитации их взаимодействия используются моки (тестовые объекты, которые имитируют зависимости), также создан тестовый контекст БД. Тестированию подвержен контроллер Account, который отвечает за обработку операций, связанных с учетными записями пользователей, таких как: регистрация, вход в систему, получение информации об авторизованном пользователе. В Таблице 1 ниже указаны тест-кейсы.

Таблица 1. Тест-кейсы для unit-тестов системы авторизации.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | Проверка успешной регистрации нового пользователя с валидными данными | 1. Имя пользователя: Mark.  2. Почта: [markmarkovich@mail.ru](mailto:markmarkovich@mail.ru)  3. Пароль: 12345678 | – Возвращается код HTTP 200 (Ok).  – Создается новый пользователь в базе данных и возвращается. | result.Should().NotBeNull();  Assert.NotNull(newUserInContext); result.Should().BeAssignableTo<OkObjectResult>(); | + |
| 2 | Проверка, что при попытке регистрации с уже существующим именем пользователя возвращается ошибка. | 1. Имя пользователя: Mark.  2. Почта: [markmarkovich1@mail.ru](mailto:markmarkovich1@mail.ru)  3. Пароль: 12345678 | – Возвращается код HTTP 400 (BadRequest). | result.Should().NotBeNull();  result.Should().BeAssignableTo<BadRequestObjectResult>(); | + |
| 3 | Проверка успешной аутентификации пользователя с правильными логином и паролем. | 1. Почта: [markmarkovich1@mail.ru](mailto:markmarkovich1@mail.ru)  2. Пароль: 12345678 | - Возвращается код HTTP 200 (Ok). - Генерируется и возвращается JWT-токен. | result.Should().NotBeNull();  Assert.NotNull(resultDto.Token);  result.Should().BeAssignableTo<OkObjectResult>(); | + |
| 4 | Проверка, что при попытке авторизоваться с неправильными логином/паролем возвращается ошибка. | - Имя пользователя: invaliduser - Пароль: invalidpassword | - Возвращается код HTTP 401 (Unauthorized). | result.Should().NotBeNull();  result.Should().BeAssignableTo<UnauthorizedObjectResult>(); | + |
| 5 | Проверка получения информации об авторизованном пользователе | Эмуляция авторизации в системе (через моки) | - Возвращается код HTTP 200 (Ok).  - Возвращается модель данных об аккаунте | result.Should().NotBeNull();  Assert.NotNull(resultDto);  result.Should().BeAssignableTo<ActionResult<UserInfoDto>>(); | + |

* 1. **Проверка работы с данными у авторизованного пользователя.**

После авторизации в системе пользователю доступен ряд GET, POST, DELETE и PUT методов для управления информацией под своей учетной записью:

GET-запросы: используются для получения данных, например, списка экзаменов, информации о профиле пользователя, получение списка требований к предмету и др. Тесты должны проверять, что авторизованный пользователь получает только информацию, относящуюся к его аккаунту.

POST-запросы: используются для создания новых данных, например, добавления нового экзамена. Тесты должны проверять, что данные создаются корректно и привязываются к аккаунту пользователя.

DELETE-запросы: используются для удаления данных, например, удаления экзамена. Тесты должны проверять, что данные действительно удаляются.

PUT-запросы: используются для обновления данных, например, изменения информации об экзамене. Тесты должны проверять, что данные обновляются корректно.

Для проверки были написаны тест-кейсы, они указаны в Таблице 2.

Таблица 2. Тест-кейсы для unit-тестов работы с данными пользователя.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | GET - Получение списка экзаменов пользователя | 1. Эмуляция авторизации в системе (через моки). | – Возвращается код HTTP 200 (Ok).  – Список экзаменов у пользователя | result.Should().NotBeNull();  Assert.NotNull(resultDto);  result.Should().BeAssignableTo<ActionResult<List<Exam>>>(); | + |
| 2 | POST – создание экзамена у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Объект создаваемого экзамена | – Возвращается код HTTP 200 (Ok).  – Добавленный экзамен у пользователя  – Количество экзаменов пользователя меняется в большую сторону | Assert.NotNull(resultDto);  Assert.True(countExamsBefore < countExamsAfter);  result.Should().NotBeNull();  result.Should().BeAssignableTo<ActionResult<Exam>>(); | + |
| 3 | POST – создание экзамена дубликата у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Объект создаваемого экзамена, который уже есть в БД | - Возвращается код HTTP 400 (BadRequest). | Assert.True(result.Result is BadRequestObjectResult); | + |
| 4 | DELETE –удаление экзамена у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Id экзамена, который будет удален | – Возвращается код HTTP 200 (Ok). | result.Should().NotBeNull();  result.Should().BeAssignableTo<ActionResult>(); | + |
| 5 | PUT – изменение экзамена у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Объект экзамена, который будет изменён | - Возвращается код HTTP 200 (Ok). | result.Should().NotBeNull();  result.Should().BeAssignableTo<ActionResult>(); | + |

# Приложение 4

Документация тестирования мобильного приложения

# 1. Тест план.

# Перечень работ.

Перечень функциональных областей приложения, которые будут подвергаться тестированию:

1. Пользовательский интерфейс: проверка отображения ошибок и самих экранов.
2. Конвертеры: проверка функций, отвечающих за преобразование одного вида даты в другой.
3. Функции валидации данных: проверка регулярных выражений для почты и никнейма.
4. Авторизация в системе: возможность создать аккаунт и авторизоваться
5. Навигация: можно ли переключать страницы.
6. Совместимость: проверка совместимости с разными версиями Андроид.

# Критерии качества.

1. Надежность: способность программного продукта выполнять свои функции без сбоев в течение определенного времени или количества операций.
2. Безопасность: обеспечение защиты информации и ресурсов от несанкционированного доступа, изменений.
3. Производительность: способность программного продукта эффективно выполнять свои функции в разумные сроки и с минимальным потреблением ресурсов.
4. Масштабируемость: способность программного продукта адаптироваться к увеличению объема данных или количества пользователей без значительного снижения производительности.
5. Удобство использования: уровень удобства и простоты использования программного продукта для пользователя, включая понятный интерфейс, интуитивную навигацию и доступность функций.
6. Сопровождаемость: способность программного продукта быть легко модифицируемым, исправляемым и обновляемым без необходимости переписывания большого количества кода.
7. Поддерживаемость: возможность программного продукта быть поддержанным и обслуживаемым в течение всего его жизненного цикла, включая обновления, исправления ошибок и обучение пользователей.
8. Эффективность: способность программного продукта выполнять свои функции с минимальным использованием ресурсов, таких как память, процессорное время и сетевой трафик.

# Оценка рисков.

Таблица 2. Оценка рисков

|  |  |  |  |
| --- | --- | --- | --- |
| **Риск** | **Оценка риска** | **Описание риска** | **Выход из ситуации** |
| Некорректный ввод данных | Высокий | Неправильный ввод данных может привести к ошибочным результатам и недовольству пользователей. | Провести тщательную проверку ввода данных, проверяя возможность ввода пустой строки, null, отрицательных значений (для числовых полей), специальных символов, проверить регулярные выражения для ввода числовых значений |
| Нестабильная работа приложения | Средняя | Нестабильная работа приложения может привести к потере данных и недовольству пользователей. | Провести тщательное тестирование на различных устройствах и в различных условиях использования - использовать механизмы отладки и мониторинга для выявления и исправления ошибок |
| Низкая производительность | Средняя | Низкая производительность приложения может привести к недовольству пользователей. | Оптимизировать код, использовав эффективные алгоритмы - произвести тестирование производительности и мониторинг нагрузки на приложение. |

# Стратегия тестирования.

Приложение планируется протестировать, применив следующие стратегии:

1. UI – тестирование.
2. Негативное тестирование - тестирование интерфейса пользователя и его удобства использования. Оно включает проверку навигации, доступности элементов управления и понятности сообщений об ошибках.
3. Позитивное тестирование - тестирование, при котором приложение проверяется на корректную работу при вводе правильных данных и выполнении ожидаемых функций.
4. Тестирование белого ящика - тестирование, при котором проверяются внутренние компоненты и функциональность приложения. Оно включает проверку алгоритмов, обработку ошибок и безопасность.
5. Тестирование черного ящика - тестирование, при котором проверяется работа приложения без знания его внутренних компонентов и функциональности. Оно включает проверку интерфейса пользователя, совместимости с различными операционными системами и браузерами, а также проверку производительности и надежности.

Стратегия тестирования – это план действий, который определяет, как будет проводиться тестирование приложения. Она включает в себя выбор методов и инструментов тестирования, а также определение приоритетов и целей тестирования.

# Ресурсы.

Для разработки мобильного приложения StudyBuddy могут быть задействованы следующие ресурсы:

Человеческие ресурсы: человек, который будут заниматься разработкой приложения.

Механические ресурсы: это оборудование, которое будет использоваться для разработки приложения. Это рабочее место студента (АРМ) в колледже, рабочее место студента дома.

Технические ресурсы: это программное обеспечение и инструменты, которые будут использоваться для разработки приложения. Это язык программирования Kotlin, современный фреймворк для создания пользовательских интерфейсов на платформе Android Jetpack Compose, библиотеки для тестирования, среда разработки Android Studio, инструменты тестирования, наличие системы удаленных версий.

Информационные ресурсы: это данные, которые будут использоваться для тестирования приложения. Видеоуроки и статьи в интернете, гайды по тестированию пользовательского интерфейса.

# Метрики

Точность: Программа должна правильно регистрировать пользователя в системе и выводить только данные авторизованного пользователя. Это может быть измерено путем ручного тестирования авторизации пользователя в системе и проверке, всегда ли пользователь получает свои данные после авторизации, нет ли случайного отображения данных пользователя, который входил ранее.

Скорость: Программа должна быть быстрой и эффективной в обработке данных.

Надежность: Программа должна быть надежной и стабильной, то есть она не должна вылетать или вызывать ошибки при обработке данных. Это может быть измерено путем тестирования программы на различных наборах данных и проверки ее стабильности.

Использование ресурсов: Программа должна потреблять минимальное количество ресурсов, таких как память и процессорное время.

# Расписание и ключевые точки.

Таблица 3. Расписание и ключевые точки.

|  |  |
| --- | --- |
| **Этап** | **Количество пар** |
| Разработка тестового набора и тест-кейсов | 1 |
| Реализация 5 unit-test и 5 UI-test к приложению | 1 |

# Тестовый набор

Тестовый набор – это набор тест-кейсов, в которых результат описывается предисловием, то есть очередность проводимых тестов.

* 1. **Выбор фреймворка для тестирования**

Для тестирования будут использованы следующие инструменты:

1. Фреймворк JUnit 4 - популярный фреймворк для написания юнит-тестов в Java (и других языках, поддерживающих JVM, Kotlin в том числе).
2. Аннотации: для обозначения методов как тестов (@Test), методов настройки перед тестом (@Before), методов очистки после теста (@After), методов настройки всего тестового набора (@BeforeClass), методов очистки всего тестового набора (@AfterClass) и др. Эти аннотации упрощают организацию и управление тестами.
3. Ассерты (Assertions): Методы для проверки ожидаемых результатов. Например, assertEquals(), assertTrue(), assertFalse() и другие, позволяют сравнивать фактические результаты с ожидаемыми и сообщать об ошибках, если они не совпадают.
4. Средства тестирования производительности в Android Studio
   1. **Проверка валидации данных при помощи модульного тестирования.**

В мобильном приложении есть регистрация и авторизация, в которых присутствуют поля, которые необходимо проверять на клиенте, перед отправкой на сервер в POST запросе, такие как почта и никнейм пользователя. Почта должна соответствовать паттерну. В Таблице 4 ниже указаны тест-кейсы.

Таблица 4. Тест-кейсы для unit-тестов системы авторизации.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | Проверка соответствия паттерну почты строки, которая соответствует паттерну почты (Позитивный тест) | 1. Почта: user@mail.ru. | – Возвращается результат проверки истина - true | Assert.assertEquals(expectedResultAfterCheckEmail, actualResultAfterCheck) | + |
| 2 | Проверка несоответствия паттерну почты строки, которая не соответствует паттерну почты (Негативный тест) | 1. Почта: usermail.ru. | – Возвращается результат проверки ложь - false | Assert.assertEquals(expectedResultAfterCheckEmail, actualResultAfterCheck) | + |
| 3 | Проверка соответствия паттерну для никнейма строки, которая состоит только из символов кириллицы и латиницы (Позитивный тест) | 1. Никнейм: NiCkНеЙм. | – Возвращается результат проверки истина - true | Assert.assertEquals(expectedResultAfterCheckNickname, actualResultAfterCheck | + |
| 4 | Проверка несоответствия паттерну для никнейму, которая состоит не только из символов кириллицы и латиницы (Негативный тест) | 1. Никнейм: \_(NiCkНеЙм)1. | – Возвращается результат проверки ложь - false | Assert.assertEquals(expectedResultAfterCheckNickname, actualResultAfterCheck) | + |

* 1. **Проверка функций-конвертеров даты.**

В мобильном приложении есть разные форматы отображения даты. Из API дата приходит в том же формате, что и хранится в базе данных (например, 2024-11-26), а значит конвертеры должны преобразовать эту дату в следующие форматы: 26 ноября 2024, 26.11.2024. В Таблице 5 ниже указаны тест-кейсы.

Таблица 5. Тест-кейсы для unit-тестов конвертеров.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | Проверка правильного преобразования даты из формата YYYY-MM-DD в формат даты, представленной в словесной форме | Строка: 2024-11-26. | Строка: 26 ноября 2024. | assertEquals(expectedFormattedDate, actualFormattedDate) | + |
| 2 | Проверка правильного преобразования даты из формата YYYY-MM-DD в формат DD.MM.YYYY | Строка: 2024-11-26. | Строка: 26.11.2024 | assertEquals(expectedFormattedDate, actualFormattedDate) | + |

* 1. **Тестирование пользовательского интерфейса на экранах авторизации и регистрации.**

В приложении при регистрации и авторизации есть определенное количество проверок полей, перед отправкой запроса на сервер, такие как: соответствие почты паттерну, наличие в никнейме только символов латиницы и кириллицы, пароли должны совпадать и так далее. Для того, чтобы пользователь понимал, какие проверки не были пройдены, отображается всплывающее сообщение пользователю в зависимости от ошибки. Корректность отображение этих сообщений и будет проверено. Также необходимо проверить контекст приложения и отображение экранов (хотя-бы 1). В Таблице 6 ниже указаны тест-кейсы.

Таблица 6. Тест-кейсы для ui-tests.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | Верификация корректного контекста приложения | Нет прямых входных данных (внутренний вызов) | Получен контекст приложения с правильным именем пакета | Утверждение, что имя пакета контекста приложения равно «com.example.studybuddy» (assertEquals) | + |
| 2 | Проверка отображения элементов на экране авторизации | Запуск экрана авторизации (Auth) | Отображение заголовка «Умное планирование учёбы», описания, кнопок «ВОЙТИ» и «СОЗДАТЬ ПРОФИЛЬ». | assertIsDisplayed() для каждого из элементов: заголовка, описания и двух кнопок. | + |
| 3 | Проверка отображения ошибки при пустых полях входа | Запуск экрана логина (Login) с пустыми полями логина и пароля | Отображение сообщения об ошибке «Не все поля заполнены» после нажатия кнопки «ВОЙТИ». | 1. Нажатие кнопки «ВОЙТИ» (performClick()).  2. Проверка отображения сообщения об ошибке (isDisplayed()). | + |
| 4 | Проверка обработки некорректного email при входе | Email: «erwer», пароль: «12345678» | Отображение сообщения об ошибке «Неверный формат почты» | 1. Ввод некорректного email.  2. Нажатие кнопки «ВОЙТИ». 3. Проверка отображения сообщения об ошибке (isDisplayed()). | + |
| 5 | Проверка обработки некорректного nickname при регистрации | Email: «[user@email.com](mailto:user@email.com)», пароль: «12345678», подтверждение пароля: «12345678», nickname: «NiCk0Name1» | Отображение сообщения об ошибке «В никнейме недопустимые символы» | 1. Ввод данных. 2. Нажатие кнопки «СОЗДАТЬ ПРОФИЛЬ». 3. Проверка отображения сообщения об ошибке (isDisplayed()). | + |
| 6 | Проверка обработки несовпадения паролей при регистрации | Email: «[user@email.com](mailto:user@email.com)», пароль: «12345678», подтверждение пароля: «1345678», nickname: «NiCkname» | Отображение сообщения об ошибке «Пароли не совпадают» | 1. Ввод данных. 2. Нажатие кнопки «СОЗДАТЬ ПРОФИЛЬ». 3. Проверка отображения сообщения об ошибке (isDisplayed()). | + |