**Билет №1**

**1. Основные платформы мобильных приложений Android и iOS: история развития, версии, отличия, архитектура.**

**2. Создать мобильное приложение для ввода данных о студенте (фамилия, имя, группа, курс, пол, дата рождения) с последующим выводом в список. При разработке приложения использовать принципы удобного ввода и проверки корректности данных. Список данных вывести на отдельной странице.**

**История развития**

**Android**

Запуск: Android был разработан компанией Android Inc., которую Google приобрела в 2005 году. Первая коммерческая версия Android (1.0) была выпущена в сентябре 2008 года.

Эволюция: С тех пор Android прошел через многочисленные обновления, включающие в себя новые функции, исправления безопасности и улучшения пользовательского интерфейса. Каждое обновление первоначально носило кодовое имя, основанное на десертах (например, Cupcake, Donut, Eclair), а затем перешло на номера версий.

**iOS**

Запуск: iOS (ранее назывался iPhone OS) был разработан Apple и впервые представлен вместе с первым iPhone в 2007 году.

Эволюция: С тех пор iOS получил множество обновлений, добавляющих новые функции и улучшения безопасности. Версии iOS носят простые числовые названия, например iOS 14, iOS 15.

**Основные версии**

**Android**

1.0 (2008): Первая версия, включающая базовые приложения и веб-браузер.

4.0 Ice Cream Sandwich (2011): Объединение смартфонов и планшетов в одну платформу.

5.0 Lollipop (2014): Новый пользовательский интерфейс Material Design.

10 (2019): Улучшения в безопасности, конфиденциальности и производительности.

12 (2021): Новый визуальный стиль и улучшенные настройки конфиденциальности.

**iOS**

1.0 (2007): Первая версия, включающая базовые функции звонков, SMS и веб-серфинга.

7 (2013): Полный редизайн интерфейса с более плоским дизайном.

11 (2017): Улучшение многозадачности и добавление функции ARKit.

14 (2020): Введение виджетов на домашнем экране и App Library.

15 (2021): Улучшения в FaceTime, фокусировка, новые уведомления.

**Отличия**

Интерфейс: iOS славится своим интуитивным и единообразным интерфейсом, в то время как Android предлагает большую гибкость и возможность настройки.

Приложения: Магазин приложений Apple (App Store) имеет строгие требования к приложениям, что обычно приводит к высокому качеству, тогда как Google Play Store предлагает больше свободы разработчикам.

Устройства: iOS работает исключительно на устройствах Apple (iPhone, iPad, iPod Touch), в то время как Android работает на широком спектре устройств от различных производителей.

Обновления: Apple контролирует все обновления для своих устройств, в то время как обновления Android зависят от производителей устройств и операторов связи.

**Архитектура**

**Android**

Ядро: Linux kernel

Библиотеки: Библиотеки C/C++ (например, libc, SSL)

Рамки приложений: Android Runtime (ART), библиотеки Java API

Приложения: Написаны на Java и Kotlin

**iOS**

Ядро: XNU kernel (основан на ядре Unix)

Библиотеки: Core Foundation, Cocoa Touch

Рамки приложений: UIKit, SwiftUI, Foundation

Приложения: Написаны на Swift и Objective-C

**Билет №2**

**1. Нативные, гибридные и кроссплатформенные приложения мобильных устройств: понятие, примеры, технология создания.**

**2. Создать мобильное приложение для ввода данных о студенте (фамилия, имя, пол, группа, курс) с последующим сохранением в базу данных SQLite. Сохраненные данные в базе данных вывести на отдельной странице мобильного приложения.**

**Нативные приложения**

Понятие: Нативные приложения создаются для конкретной платформы (iOS или Android) с использованием инструментов и языков программирования, предназначенных для этой платформы.

**Примеры:**

iOS: Instagram, WhatsApp, Spotify (написаны на Swift или Objective-C).

Android: Google Maps, Facebook, TikTok (написаны на Kotlin или Java).

**Технология создания:**

Языки программирования: Swift, Objective-C (iOS); Kotlin, Java (Android).

Инструменты разработки: Xcode (iOS), Android Studio (Android).

Преимущества: Высокая производительность, доступ ко всем возможностям устройства и платформы, более качественный пользовательский интерфейс.

Недостатки: Требуется отдельная разработка и поддержка для каждой платформы, что увеличивает затраты времени и ресурсов.

**Гибридные приложения**

Понятие: Гибридные приложения объединяют элементы нативных и веб-приложений. Они разрабатываются с использованием веб-технологий (HTML, CSS, JavaScript) и оборачиваются в нативный контейнер для запуска на мобильных устройствах.

**Примеры:**

Twitter, Uber, Evernote (разработаны с использованием гибридных фреймворков).

**Технология создания:**

Языки программирования: HTML, CSS, JavaScript.

Фреймворки: Apache Cordova, Ionic, PhoneGap.

Преимущества: Быстрая разработка, возможность кроссплатформенной поддержки с единой кодовой базой, снижение затрат.

Недостатки: Ограниченная производительность, доступ к нативным функциям может быть ограничен, менее качественный пользовательский интерфейс.

**Кроссплатформенные приложения**

Понятие: Кроссплатформенные приложения разрабатываются с использованием инструментов, которые позволяют создавать приложения для нескольких платформ (iOS и Android) из одной кодовой базы.

**Примеры:**

Airbnb, Alibaba, Skype (разработаны с использованием кроссплатформенных фреймворков).

**Технология создания:**

Языки программирования: Dart, C#, JavaScript.

Фреймворки: Flutter, React Native, Xamarin.

Преимущества: Единая кодовая база для всех платформ, сокращение времени разработки, снижение затрат.

Недостатки: Возможны ограничения в доступе к нативным функциям, производительность ниже, чем у нативных приложений.

**Билет №3**

**1. Веб-приложения мобильных устройств: понятие, технология создания веб приложения, технология создания веб-сервиса.**

**2. Создать мобильное приложение «Автопарк» с возможностью последовательно просматривать на экране изображения автомобилей и их кратким описанием. Отображение элементов в вертикальном прокручиваемом списке. Изображения и описания автомобилей использовать из каталога.**

Веб-приложения для мобильных устройств представляют собой приложения, которые запускаются на мобильных браузерах и не требуют установки на устройстве. Они предоставляют функционал, аналогичный нативным приложениям, используя технологии веб-разработки, такие как HTML, CSS и JavaScript.

**Технология создания веб-приложения**

Выбор технологии: Основные технологии для создания веб-приложений включают HTML, CSS и JavaScript. Выбор фреймворков, таких как Angular, React, или Vue.js, также важен для улучшения производительности и функциональности.

Разработка фронтенда: Создание пользовательского интерфейса, который адаптируется к различным экранам мобильных устройств.

Разработка бэкенда: Обработка данных и бизнес-логики приложения. Включает создание серверов, баз данных и API (интерфейсов программирования приложений).

Тестирование и отладка: Обеспечение качества и функциональности веб-приложения на различных устройствах и платформах.

Деплой и хостинг: Размещение веб-приложения на сервере и предоставление доступа пользователям через интернет.

**Технология создания веб-сервиса**

Выбор архитектуры: Определение архитектуры веб-сервиса, например, RESTful или SOAP, для обмена данными между клиентами и серверами.

Разработка API: Создание API для взаимодействия с клиентами. RESTful API, например, использует HTTP-запросы для выполнения операций с данными.

Интеграция с базой данных: Создание и настройка базы данных для хранения и управления данными, используемыми веб-сервисом.

Тестирование и обеспечение безопасности: Тестирование функциональности и безопасности веб-сервиса для защиты данных и предотвращения уязвимостей.

Документация: Подготовка документации для разработчиков, использующих веб-сервис, чтобы облегчить их интеграцию с сервисом.

**Билет №4**

**1. Публикация мобильного приложения: подготовка, компиляция, создание пакета.**

**2. Создать мобильное приложение для ввода данных о сотруднике (фамилия, имя, отчество, дата рождения, город проживания) на первом экране и выводом информации о сотруднике на втором экране (Фамилия И.О., возраст (полных лет), город проживания). Город проживания должен выбираться из списка: Москва, Киров, Казань, Томск, Пермь, Екатеринбург, Слободской, Воронеж.**

**Xamarin**

Настройки проекта(указать SDK, манифест, ключи) далее архивировать приложение после выложить в Google Play

**Android Studio**

Поменять тип сборки на релиз в Build – Select build variant, далее Build - Generate Signed APK там указать ключи.

**Xamarin**

**Подготовка**

Проверьте код на ошибки и исправьте их.

Обновите все используемые библиотеки и зависимости до последних версий.

Убедитесь, что у вас установлены все необходимые пакеты.

**Компиляция**

Откройте проект в Visual Studio.

Выберите конфигурацию сборки (Debug или Release).

Скомпилируйте проект (Build > Build Solution).

**Создание пакета**

Перейдите в меню Build и выберите "Archive for Publishing".

Создайте архив и откройте окно архивов.

Выберите "Distribute", укажите платформу и следуйте инструкциям для создания пакета.

Подписывание пакета и публикация в Google Play

Создайте новый сертификат подписи в Visual Studio, если его у вас еще нет.

Перейдите в "Project > Archive for Publishing".

Выберите "Distribute", затем "Ad Hoc" и нажмите "+" для создания нового ключа подписи.

Заполните необходимые поля и создайте ключ.

Подпишите APK-файл с помощью созданного сертификата.

Загрузите подписанный APK-файл в Google Play Console:

Войдите в Google Play Console и выберите ваш проект.

Перейдите в "Release management" > "App releases" и выберите тип выпуска (Production, Beta и т.д.).

Нажмите "Create release" и загрузите подписанный APK-файл.

Заполните все необходимые поля, включая описание, скриншоты и иконку.

Опубликуйте приложение, следуя инструкциям в Google Play Console.

**Android Studio**

**Подготовка**

Проверьте код на ошибки и исправьте их.

Обновите все зависимости и библиотеки.

Убедитесь, что у вас установлены все необходимые SDK.

**Компиляция**

Откройте проект в Android Studio.

Выберите конфигурацию сборки (Debug или Release).

Скомпилируйте проект (Build > Make Project).

Создание пакета

Перейдите в меню Build и выберите "Build Bundle(s) / APK(s)".

Выберите "Build APK(s)" или "Build Bundle(s)".

После завершения сборки перейдите в "Build > Locate APK/Bundles" для доступа к созданному пакету.

**Подписывание пакета и публикация в Google Play**

Создайте ключ подписи в Android Studio:

Перейдите в "Build" > "Generate Signed Bundle / APK".

Выберите "APK" и нажмите "Next".

Создайте новый или используйте существующий ключ в "Key Store".

Заполните необходимые поля и подпишите APK.

Загрузите подписанный APK-файл в Google Play Console:

Войдите в Google Play Console и выберите ваш проект.

Перейдите в "Release management" > "App releases" и выберите тип выпуска (Production, Beta и т.д.).

Нажмите "Create release" и загрузите подписанный APK-файл.

Заполните все необходимые поля, включая описание, скриншоты и иконку.

Опубликуйте приложение, следуя инструкциям в Google Play Console.

**Билет №5**

**1. Основные языки для разработки мобильных приложений (Java, Kotlin, С#). Взаимодействие XAML и C#, Xml и Kotlin.**

**2. Создать мобильное приложение «Калькулятор» с основными арифметическими операциями (+, - , \*, / ) над целыми числами. Числа вводятся только в одном поле, вычисления производятся при нажатии на кнопку «=».**

**Основные языки:**

Java был одним из основных языков для разработки Android-приложений на протяжении многих лет. Он позволяет создавать производительные и надежные приложения.

Kotlin — это современный язык программирования, который был официально признан Google в качестве предпочтительного языка для разработки Android-приложений. Kotlin предлагает более краткий и безопасный синтаксис по сравнению с Java.

C# используется для разработки кроссплатформенных мобильных приложений с использованием фреймворка Xamarin. Xamarin позволяет разрабатывать приложения для Android, iOS и Windows с использованием единой кодовой базы.

**Взаимодействие XAML и C#:**

XAML (Extensible Application Markup Language): Это язык разметки, используемый для определения пользовательского интерфейса (UI) в приложениях Xamarin и WPF (Windows Presentation Foundation).

C#: C# используется для написания логики приложения, обработки событий, управления данными и взаимодействия с пользовательским интерфейсом, определенным в XAML.

Взаимодействие: XAML и C# работают вместе. XAML определяет структуру UI, а C# обрабатывает события и управляет поведением UI.

**Взаимодействие XML и Kotlin:**

XML (Extensible Markup Language): XML используется в Android для определения макетов пользовательского интерфейса, хранения данных и конфигурации.

Kotlin: Kotlin используется для написания логики приложения и взаимодействия с UI, определенным в XML.

Взаимодействие: Kotlin может использовать XML для определения макетов UI. Kotlin-код может получать доступ к элементам UI, определенным в XML, и управлять ими.

**Взаимодействие XAML и C#:**

**Разметка UI в XAML:**

XAML (Extensible Application Markup Language) описывает визуальную структуру пользовательского интерфейса (UI). Он определяет элементы управления (кнопки, текстовые поля, изображения и т.д.), их свойства (размер, положение, цвет) и их взаимосвязи (расположение внутри контейнеров, связь с другими элементами).

**Связывание XAML с C# (Code-Behind):**

Каждый XAML-файл обычно связан с соответствующим C# файлом, который называется “code-behind” файл (например, MainPage.xaml -> MainPage.xaml.cs).

x:Class в XAML указывает на имя класса C# в code-behind файле.

x:Name в XAML присваивает уникальное имя элементу UI, чтобы к нему можно было обратиться из C#.

Обработчики событий (Clicked="OnButtonClicked" в примере) указывают, какую функцию C# нужно вызвать при возникновении события (например, нажатие кнопки).

**Доступ к элементам UI из C#:**

В code-behind файле, вы можете получить доступ к элементам UI, определенным в XAML, используя их имена (например, myButton).

Вы можете изменять свойства этих элементов, вызывать методы, изменять их видимость и т.д.

**Обработка событий:**

Когда пользователь взаимодействует с UI (нажимает кнопку, вводит текст и т.д.), возникают события.

C# код, связанный с этими событиями (обработчики событий), выполняется для реагирования на эти взаимодействия (например, обновление UI, выполнение действий).

**Взаимодействие XML и Kotlin:**

**Разметка UI в XML:**

XML (Extensible Markup Language) используется для определения UI в Android приложениях. Он определяет элементы управления (кнопки, текстовые поля, изображения и т.д.), их свойства и структуру (расположение внутри контейнеров).

**Доступ к элементам UI из Kotlin:**

В Kotlin коде (например, в Activity или Fragment), вы используете findViewById() для получения ссылки на элементы UI, определенные в XML, по их id.

View Binding (рекомендуется с Android Studio 3.6 и выше): Более современный и безопасный способ. View Binding автоматически генерирует классы связывания, которые упрощают доступ к элементам UI, устраняя необходимость использования findViewById().

Data Binding: Более продвинутый способ, позволяющий связывать данные напрямую с элементами UI в XML, автоматически обновляя UI при изменении данных (и наоборот).

**Обработка событий:**

В Kotlin коде вы устанавливаете слушателей событий (например, setOnClickListener) для элементов UI.

Код внутри слушателя события выполняется при возникновении события (например, нажатие кнопки).

**Билет №6**

**1. Инструмент разработки мобильных приложений: Visual Studio, JDK, Android Studio, установка, компоненты, настройка, структура проекта, запуск приложения.**

**2. Создать мобильное приложение «Калькулятор» с основными арифметическими операциями (+, - , \*, / ) над действительными числами. Операция выбирается из списка. Числа вводятся в двух полях, вычисления производятся при нажатии на кнопку «=».**

**Visual Studio**

**Установка**

Перейдите на официальный сайт Visual Studio и скачайте установочный файл.

Запустите установщик и следуйте инструкциям мастера установки.

Выберите рабочую нагрузку "Mobile development with .NET" (Разработка мобильных приложений с .NET), которая включает Xamarin.

**Компоненты**

Xamarin: фреймворк для кроссплатформенной разработки мобильных приложений.

.NET Core: платформа для создания серверной части приложений.

NuGet: система управления пакетами для .NET.

UWP: разработка приложения для UWP

**Настройка**

Откройте Visual Studio и создайте новый проект.

Выберите шаблон для мобильного приложения (например, Xamarin.Forms App).

Установите необходимые пакеты с помощью NuGet Package Manager

Настройте проект, указав целевые платформы (Android, iOS) и версии SDK.

**Структура проекта**

MainActivity.cs: файл, содержащий главную активность приложения для Android.

App.xamlи App.xaml.cs: файлы, определяющие общие настройки и инициализацию приложения.

Resources: директория, содержащая ресурсы, такие как изображения, макеты, стили.

XAML файлы: файлы разметки интерфейса пользователя.

**Запуск приложения**

Подключите физическое устройство или настройте эмулятор Android.

Выберите целевое устройство в Visual Studio.

Нажмите кнопку "Run" (Запустить) или используйте сочетание клавиш Ctrl+F5 для запуска приложения.

**JDK (Java Development Kit)**

**Установка**

Перейдите на официальный сайт Oracle и скачайте последнюю версию JDK.

Установите JDK, следуя инструкциям мастера установки.

**Настройте переменные среды:**

Добавьте путь к директории установки JDK в переменную среды PATH.

Создайте переменную среды JAVA\_HOME, указывающую на директорию установки JDK.

**Компоненты**

javac: компилятор, преобразующий исходный код Java в байт-код

java: интерпретатор, выполняющий байт-код Java.

JRE: среда выполнения Java-приложений.

**Настройка**

Убедитесь, что JDK корректно установлен и доступен через командную строку.

Настройте переменные среды PATH и JAVA\_HOME.

**Структура проекта**

src: директория для хранения исходного кода Java.

bin: директория для хранения скомпилированных классов.

lib: директория для хранения внешних библиотек.

**Запуск приложения**

Напишите исходный код Java и сохраните его в файле с расширением .java.

Скомпилируйте код с помощью команды javac MyApp.java.

Запустите приложение с помощью команды java MyApp.

**Android Studio**

**Установка**

Перейдите на официальный сайт Android Studio и скачайте установочный файл.

Запустите установщик и следуйте инструкциям мастера установки.

Установите необходимые компоненты, такие как Android SDK, эмуляторы и другие инструменты разработки.

**Компоненты**

Android SDK: набор инструментов и библиотек для разработки Android-приложений.

Emulator: виртуальные устройства для тестирования приложений.

Gradle: система сборки для автоматизации процесса сборки и управления зависимостями.

**Настройка**

Создайте новый проект в Android Studio.

Выберите шаблон проекта (например, Empty Activity) и настройте параметры, такие как минимальная и целевая версия SDK.

Настройте файлы сборки Gradle для добавления зависимостей и плагинов.

**Структура проекта**

app: основной модуль приложения, содержащий весь код и ресурсы.

src: исходный код приложения.

main: основной код и ресурсы.

java: директория для файлов Java или Kotlin.

res: директория для ресурсов (макеты, изображения, строки).

build.gradle: файлы конфигурации сборки для проекта и модулей.

**Запуск приложения**

Подключите физическое устройство или настройте эмулятор Android.

Выберите целевое устройство в Android Studio.

Нажмите кнопку "Run" (Запустить) или используйте сочетание клавиш Shift+F10 для запуска приложения.

**Билет №7**

**1. Асинхронное программирование, сопрограммы (Coroutines) виды и назначение.**

**2. Создать мобильное приложение для заказа пиццы. Выбор наименования пиццы через список, количество задается через Slider (SeekBar) и отображается на экране. Информация о заказе выводится в диалоговом окне при нажатии на кнопку «Отправить заказ».**

Асинхронное программирование позволяет выполнять несколько операций одновременно, не блокируя основной поток выполнения программы. Это особенно полезно для ввода-вывода, сетевых операций и работы с базами данных, где время ожидания может быть значительным.

Сопрограммы (корутины) — это блоки кода, которые работают по очереди. В нужный момент исполнение такого блока приостанавливается с сохранением всех его свойств, чтобы запустился другой код. Когда управление возвращается к первому блоку, он продолжает работу. 3

**Виды сопрограмм:**

Корутины на уровне языка. Например, в C# для объявления и использования корутин используются ключевые слова async и await, в Kotlin — ключевое слово suspend.

Корутины на уровне библиотеки. Например, C# имеет класс System.Threading.Tasks.Task для представления асинхронных операций, которые могут быть запущены как корутины, Java — библиотеку java.util.concurrent.CompletableFuture для создания и комбинирования асинхронных задач.

**Назначение сопрограмм:**

создание асинхронных приложений, которые могут выполнять несколько действий одновременно;

гибкая и удобная реализация многозадачности;

больший контроль при переключении между разными задачами (корутинами управляют разработчик и программа, а не операционная система);

снижение нагрузки на аппаратные ресурсы устройства.

**C#**

async и await: Ключевые слова async и await позволяют создавать и вызывать асинхронные методы.

async: Указывает, что метод является асинхронным и может содержать оператор await.

await: Приостанавливает выполнение метода до завершения асинхронной операции. Операция await должна применяться к Task или Task<T>.

Task и Task<T>: Представляют асинхронную операцию. Task используется для операций, не возвращающих значение, а Task<T> - для операций, возвращающих значение типа T.

**Kotlin**

suspend: Ключевое слово suspend указывает, что функция может быть приостановлена и возобновлена. Только другие suspend функции или сопрограммы могут вызывать suspend функции.

CoroutineScope: Определяет жизненный цикл сопрограмм. Сопрограммы должны запускаться в рамках CoroutineScope.

CoroutineDispatcher: Определяет, в каком потоке будет выполняться сопрограмма. Например, Dispatchers.Main - для основного потока, Dispatchers.IO - для операций ввода/вывода, Dispatchers.Default - для вычислительно-интенсивных задач.

launch и async (Coroutine builders):

launch: Запускает сопрограмму и не возвращает результат.

async: Запускает сопрограмму и возвращает Deferred<T>, который представляет результат.

withContext: Позволяет переключать контекст выполнения сопрограммы.

lifecycle(-Scope): позволяет создавать и управлять сопрограммами, привязанными к жизненному циклу конкретного компонента

**Билет №8**

**1. Контейнеры в Xamarin Forms и Android Studio. Описание, назначение, примеры, принцип использования в коде проекта.**

**2. Создать в мобильном приложении стандартный модуль для получения изображения с камеры.**

**Xamarin Forms**

StackLayout: Располагает элементы друг под другом или друг возле друга.

AbsoluteLayout: Позволяет указать абсолютное позиционирование элементов.

RelativeLayout: Позволяет размещать элементы относительно друг друга.

Grid: Система сеток, где элементы добавляются в ячейки.

FlexLayout: Располагает элементы друг возле друга, перекидывая их на новый ряд, если они не вмещаются.

ScrollView: Если контент выходит за пределы экрана, его можно пролистать.

**Пример использования StackLayout в коде C#:**

public class MainPage : ContentPage

{

public MainPage()

{

Label label1 = new Label { Text = "Первая метка" };

Label label2 = new Label { Text = "Вторая метка" };

StackLayout stackLayout = new StackLayout { Children = { label1, label2 } };

this.Content = stackLayout;

}

}

**Android Studio**

ConstraintLayout: Это мощный макет, который позволяет вам позиционировать и размерять виджеты в зависимости от других виджетов и предков, используя ограничения. Он очень гибкий и особенно полезен для сложных макетов.

LinearLayout: Этот макет размещает дочерние элементы по горизонтали или вертикали.

Horizontal: Все элементы располагаются в строку слева направо.

Vertical: Все элементы располагаются в столбец сверху вниз.

FrameLayout: Простой макет, предназначенный для размещения одного виджета или нескольких виджетов поверх друг друга. Он часто используется для размещения одного элемента в центре экрана или для наложения элементов друг на друга.

TableLayout: Этот макет организует элементы в виде таблицы с рядами и колонками. Каждый элемент помещается в ячейку таблицы. Он полезен для создания макетов с фиксированным количеством строк и столбцов.**Пример использования ConstraintLayout в XML:**

<androidx.constraintlayout.widget.ConstraintLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:id="@+id/textView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Пример текста"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

**Билет №9**

**1. Структура типичного мобильного приложения: активности и интенты, архитектура MVC.**

**2. Выполнить тестирование и оптимизацию предложенного мобильного приложения «Калькулятор», расположенного в каталоге «Экзамен».**

**Структура типичного мобильного приложения состоит из нескольких ключевых компонентов, включая активности и интенты, а также архитектурные паттерны, такие как MVC (Model-View-Controller).**

**Активности и Интенты:**

**Активности (Activities):**

Активность представляет собой один экран пользовательского интерфейса в приложении. Каждая активность отвечает за взаимодействие с пользователем и может содержать различные элементы интерфейса, такие как кнопки, текстовые поля и другие виджеты.

Приложение может содержать несколько активностей, и переход между ними осуществляется с помощью интентов.

Жизненный цикл активности включает такие стадии, как создание (onCreate), запуск (onStart), возобновление (onResume), пауза (onPause), остановка (onStop), уничтожение (onDestroy).

**Интенты (Intents):**

Интенты — это механизмы, с помощью которых активности взаимодействуют друг с другом. Они позволяют запускать новые активности, передавать данные между ними и даже взаимодействовать с другими приложениями.

Существует два типа интентов: явные (explicit intents) и неявные (implicit intents). Явные интенты указывают конкретную активность, которая должна быть запущена, а неявные интенты позволяют запускать активности, соответствующие заданным критериям (например, открытие URL в браузере).

**Архитектура MVC:**

Архитектурный паттерн MVC (Model-View-Controller) разделяет приложение на три основные компонента:

**Model (Модель):**

Содержит бизнес-логику и данные приложения. Модель управляет состоянием данных и отвечает за их сохранение и получение.

В Android это может быть база данных, репозиторий данных, классы данных и другие компоненты, которые обрабатывают данные.

**View (Представление):**

Отвечает за отображение данных и взаимодействие с пользователем. Представление получает данные из модели и отображает их в соответствующих элементах интерфейса.

В Android это могут быть XML-ресурсы макетов, фрагменты, активности и другие компоненты, которые формируют пользовательский интерфейс.

**Controller (Контроллер):**

Управляет взаимодействием между моделью и представлением. Контроллер получает ввод от пользователя через представление, обрабатывает его и обновляет модель. Затем модель уведомляет представление об изменениях данных.

В Android контроллер может быть представлен активностями, фрагментами или ViewModel-классами, которые связывают модель и представление.

**Билет №10**

**1. Структура типичного мобильного приложения: активности и интенты, архитектура MVVM.**

**2. Написать в мобильном приложении получение массива данных с API используя паттерн MVVM. Ссылка на API расположена в каталоге «Экзамен».**

**Активности (Activities)**

Активность представляет собой один экран пользовательского интерфейса в Android-приложении. Это основной компонент, который взаимодействует с пользователем.

Каждая активность имеет жизненный цикл, который управляется системой Android (onCreate, onStart, onResume, onPause, onStop, onDestroy).

Приложение может содержать несколько активностей, между которыми можно переходить с помощью интентов.

**Интенты (Intents)**

Интенты используются для взаимодействия между компонентами приложения. Они могут запускать активности, службы и передавать данные.

Явные интенты (Explicit Intents): указывают конкретную активность, которую нужно запустить.

Неявные интенты (Implicit Intents): указывают действие и данные, и система Android выбирает подходящий компонент для выполнения этого действия.

**Архитектура MVVM (Model-View-ViewModel)**

Архитектура MVVM разделяет код на три основные компонента: Модель (Model), Представление (View) и Модель представления (ViewModel). Это позволяет упростить управление данными и взаимодействие между слоями, что делает приложение более устойчивым и легко масштабируемым.

Model (Модель)

Модель отвечает за данные и бизнес-логику приложения. Она управляет состоянием данных и выполняет операции чтения и записи.

View (Представление)

Представление отвечает за отображение данных и взаимодействие с пользователем. Оно получает данные из модели представления и отображает их в соответствующих элементах интерфейса.

ViewModel (Модель представления)

Модель представления служит посредником между моделью и представлением. Она предоставляет данные для представления и обрабатывает пользовательские действия.

ViewModel не знает о существовании представления, что позволяет легко тестировать и повторно использовать код.

**Взаимодействие компонентов в MVVM**

Модель (Model) содержит данные и бизнес-логику.

Модель представления (ViewModel) взаимодействует с моделью и предоставляет данные для представления.

Представление (View) получает данные из модели представления и отображает их. Любые изменения, внесенные пользователем, передаются обратно в модель представления.

**Ключевые отличия**

**Обработка привязки данных:**

В MVC контроллер вручную обновляет представление в ответ на изменения модели.

В MVVM привязка данных автоматически обновляет представление при изменениях модели.

**Прямое взаимодействие с представлением:**

В MVC контроллер напрямую взаимодействует с представлением.

В MVVM ViewModel не знает о конкретном представлении; вместо этого используется механизм привязки данных.

**Сложность:**

MVC более прост в реализации и подходит для небольших проектов.

MVVM сложнее, но более эффективен для крупных приложений с богатым интерфейсом и большим количеством данных.

**Тестирование:**

В MVVM ViewModel легко тестировать, так как он не зависит от представления.

В MVC тестирование может быть сложнее из-за зависимости контроллера от представления.

**Билет №11**

**1. Способы хранения данных: работа с базой данных SQLite и файловой системой.**

**2. Реализовать интерфейс мобильного приложения по образцу в Приложении №1. При нажатии на кнопку «Товары» должен быть переход на страницу со списком товаром согласно данным ресурса к билету.**

**SQLite** — это легковесная реляционная база данных, которая встроена в Android. Она часто используется для локального хранения данных в мобильных приложениях.

**Xamarin**

В Xamarin можно работать с SQLite, используя пакет NuGet sqlite-net-pcl. Вот пример, как это можно сделать:

Установите пакет sqlite-net-pcl через NuGet Package Manager.

Создайте модель данных:

public class Item

{

[PrimaryKey, AutoIncrement]

public int Id { get; set; }

public string Name { get; set; }

}

Создайте класс для управления базой данных:

public class Database

{

private readonly SQLiteAsyncConnection \_database;

public Database(string dbPath)

{

\_database = new SQLiteAsyncConnection(dbPath);

\_database.CreateTableAsync<Item>().Wait();

}

public Task<List<Item>> GetItemsAsync()

{

return \_database.Table<Item>().ToListAsync();

}

public Task<int> SaveItemAsync(Item item)

{

return \_database.InsertAsync(item);

}

public Task<int> DeleteItemAsync(Item item)

{

return \_database.DeleteAsync(item);

}

}

В **Kotlin** для работы с SQLite можно использовать встроенные классы SQLiteOpenHelper и SQLiteDatabase. Вот пример:

Создайте класс DatabaseHelper, унаследованный от SQLiteOpenHelper:

class DatabaseHelper(context: Context) : SQLiteOpenHelper(context, DATABASE\_NAME, null, DATABASE\_VERSION) {

companion object {

private const val DATABASE\_NAME = "example.db"

private const val DATABASE\_VERSION = 1

private const val TABLE\_NAME = "items"

private const val COLUMN\_ID = "id"

private const val COLUMN\_NAME = "name"

}

override fun onCreate(db: SQLiteDatabase) {

val createTable = "CREATE TABLE $TABLE\_NAME ($COLUMN\_ID INTEGER PRIMARY KEY, $COLUMN\_NAME TEXT)"

db.execSQL(createTable)

}

override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {

db.execSQL("DROP TABLE IF EXISTS $TABLE\_NAME")

onCreate(db)

}

}

Используйте SQLiteDatabase для работы с данными:

val dbHelper = DatabaseHelper(context)

val db = dbHelper.writableDatabase

val values = ContentValues().apply {

put("name", "Item Name")

}

db.insert("items", null, values)

val cursor = db.query("items", arrayOf("id", "name"), null, null, null, null, null)

while (cursor.moveToNext()) {

val itemName = cursor.getString(cursor.getColumnIndexOrThrow("name"))

}

cursor.close()

**Файловая система**

Работа с файловой системой позволяет хранить файлы и данные в виде файлов на устройстве.

**Xamarin**

В Xamarin работа с файловой системой может включать такие действия, как сохранение текстовых данных, изображений или других файлов. Вот дополнительные примеры для работы с файлами и каталогами:

**Создание и запись в файл:**

var folderPath = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "MyAppData");

Directory.CreateDirectory(folderPath);

var filePath = Path.Combine(folderPath, "example.txt");

File.WriteAllText(filePath, "Hello, Xamarin!");

**Чтение данных из файла:**

var text = File.ReadAllText(filePath);

Console.WriteLine(text);

**Получение списка файлов в каталоге:**

var files = Directory.GetFiles(folderPath);

foreach (var file in files)

{

Console.WriteLine(file);

}

В **Kotlin** можно также работать с файлами и каталогами. Вот дополнительные примеры для выполнения различных операций:

**Создание и запись в файл:**

val folderPath = File(context.filesDir, "MyAppData")

if (!folderPath.exists()) {

folderPath.mkdirs()

}

val filePath = File(folderPath, "example.txt")

filePath.writeText("Hello, Kotlin!")

**Чтение данных из файла:**

val text = filePath.readText()

println(text)

**Получение списка файлов в каталоге:**

val files = folderPath.listFiles()

files?.forEach { file -> println(file.name)

**Билет №12**

**1. Элемент ListView: виды, назначение, способы применения, пользовательские шаблоны.**

**2. Создать мобильное приложение для оценки качества (от 1 до 5) оказания услуг в автосервисе. Выбор наименования услуги (замена масла, ремонт подвески, замена АКПП, диагностика) должен быть удобным для пользователя. При нажатии на кнопку «Отправить оценку» пользователь должен быть оповещен об успешной отправке и выводом значения оценки.**

ListView - это распространенный элемент интерфейса пользователя (UI), который отображает вертикально прокручиваемый список элементов. Он используется для представления коллекций данных в структурированном и удобном для пользователя виде. В этой статье мы рассмотрим ListView с точки зрения Kotlin (Android) и Xamarin (Android и iOS), включая его виды, назначение, применение и создание пользовательских шаблонов.

**Назначение ListView:**

Отображение коллекций данных: Основное назначение - представить списки данных, такие как имена, сообщения, изображения, продукты и т.д.

Прокрутка: Обеспечивает прокрутку для отображения больших объемов данных, которые не помещаются на экране.

Выбор элементов: Позволяет пользователю выбирать один или несколько элементов из списка.

Взаимодействие: Может быть использован для выполнения действий при выборе элемента, например, открытия новой страницы или отображения деталей.

Эффективность: ListView в Android и его эквиваленты в Xamarin оптимизированы для эффективной работы с большими списками данных, используя механизмы повторного использования View (view recycling), чтобы минимизировать создание новых объектов и улучшить производительность.

**Виды ListView:**

В Android (Kotlin) и Xamarin существует несколько способов реализации ListView, отличающихся по функциональности и сложности реализации:

**Android (Kotlin)**

**ListView (стандартный):**

Назначение: Базовый класс для отображения списков.

Особенности: Требует ручной реализации адаптера для связывания данных и отображения элементов. Менее эффективен для очень больших списков из-за отсутствия встроенной поддержки RecyclerView.

Пример: Для простых списков с небольшим объемом данных.

**RecyclerView:**

Назначение: Более продвинутый и гибкий контейнер для отображения списков и сеток.

Особенности:

Использует ViewHolder для повторного использования View, что значительно повышает производительность для больших списков.

Поддерживает различные LayoutManager’ы (LinearLayoutManager, GridLayoutManager, StaggeredGridLayoutManager) для создания разных видов макетов.

Более сложен в настройке, чем ListView, но предоставляет больше возможностей.

Пример: Рекомендуется для большинства случаев, особенно для больших и сложных списков.

**ExpandableListView:**

Назначение: Представляет данные в виде иерархической структуры (группы и дочерние элементы).

Особенности: Специализированный класс для отображения раскрывающихся списков.

Пример: Для отображения категорий и подкатегорий, меню с подменю и т.д.

**Xamarin (Android & iOS):**

**ListView (Xamarin.Forms):**

Назначение: Кросс-платформенный ListView для Android и iOS.

Особенности: Использует data binding для связывания данных с UI. Поддерживает шаблоны данных (DataTemplate) для определения внешнего вида элементов.

Пример: Подходит для большинства случаев, когда нужна кросс-платформенность и не требуется максимальная оптимизация.

**RecyclerView (Xamarin.Android):**

Назначение: Xamarin-обертка для нативного Android RecyclerView.

Особенности: Предоставляет доступ к преимуществам RecyclerView в Xamarin.Android.

Пример: Для случаев, когда требуется максимальная производительность на Android и вы готовы писать платформо-специфичный код.

**UITableView (Xamarin.iOS):**

Назначение: Нативный ListView для iOS

Особенности: Для iOS разработки.

Пример: Для случаев, когда требуется максимальная производительность на iOS и вы готовы писать платформо-специфичный код.

**ИЛИ**

**Xamarin**

TextCell: Отображает одну строку текста.

ImageCell: Включает изображение и текст.

SwitchCell: Включает переключатель (toggle switch).

EntryCell: Включает текстовое поле для ввода данных.

CustomCell: Создание собственных ячеек с уникальным оформлением.

**Kotlin**

В Kotlin, для платформы Android, ListView использует адаптеры для отображения данных:

ArrayAdapter: Простой адаптер для массива данных.

SimpleAdapter: Более сложный адаптер, позволяющий отображать список с несколькими типами данных.

Custom Adapter: Позволяет создавать собственные адаптеры для сложных данных и интерфейсов.

**Некоторые способы применения ListView:**

Прокручиваемый список элементов. ListView популярен на мобильных устройствах благодаря удобству использования. Работа со списком состоит из двух частей: сначала добавляется на форму сам ListView, а затем он заполняется элементами списка.

Создание необычной компоновки. Например, чтобы построить таблицу, размещающую более одного элемента в одной строке, или же вообще не использующую обычную табличную компоновку.

Создание иерархического меню. Первый элемент ListView используется для создания элементов H1, а вложенный ListView — для последующей визуализации списка ссылок.

Множественный выбор. Компонент ListView позволяет сделать не только единичный выбор, но и множественный. В этом случае необходимо использовать соответствующую разметку и установить свойство ChoiceMode в описании интерфейса компонента равным multiplyChoice.

**Билет №13**

**1. Глобализация и локализация при разработке мобильных приложений.**

**2. Создать интерфейс окна «Авторизация» мобильного приложения (Приложение №2) с возможностью локализации на двух языках: русский и английский.**

**Глобализация:**

Единый код для разных рынков: Глобализация подразумевает создание единого кода, который можно адаптировать под разные языки и культуры без значительных изменений.

Поддержка разных валют и форматов данных: Приложение должно правильно обрабатывать различные валюты, форматы даты и времени, системы измерений и другие региональные особенности.

Многоязычный интерфейс: Интерфейс приложения должен поддерживать множественные языки и автоматически переключаться в зависимости от предпочтений пользователя или региона его нахождения.

**Локализация:**

Перевод текста: Перевод интерфейса приложения на разные языки – это основная задача локализации. Важно обеспечить качественный перевод с учетом культурных особенностей.

Адаптация контента: Необходимо учитывать культурные различия при создании контента. Например, изображения и символы, которые могут быть понятны и уместны в одном регионе, могут быть неприемлемы в другом.

Поддержка локальных нормативов: Приложение должно соответствовать законодательству и нормативам каждой страны, в которой оно используется. Это может включать в себя защиту данных, возрастные ограничения и так далее.

**ИЛИ**

Глобализация при разработке мобильных приложений — это стратегия по подготовке приложения к интернациональному использованию. Она заключается в использовании форматов данных, применяемых в том или ином регионе.

Локализация — это адаптация продукта и его документации к языковым соответствиям, культурным и другим требованиям целевого рынка или рынков (локалей). Она включает в себя перевод текстовых элементов пользовательского интерфейса, подписей к картинкам, сообщений пользователям, элементов управления на все языки, которые поддерживаются в программном продукте.

**Билет №14**

**1. Навигация между страницами в Xamarin Forms и Android Studio: стек навигации, методы, примеры.**

**2. Создать мобильное приложение «Планировщик задач» с возможностью создать до 3 задач на одном экране: название задачи, статус (выполнено/не выполнено), дата задачи (выбор через DatePicker). Добавление новой задачи на экране происходит при нажатии отдельной кнопки.**

**Навигация в Xamarin Forms**

Навигация в Xamarin Forms основывается на двух ключевых компонентах: NavigationPage и INavigation.

NavigationPage: Это специальный тип страницы, которая управляет стеком других страниц. Она предоставляет встроенную панель навигации, которая позволяет пользователям перемещаться между страницами.

INavigation: Интерфейс, который предоставляет методы для управления навигацией. Он позволяет добавлять и удалять страницы из стека навигации.

**Стек навигации в Xamarin Forms состоит из двух основных свойств:**

NavigationStack: Хранит все немодальные страницы (те, которые отображаются в основном приложении).

ModalStack: Хранит все модальные страницы (те, которые отображаются поверх основной страницы и требуют взаимодействия пользователя перед продолжением работы с приложением).

**Методы навигации включают:**

PushAsync(): Добавляет новую страницу в стек навигации.

PopAsync(): Удаляет текущую страницу из стека навигации.

InsertPageBefore(): Вставляет страницу в стек перед другой страницей.

RemovePage(): Удаляет определенную страницу из стека.

PopToRootAsync(): Возвращает пользователя на корневую страницу.

Эти методы позволяют создавать последовательные, логичные пути навигации для пользователей.

**Управление стеком:**

InsertPageBefore(): Вставляет страницу перед другой страницей в NavigationStack.

RemovePage(): Удаляет конкретную страницу из NavigationStack.

**Навигация в Android Studio**

Навигация в Android часто осуществляется с помощью Intents и Activities:

Intent: Это объект, который представляет собой намерение выполнить какое-либо действие, например, открыть новую активность или отправить данные.

Activity: Это основная единица приложения, представляющая один экран с пользовательским интерфейсом.

**Процесс навигации обычно включает в себя следующие шаги:**

Создание новой активности

Настройка кнопки или другого элемента интерфейса для запуска нового Intent.

Обработка намерения для запуска новой активности.

Существует также возможность использовать Fragments и Navigation Components для более сложных сценариев навигации. Fragments позволяют разбить интерфейс на меньшие части, которые можно динамически менять, а Navigation Components предоставляют архитектурный подход к управлению навигацией внутри приложения.

Стек навигации в **Kotlin** используется для отслеживания переходов по различным экранам в приложении. При первом запуске приложения начальный экран помещается в стек. Когда пользователь переходит к другому экрану, то этот экран помещается на верхушку стека. Когда пользователь перемещается обратно по экранам с помощью системной кнопки «Назад», каждый компонент извлекается из стека до тех пор, пока главный экран не окажется единственным элементом в стеке.

**Билет №15**

**1. Тестирование мобильных приложений: ручное и автоматизированное тестирование, тестирование интерфейса.**

**2. Создать интерфейс мобильного приложения с использованием элемента ListView (Приложение №3). При выборе элемента должна открываться страница с указанным веб-адресом.**

**Ручное тестирование**

Ручное тестирование включает в себя выполнение тестов вручную без использования автоматизированных инструментов. Тестировщики проходят через различные сценарии использования приложения и проверяют его функциональность, производительность и удобство использования.

**Преимущества:**

Позволяет выявлять ошибки, которые трудно обнаружить автоматически.

Оценивает удобство использования и интерфейс с точки зрения пользователя.

Гибкость в выполнении тестов.

**Недостатки:**

Может быть трудоемким и дорогим

Человеческий фактор может привести к пропуску некоторых ошибок.

**Автоматизированное тестирование**

Автоматизированное тестирование использует скрипты и инструменты для автоматического выполнения тестов. Этот метод позволяет более эффективно и быстро проверять приложение на наличие ошибок и регрессий.

**Примеры инструментов:**

Appium: Кроссплатформенный инструмент для автоматизации мобильных приложений.

Espresso: Инструмент для автоматизации UI-тестов, разработанный Google для Android.

XCUITest: Инструмент для автоматизации UI-тестов, разработанный Apple для iOS.

**Преимущества:**

Повышенная точность и консистентность тестирования.

Возможность быстрого выполнения большого количества тестов.

Легкость в интеграции в CI/CD процессы.

**Недостатки:**

Требует начальных затрат на написание тестов.

Ограниченная гибкость для креативных и нетривиальных тестов.

Тестирование интерфейса

**Тестирование интерфейса** (UI-тестирование) включает в себя проверку визуального и функционального поведения пользовательского интерфейса приложения. Это важно для обеспечения удобства использования и правильной работы всех элементов интерфейса.

**Примеры подходов:**

Тесты на соответствие макету: Проверка соответствия элементов интерфейса утвержденным дизайнам.

Тесты на функциональность UI: Проверка правильного поведения элементов интерфейса при взаимодействии с пользователем.

Тесты на адаптивность: Проверка корректного отображения интерфейса на различных устройствах и экранах.

**Инструменты для UI-тестирования:**

Selenium: Используется для тестирования веб-приложений, но можно адаптировать для мобильных приложений.

Appium: Поддерживает тестирование интерфейса для мобильных приложений.

Espresso и XCUITest: Поддерживают автоматизацию UI-тестов для Android и iOS соответственно.

**Билет №16**

**1. Привязка данных (data binding), создание адаптера: способы задания, виды, примеры.**

**2. Создать мобильное приложение для решения квадратного уравнения ax2+bx+c=0. Выполнить тестирование функций приложения и зафиксировать результаты с пояснением в текстовом документе.**

**Привязка данных (Data Binding)** - это техника, которая позволяет установить связь между данными в вашем приложении и элементами пользовательского интерфейса (UI). Вместо того, чтобы вручную обновлять UI при изменении данных, система привязки данных автоматически отражает эти изменения.

**Адаптер (Adapter)** - это компонент, который служит посредником между источником данных (например, списком объектов) и View, отображающим эти данные (например, RecyclerView, ListView). Адаптер берет данные из источника и преобразует их в формат, понятный для View.

**Виды адаптеров:**

ArrayAdapter: Простой адаптер для отображения массива данных в ListView или Spinner.

BaseAdapter: базовый класс для создания своих адаптеров, позволяя разработчикам переопределять методы для более сложных операций с данными и макетами.

RecyclerView.Adapter: Более гибкий адаптер для отображения данных в RecyclerView. Поддерживает различные типы макетов и улучшенную производительность.

CursorAdapter: Адаптер для отображения данных из Cursor (результата запроса к базе данных).

**Привязка данных (Data Binding)**

Привязка данных позволяет автоматически синхронизировать данные между источником и элементами пользовательского интерфейса. В Xamarin это делается с помощью XAML, а в Kotlin - с использованием Data Binding API.

**Способы задания**

XAML (Xamarin):

<TextBlock Text="{Binding Name}" />

Здесь Binding указывает на свойство Name объекта, связанного с данными.

**Kotlin (Android):**

binding.textView.text = viewModel.name

Здесь binding - это объект Data Binding, а viewModel.name - это свойство, к которому привязан текстовый элемент.

Виды привязки

OneWay: Изменения в источнике данных отражаются в элементе пользовательского интерфейса.

TwoWay: Изменения в источнике данных и элементе пользовательского интерфейса взаимно синхронизируются.

OneTime: Значение привязки берется из источника данных один раз при инициализации.

OneWayToSource: Изменения в элементе пользовательского интерфейса отражаются в источнике данных.

**Примеры**

**Xamarin (C#):**

public class Person

{

public string Name { get; set; }

}

public MainPage()

{

InitializeComponent();

BindingContext = new Person { Name = "John Doe" };

}

В этом примере BindingContext устанавливается для всего окна, и Name отображается в элементах пользовательского интерфейса.

**Kotlin (Android):**

class MainActivity : AppCompatActivity() {

private lateinit var binding: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

binding = DataBindingUtil.setContentView(this, R.layout.activity\_main)

binding.lifecycleOwner = this

binding.viewModel = ViewModelProvider(this).get(ExampleViewModel::class.java)

}

}

В этом примере используется Data Binding API для привязки viewModel к Activity.

**Создание адаптера в Xamarin**

Адаптер в Xamarin используется для преобразования данных из источника в формат, который можно отобразить в элементах пользовательского интерфейса.

**Пример адаптера**

public class MyAdapter : BaseAdapter<MyData>

{

private List<MyData> data;

public MyAdapter(List<MyData> data)

{

this.data = data;

}

public override MyData this[int position]

{

get { return data[position]; }

}

public override int Count

{

get { return data.Count; }

}

public override View GetView(int position, View convertView, ViewGroup parent)

{

var view = convertView ?? LayoutInflater.From(parent.Context).Inflate(Resource.Layout.MyDataItem, parent, false);

var textName = view.FindViewById<TextView>(Resource.Id.textName);

textName.Text = data[position].Name;

return view;

}

}

В **Kotlin**, создание адаптера часто связано с использованием RecyclerView, который позволяет отображать длинные списки данных в прокручиваемом виде.

class MyAdapter(private val data: List<MyData>) : RecyclerView.Adapter<MyAdapter.ViewHolder>() {

class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {

val textName: TextView = itemView.findViewById(R.id.textName)

}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {

val view = LayoutInflater.from(parent.context).inflate(R.layout.my\_data\_item, parent, false)

return ViewHolder(view)

}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {

holder.textName.text = data[position].name

}

override fun getItemCount(): Int {

return data.size

}

}

**Билет №17**

**1. Объект Json, сериализация и десериализация объектов.**

**2. Создать интерфейс мобильного приложения «Покупки» (Приложение №4).**

**JSON** (JavaScript Object Notation) - это текстовый формат обмена данными, который легко читается человеком и легко парсится и генерируется машинами. Он используется для передачи данных между клиентом и сервером в веб-приложениях.

**Сериализация** - это процесс преобразования объекта в формат, который можно сохранить или передать (например, JSON).

**Десериализация** - это обратный процесс, преобразующий форматированное представление данных обратно в объект.

**Примеры:**

**Xamarin**

using Newtonsoft.Json;

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

public bool IsStudent { get; set; }

public List<string> Courses { get; set; }

public Address Address { get; set; }

}

public class Address

{

public string Street { get; set; }

public string City { get; set; }

}

var person = new Person

{

Name = "John Doe",

Age = 30,

IsStudent = false,

Courses = new List<string> { "Math", "Physics" },

Address = new Address

{

Street = "123 Main St",

City = "Anytown"

}

};

string json = JsonConvert.SerializeObject(person);

**Kotlin**

import kotlinx.serialization.\*

import kotlinx.serialization.json.\*

@Serializable

data class Person(

val name: String,

val age: Int,

val isStudent: Boolean,

val courses: List<String>,

val address: Address

)

@Serializable

data class Address(

val street: String,

val city: String

)

val person = Person(

name = "John Doe",

age = 30,

isStudent = false,

courses = listOf("Math", "Physics"),

address = Address(

street = "123 Main St",

city = "Anytown"

)

)

val json = Json.encodeToString(person)

**Билет №18**

**1. Подключение к API, способы в Xamarin Forms и Android Studio, библиотека Retrofit**

**2. Создать мобильное приложение с динамическим списком согласно примеру (Приложение №5). При нажатии кнопки «+» или «-» должно происходить увеличение или уменьшение количества выбранного товара соответственно.**

Подключение к API - это процесс взаимодействия вашего мобильного приложения с удаленным сервером для обмена данными. API (Application Programming Interface) определяет набор правил и спецификаций, которым ваше приложение должно следовать для отправки запросов и получения ответов от сервера.

Для подключения к API в **Xamarin Forms** можно использовать класс HttpClient. Вот пример, как это сделать:

Добавьте в проект NuGet пакет System.Net.Http (если он ещё не добавлен).

**Создайте метод для выполнения HTTP-запроса:**

using System.Net.Http;

using System.Threading.Tasks;

using Newtonsoft.Json;

public class ApiService

{

private static readonly HttpClient client = new HttpClient();

public async Task<T> GetDataAsync<T>(string url)

{

var response = await client.GetAsync(url);

response.EnsureSuccessStatusCode();

var content = await response.Content.ReadAsStringAsync();

return JsonConvert.DeserializeObject<T>(content);

}

**Используйте ApiService для получения данных:**

public class MainPageViewModel

{

private readonly ApiService \_apiService;

public MainPageViewModel()

{

\_apiService = new ApiService();

LoadData();

}

private async void LoadData()

{

var data = await \_apiService.GetDataAsync<List<MyData>>("https://api.example.com/data");

}

}

Для подключения к API в **Kotlin** с использованием библиотеки Retrofit выполните следующие шаги:

**Добавьте зависимости Retrofit в файл build.gradle вашего проекта:**

implementation 'com.squareup.retrofit2:retrofit:2.9.0'

implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

**Создайте модель данных:**

data class MyData(

val id: Int,

val name: String

)

**Создайте интерфейс API:**

import retrofit2.Call

import retrofit2.http.GET

interface ApiService {

@GET("data")

fun getData(): Call<List<MyData>>

}

**Настройте Retrofit:**

import retrofit2.Retrofit

import retrofit2.converter.gson.GsonConverterFactory

val retrofit = Retrofit.Builder()

.baseUrl("https://api.example.com/")

.addConverterFactory(GsonConverterFactory.create())

.build()

val apiService = retrofit.create(ApiService::class.java)

**Выполните запрос и обработайте данные:**

apiService.getData().enqueue(object : retrofit2.Callback<List<MyData>> {

override fun onResponse(call: Call<List<MyData>>, response: Response<List<MyData>>) {

if (response.isSuccessful) {

val data = response.body()

**Retrofit** — это библиотека для языка программирования Java (или Kotlin), которая позволяет удобно выполнять сетевые запросы к удалённым серверам в Android-приложениях.

Она предоставляет высокоуровневый интерфейс для взаимодействия с веб-сервисами, такими как RESTful API.

**Некоторые преимущества Retrofit:**

Удобство использования. Библиотека предоставляет аннотации для определения структуры запросов и ответов, что делает код более читаемым и понятным.

Автоматическая сериализация и десериализация данных. Retrofit может автоматически преобразовывать JSON-ответы от сервера в объекты Java или Kotlin и наоборот.

Многопоточная поддержка. Библиотека интегрируется с библиотекой OkHttp, что позволяет выполнять сетевые запросы асинхронно и управлять потоками.

Поддержка различных HTTP-методов. Можно использовать различные HTTP-методы, такие как GET, POST, PUT, DELETE и другие.

Интерактивность. Можно легко настраивать параметры запросов, заголовки и интерцепторы для работы с сетевыми запросами в соответствии с потребностями.

Расширяемость. Retrofit позволяет создавать собственные конвертеры для обработки данных, а также интерцепторы для выполнения дополнительных действий перед отправкой или после получения запросов.

**Билет №19**

**1. Жизненный цикл мобильного приложения: основные состояния, переход между состояниями.**

**2. Разработать интерфейс мобильного приложения «Заставка». При запуске приложения открывается окно с логотипом и названием приложения, по истечении 3 секунд автоматически осуществляется переход в окно приветствия (Приложение №6).**

**Жизненный цикл** мобильного приложения включает несколько основных состояний и переходов между ними. В Android это:

OnCreate() (Создано): Приложение только что создано. Здесь вы инициализируете основные компоненты.

OnStart() (Запущено): Приложение становится видимым для пользователя.

OnResume() (Возобновлено): Приложение становится активным и готово к взаимодействию с пользователем.

OnPause() (Приостановлено): Приложение более не активно, но по-прежнему видимо (например, при открытии диалогового окна).

OnStop() (Остановлено): Приложение более не видно пользователю и переходит в фоновый режим.

OnDestroy() (Уничтожено): Приложение завершается, освобождая ресурсы.

Жизненный цикл приложения в Xamarin

OnCreate() → OnStart()

Когда приложение только что создано (OnCreate()), происходит инициализация основных компонентов и ресурсов. Затем приложение переходит в состояние OnStart(), становясь видимым для пользователя, но ещё не активным.

OnStart() → OnResume()

После запуска (OnStart()) приложение становится активным (OnResume()) и готовым к взаимодействию с пользователем. В этом состоянии приложение отображается на переднем плане и обрабатывает события пользователя.

OnResume() → OnPause()

Если приложение временно теряет фокус (например, когда поверх него открывается другое приложение), оно переходит в состояние OnPause(). В этом состоянии приложение больше не активно, но всё ещё видно пользователю.

OnPause() → OnStop()

Когда приложение больше не видно пользователю (например, когда пользователь открывает другое приложение на весь экран), оно переходит в состояние OnStop(). В этом состоянии приложение находится в фоновом режиме и не активно.

OnStop() → OnDestroy()

Когда приложение закрывается (либо пользователем, либо системой), оно переходит в состояние OnDestroy(), где завершает свою работу и освобождает ресурсы.

OnStop() → OnRestart()

Если приложение было остановлено (OnStop()) и пользователь снова его открыл, приложение переходит в состояние OnRestart(), а затем повторно в OnStart() и OnResume().

**Билет №20**

**1. Работа с Supabase. Процесс подключения, система авторизации через OTP.**

**2. Создать мобильное приложение с одной кнопкой «Авторизация», при нажатии на которую вызывается диалоговое окно с вводом данных для авторизации (логин, пароль). Список заполнить тестовыми данными. Создать алгоритм работы мобильного приложения.**

Supabase - это open-source платформа для разработки, представляющая собой альтернативу Firebase. Она предоставляет широкий спектр сервисов, включая базу данных PostgreSQL, аутентификацию, хранилище файлов, функции и многое другое.

OTP (One-Time Password) — это одноразовый пароль, который используется для аутентификации пользователя. Этот пароль генерируется случайным образом и действителен только для одного сеанса или транзакции. Он обеспечивает дополнительный уровень безопасности по сравнению с традиционными паролями, так как даже если одноразовый пароль будет перехвачен, его нельзя будет использовать повторно.

**Зависимости**

implementation(platform("io.github.jan-tennert.supabase:bom:VERSION"))

implementation("io.github.jan-tennert.supabase:postgrest-kt")

implementation("io.github.jan-tennert.supabase:auth-kt")

implementation("io.github.jan-tennert.supabase:realtime-kt")

**Подключение**

val supabase = createSupabaseClient(

supabaseUrl = "https://xyzcompany.supabase.co",

supabaseKey = "public-anon-key"

) {

install(Auth)

install(Postgrest)

//install other modules

}

**Авторизация через OTP**

supabase.auth.signInWith(OTP) {

phone = "+4912345679"

}

supabase.auth.signInWith(OTP) {

email = "example@email.com"

}

supabase.auth.verifyEmailOtp(type = OtpType.Email.EMAIL, email = "example@email.com", token = "token")

supabase.auth.verifyPhoneOtp(type = OtpType.Phone.SMS, phone = "+491234567", token = "token")

**Билет №21**

**1. Элемент Dialog: виды, назначение, способы применения, пользовательские шаблоны.**

**2. Разработать интерфейс мобильного приложения с использованием списка (Приложение №7). При выборе цвета из списка изменить цвет элемента внизу экрана согласно данному цвету.**

**Виды диалоговых окон и их назначение**

**Xamarin.Forms**

**AlertDialog**. Диалоговое окно с заголовком, до трёх кнопок, списком выбираемых элементов или настраиваемым макетом.

**DatePickerDialog**. Диалоговое окно с предопределённым пользовательским интерфейсом, позволяющим выбрать дату.

**TimePickerDialog**. Диалоговое окно с предопределённым пользовательским интерфейсом, позволяющим выбрать время.

В Xamarin.Forms для взаимодействия с пользователем через всплывающее окно используются три метода класса **Page**:

**DisplayAlert**. Отображает модальное всплывающее окно для оповещения пользователя или с целью задать ему простые вопросы.

**DisplayActionSheet**. Создаёт всплывающее окно, которое предоставляет возможность выбрать одно из ряда действий.

**DisplayPromptAsync**. Позволяет отобразить подсказку, передав заголовок и сообщение в качестве строковых аргументов.

**Kotlin (Android)**

**AlertDialog:** Это диалоговое окно для отображения сообщений и получения простых ответов от пользователя. Оно может содержать заголовок, сообщение и кнопки для подтверждения или отмены.

**ProgressDialog**: Это диалоговое окно, отображающее индикатор прогресса, например, во время выполнения длительных операций. Однако класс ProgressDialog устарел в API 26, и рекомендуется использовать ProgressBar вместо него.

**DatePickerDialog**: Диалоговое окно для выбора даты. Оно позволяет пользователю выбрать дату из календаря.

**TimePickerDialog**: Диалоговое окно для выбора времени. Оно позволяет пользователю выбрать время из часов и минут.

**Способы применения диалоговых окон**

Уведомления и предупреждения: Используются для вывода важной информации и запросов на подтверждение.

Ввод данных: Окна ввода (Prompt) позволяют собирать информацию от пользователей, такую как логин, пароль или другие данные.

Выбор действий: ActionSheet или аналогичные механизмы предоставляют пользователям возможность выбора из нескольких вариантов.

**Создание пользовательских шаблонов**

**Xamarin.Forms**

**Создание пользовательского контента для диалога:**

public class CustomDialog : ContentPage

{

public CustomDialog()

{

Content = new StackLayout

{

VerticalOptions = LayoutOptions.Center,

HorizontalOptions = LayoutOptions.Center,

Children =

{

new Label { Text = "Введите данные:" },

new Entry { Placeholder = "Логин" },

new Entry { Placeholder = "Пароль", IsPassword = true },

new Button

{

Text = "ОК",

Command = new Command(() => Navigation.PopModalAsync())

}

}

};

}

}

**Открытие пользовательского диалогового окна:**

private async void OnCustomDialogButtonClicked(object sender, EventArgs e)

{

await Navigation.PushModalAsync(new CustomDialog());

}

**Kotlin (Android)**

**Создание пользовательского макета диалога (res/layout/custom\_dialog.xml):**

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<EditText

android:id="@+id/login"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Логин"/>

<EditText

android:id="@+id/password"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Пароль"

android:inputType="textPassword"/>

<Button

android:id="@+id/sendButton"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="ОК"/>

</LinearLayout>

**Создание и показ пользовательского диалогового окна в Activity:**

val dialogView = LayoutInflater.from(this).inflate(R.layout.custom\_dialog, null)

val dialogBuilder = AlertDialog.Builder(this)

.setView(dialogView)

.setTitle("Пользовательский диалог")

val alertDialog = dialogBuilder.show()

val login = dialogView.findViewById<EditText>(R.id.login)

val password = dialogView.findViewById<EditText>(R.id.password)

val sendButton = dialogView.findViewById<Button>(R.id.sendButton)

sendButton.setOnClickListener {

alertDialog.dismiss()

// Ваш код для обработки данных логина и пароля

}