Кнопка прокрутки

В ряде приложений применяется кнопка перемещение в начало контента, то есть фактически кнопка прокрутки в начало. Посмотрим, как сделать такую кнопку на Jetpack Compose. Для этого рассмотрим следующее приложение:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.animation.AnimatedVisibility
import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.OutlinedButton
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.launch

class MainActivity : ComponentActivity() {


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val listState = rememberLazyListState()
            val coroutineScope = rememberCoroutineScope()
            val displayButton =remember { derivedStateOf {
listState.firstVisibleItemIndex > 5 } }
            Box(Modifier.fillMaxSize()) {
                LazyColumn(state = listState) {
                    items(30) {
                        Text("Item $it", Modifier.padding(8.dp), fontSize = 28.sp)
                    }
                }
                AnimatedVisibility(visible = displayButton.value,
Modifier.align(Alignment.BottomCenter)
```

```
                ) {
                    OutlinedButton(
                        onClick = { coroutineScope.launch {
listState.scrollToItem(0) } },
                        border = BorderStroke(1.dp, Color.Gray),
                        shape = RoundedCornerShape(50),
                        colors = ButtonDefaults.outlinedButtonColors(contentColor
= Color.DarkGray),
                        modifier = Modifier.padding(5.dp)
                    ) { Text("Top", fontSize = 22.sp) }
                }
            }
        }
    }
}
```

Для управления прокруткой списка применяется состояние LazyListState:

```
val listState = rememberLazyListState()
```

Поскольку функции программной прокрутки представляют suspend-функции, и соответственно их надо запускать из корутин, то для программной прокрутки определятся область корутины:

```
val coroutineScope = rememberCoroutineScope()
```

И также определяется производное состояние, которое указывает, будет ли отображаться кнопка прокрутки:

```
val displayButton =remember { derivedStateOf { listState.firstVisibleItemIndex > 5
} }
```

В данном случае мы говорим, что этот состояние будет равно true, если индекс первого видимого элемента в списке больше 5. А это значит, что кнопка прокрутки будет отображаться. Если же мы находимся в самом начале списка (индекс первого видимого элемента в списке равен или меньше 5), то нет смысла отображать кнопку прокрутки, поэтому это состояние будет равно false.

Для хранения всего интерфейса определяется компонент Box. Здесь список, для которого определяется прокрутка, определен в компоненте LazyList. В данном случае это просто 30 компонентов Text:

```
LazyColumn(state = listState) {
    items(30) {
        Text("Item $it", Modifier.padding(8.dp), fontSize = 28.sp)
    }
}
```

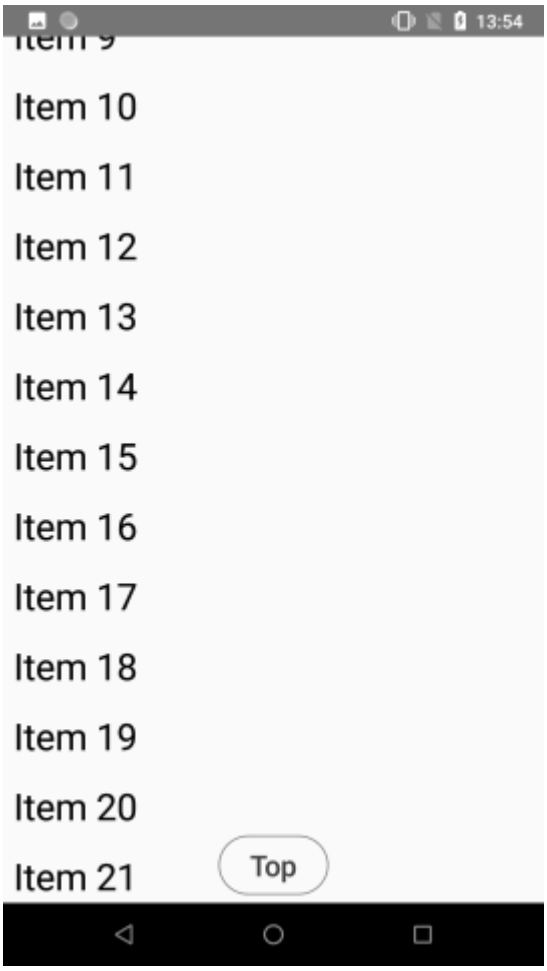
Также в Box располагается элемент AnimatedVisibility, который представляет анимацию видимости и будет управлять видимостью кнопки:

```
AnimatedVisibility(visible = displayButton.value,
Modifier.align(Alignment.BottomCenter)) {
    OutlinedButton(
        onClick = { coroutineScope.launch { listState.scrollToItem(0) } },
        border = BorderStroke(1.dp, Color.Gray),
        shape = RoundedCornerShape(50),
        colors = ButtonDefaults.outlinedButtonColors(contentColor =
Color.DarkGray),
        modifier = Modifier.padding(5.dp)
    ) { Text("Top", fontSize = 22.sp) }
 }
```

Этот компонент будет накладываться на список LazyList, но благодаря состоянию displayButton, которое зависит от индекса первого видимого элемента списка, мы сможем автоматически и динамически управлять видимостью этого компонента.

Непосредственно сама кнопка представлена компонентом OutlinedButton - кнопкой, по нажатию на которую происходит запуск корутины, в которой выполняется переход в начало списка с помощью вызова listState.scrollToItem(0)

Таким образом, если мы уйдем вниз по списку, то мы увидим кнопку возврата в начало списка:

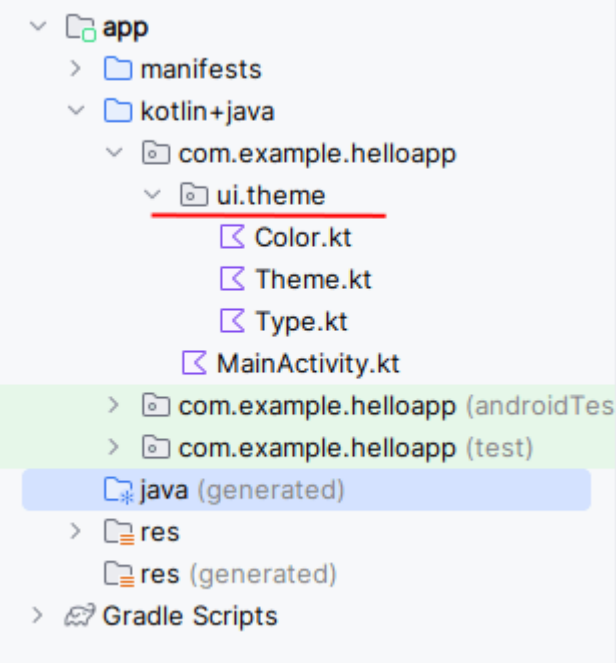


Темы. Material Design

Чтобы обеспечить определенный уровень единообразия дизайна между различными приложениями с точки зрения выбора цвета, типографики, формы и других параметров пользовательского интерфейса компания Google разработала набор рекомендаций, который называется Material Design. Помимо рекомендаций по проектированию, Material Design также включает набор компонентов пользовательского интерфейса и тем, которые можно использовать при разработке макетов пользовательского интерфейса.

Стоит отметить, что сам Material Design постоянно развивается. Так, на момент написания текущей статьи в Android Studio (в частности, в версии Android Studio Iguana) по умолчанию используется Material Design 3. Material Design 3 обеспечивает основу для Material You — функциональности, представленной в Android 12, которая позволяет приложению применять настройки, установленные пользователем на устройстве.

При создании проекта по умолчанию в Android Studio, например, по умолчанию в проект добавляется каталог ui.theme, который содержит ряд файлов:



Сама тема объявляется в файле Theme.kt, который начинается с объявления различных цветовых палитр для использования, когда устройство находится в светлом (дневном) или темном (ночном) режиме. Эти палитры создаются путем вызова функций darkColorScheme() и lightColorScheme(). Параметры этих функций представляют цветовые слоты, которые задают цвета для различных ситуаций:

```
private val DarkColorScheme = darkColorScheme(
        primary = Purple80,
        secondary = PurpleGrey80,
        tertiary = Pink80
)

private val LightColorScheme = lightColorScheme(
        primary = Purple40,
        secondary = PurpleGrey40,
        tertiary = Pink40

        /* Other default colors to override
    background = Color(0xFFFFFBFE),
    surface = Color(0xFFFFFBFE),
    onPrimary = Color.White,
    onSecondary = Color.White,
    onTertiary = Color.White,
    onBackground = Color(0xFF1C1B1F),
    onSurface = Color(0xFF1C1B1F),
    */
)
```

Стоит отметить, что Material Design 3 в общей сложности доступно более 30 цветовых слота, которые можно использовать при разработке темы:

- background
- error

- errorContainer
- inverseOnSurface
- inversePrimary
- inverseSurface
- onBackground
- onError
- onErrorContainer
- onPrimary
- onPrimaryContainer
- onSecondary
- onSecondaryContainer
- onSurface
- onSurfaceVariant
- onTertiary
- onTertiaryContainer
- outline
- outlineVariant
- primary
- primaryContainer
- scrim
- secondary
- secondaryContainer
- surface
- surfaceBright
- surfaceContainer
- surfaceContainerHigh
- surfaceContainerHighest
- surfaceContainerLow
- surfaceContainerLowest
- surfaceDim
- surfaceTint
- surfaceVariant
- tertiary
- tertiaryContainer

Если какой-то слот не указан, то для него применяются значения по умолчанию.

Эти цветовые слоты используются рядом компонентов для установки цвета. Например, слот primary используется в качестве цвета фона для кнопки. Фактические цвета, назначенные слотам, объявлены в файле Color.kt:

```
val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCCC2DC)
val Pink80 = Color(0xFFEFB8C8)

val Purple40 = Color(0xFF6650a4)
```

```
val PurpleGrey40 = Color(0xFF625b71)
val Pink40 = Color(0xFF7D5260)
```

Темы Material Design 3 также могут включать поддержку динамических цветов посредством вызовов функций dynamicDarkColorScheme() и dynamicLightColorScheme(), передавая текущий локальный контекст в качестве параметра. Эти функции затем сгенерируют цветовые схемы, соответствующие настройкам пользователя на устройстве. Поскольку динамические цвета поддерживаются только начиная с Android 12 (S), то для более ранних версий Android определяются резервные цветовые темы:

```
@Composable
fun HelloAppTheme(
        darkTheme: Boolean = isSystemInDarkTheme(),
        // динамические цвета доступны только на Android 12+
        dynamicColor: Boolean = true,
        content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (darkTheme) dynamicDarkColorScheme(context) else
dynamicLightColorScheme(context)
        }
        // резервные цветые темы, если Android 11 и ниже
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }
    val view = LocalView.current
    if (!view.isInEditMode) {
        SideEffect {
            val window = (view.context as Activity).window
            window.statusBarColor = colorScheme.primary.toArgb()
            WindowCompat.getInsetsController(window,
view).isAppearanceLightStatusBars = darkTheme
        }
    }

    MaterialTheme(
            colorScheme = colorScheme,
            typography = Typography,
            content = content
    )
}
```

Обратите внимание, что динамические цвета будут работать только в том случае, если они включены на устройстве пользователем в настройках в разделе обоев и стилей приложения.

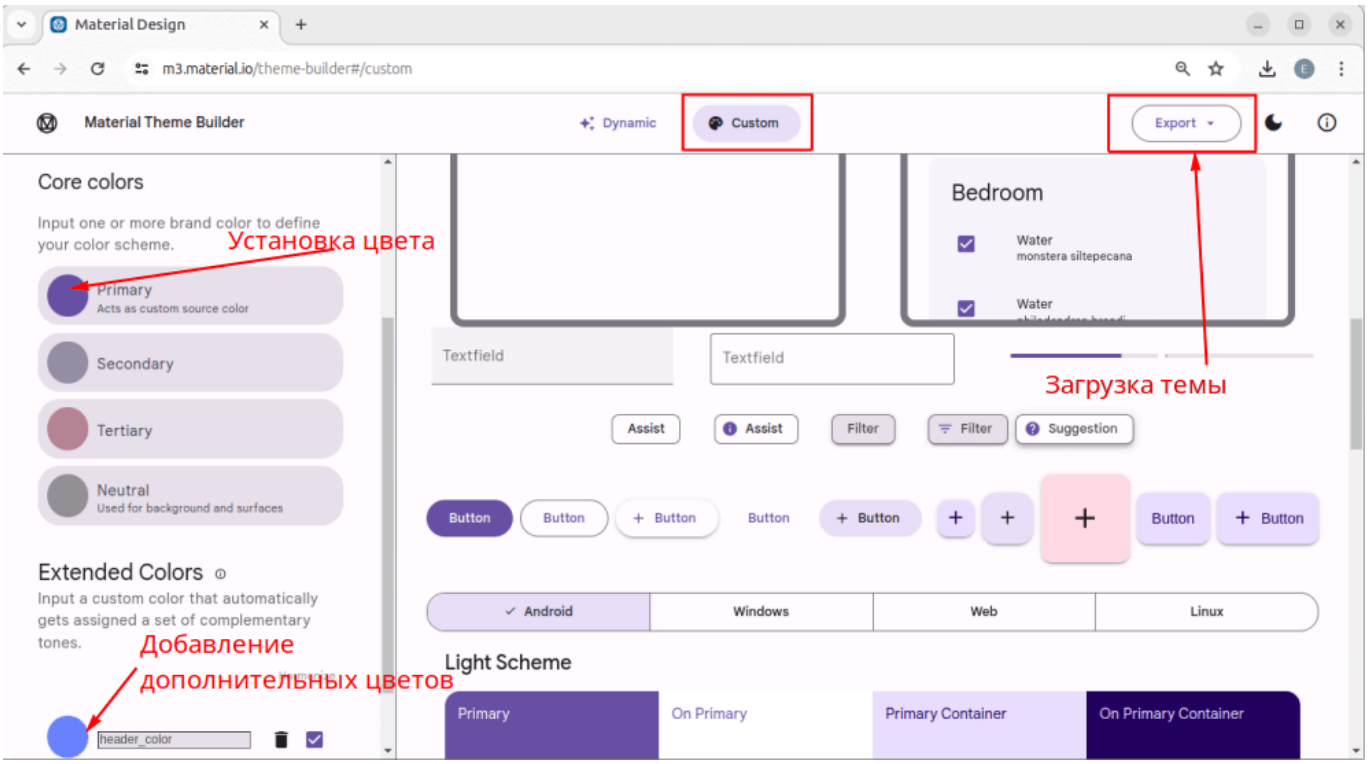
Также по умолчанию проект имеет настройки типографики - в файле Type.kt. В частности, Material Design имеет набор масштабов шрифта (два из них закомментированы):

```
// Настройки типографики Material
val Typography = Typography(
        bodyLarge = TextStyle(
                fontFamily = FontFamily.Default,
                fontWeight = FontWeight.Normal,
                fontSize = 16.sp,
                lineHeight = 24.sp,
                letterSpacing = 0.5.sp
        )
        /* Остальные стили текста
    titleLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 22.sp,
        lineHeight = 28.sp,
        letterSpacing = 0.sp
    ),
    labelSmall = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Medium,
        fontSize = 11.sp,
        lineHeight = 16.sp,
        letterSpacing = 0.5.sp
    )
    */
)
```

Создание темы

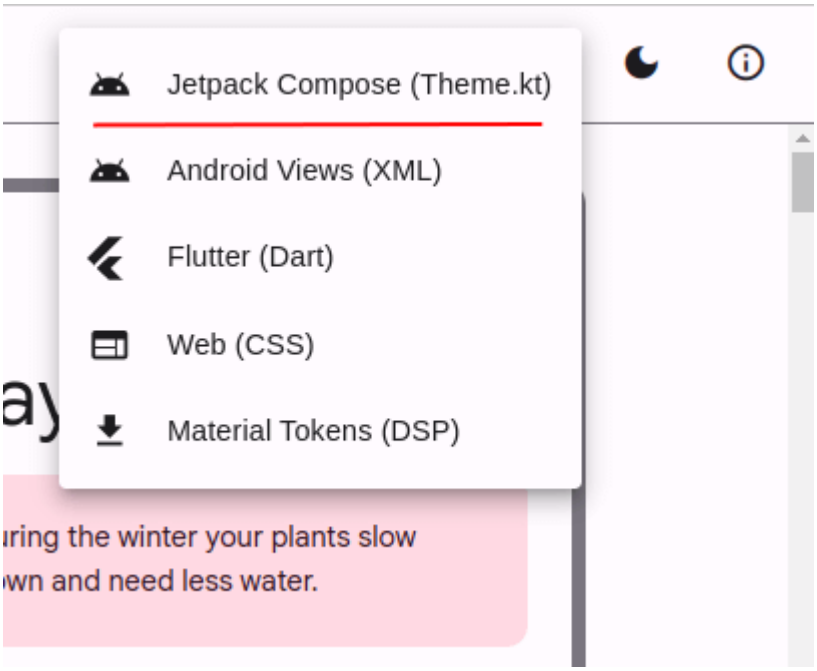
Создание собственной темы включает в себя редактирование этих файлов с использованием различных цветов, типографики и настроек формы. Эти изменения затем будут использоваться компонентами Material, которые составляют пользовательский интерфейс приложения.

Для создания темы в графическом режиме можно использовать такой инструмент как Material Theme Builder, который предоставляется компанией Google и который доступен по адресу https://m3.material.io/theme-builder#/custom:



На этой странице надо нажать в верхней части экрана на кнопку "Custom". Затем в разделе "Core colors" можно нажать на кружок цвета и установить любой цвет

Когда тема будет готова, нажмите кнопку "Export" в правом верхнем углу и выбрать опцию «Jetpack Compose (Theme.kt)». При появлении запроса сохраните файл в подходящем месте файловой системы вашего компьютера. Тема будет сохранена в виде архива с именем "material-theme.zip".



Распакуем загруженный файл и в папке "material-theme/ui/theme" мы сможем найти два файла:

- Color.kt
- Theme.kt

Для добавления этих файлрв проект сначала нужно удалить файлы старой темы - файлы Color.kt и Theme.kt из папки ui.theme проекта Android Studio. После удаления файлов возьмем файлы

пользовательских тем из распакованной папки "material-theme/ui/theme" и скопируум их в папку ui.theme проекта. После добавления файлов может потребоваться отредактировать новые файлы, чтобы объявление пакета в этих файлах соответствовало текущему проекту.

Биометрическая аутентификация

Многие устройства Android активно используют сенсорные датчики для различных задач, прежде всего для идентификации пользователя. И на уровне Jetpack Compose мы тоже можем задействовать эти возможности. Ключевыми компонентами биометрической аутентификации являются классы BiometricManager и BiometricPrompt. BiometricManager позволяет проверить, что устройство поддерживает биометрическую аутентификацию, и что пользователь включил необходимые параметры аутентификации (например, отпечатки пальцев или распознавание лица). А класс BiometricPrompt позволяет отобразить стандартное диалоговое окно, которое помогает пользователю пройти процесс аутентификации, выполнить аутентификацию и сообщить приложению результатов операции аутентификации.

Настройка проекта

Прежде всего надо учитывать, что для работы с биометрией уровень API должен быть как минимум 29 (Android 10). Поэтому перейдем в файл build.gradle.kts (Module :app) и изменим в нем значение android/defaultConfig/minSdk на 29:

```
.........................................
android {
    namespace = "com.example.helloapp"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.helloapp"
        minSdk = 29  // изменим минимальный уровень API на 29
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

.....................................
```

Кроме того, добавим необходимые зависимости в проект. Для этого изменим файл libs.version.toml изменим следующим образом:

```
[versions]
biometric = "1.2.0-alpha05"

.....................

[libraries]
androidx-biometric = { module = "androidx.biometric:biometric", version.ref =
"biometric" }
```

```
.....................
```

Затем в файл build.gradle.kts (Module :app) в секцию dependencies добавим следующую директиву:

```
dependencies {

    implementation(libs.androidx.biometric)
    ......................................
```

После этого нажмем на кнопку "Sync Now" для синхронизации проекта.

Настройка разрешений

Для поддержки аутентификации по отпечатку пальца и лицу требуется, чтобы приложение запрашивало разрешения USE_BIOMETRIC и CAMERA, а также функцию android.hardware.camera. Для этого перейдем к файлу манифеста AndroidManifest.xml и добавим в него соответствующие разрешения:

```
<?xml version="1.0" encoding="utf-8"?>
 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.biometricdemo" >


     <uses-feature android:name="android.hardware.camera" android:required="false"
/ >
     <uses-permission android:name="android.permission.USE_BIOMETRIC" / >
     <uses-permission android:name="android.permission.CAMERA" / >

.....................................................................
```

Пример биометрической аутентификации

Сначала определим все приложение, которое будет представлять следующий код:

```
package com.example.helloapp

import android.os.Bundle
import android.widget.Toast
import androidx.activity.compose.setContent
import androidx.biometric.BiometricManager
import androidx.biometric.BiometricPrompt
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
```

```
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.fragment.app.FragmentActivity


class MainActivity : FragmentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AuthenticationScreen()
        }
    }
}

@Composable
fun AuthenticationScreen() {
    var supportsBiometrics by remember { mutableStateOf(false) }
    val context = LocalContext.current as FragmentActivity
    val biometricManager = BiometricManager.from(context)

    supportsBiometrics = when (biometricManager.canAuthenticate(
        BiometricManager.Authenticators.BIOMETRIC_STRONG)) {
        BiometricManager.BIOMETRIC_SUCCESS -> true
        else -> {
            Toast.makeText(context, "Биометрия недоступна",
Toast.LENGTH_LONG).show()
            false
        }
    }

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Button(
            enabled = supportsBiometrics,
            onClick = {
                authenticate(context)
            },
            modifier = Modifier.padding(8.dp)
        ) {
            Text("Аутентификация", fontSize = 22.sp)
```

```
        }
    }
}

fun authenticate(context: FragmentActivity) {

    val executor = context.mainExecutor
    val biometricPrompt = BiometricPrompt(
        context,
        executor,
        object : BiometricPrompt.AuthenticationCallback() {

            override fun onAuthenticationSucceeded(
                result: BiometricPrompt.AuthenticationResult) {
                Toast.makeText(context, "Аутентификация пройдена",
Toast.LENGTH_LONG).show()
            }

            override fun onAuthenticationError(errorCode: Int, errString:
CharSequence) {
                Toast.makeText(context, "Ошибка при аутентификации: $errString",
Toast.LENGTH_LONG).show()
            }

            override fun onAuthenticationFailed() {
                Toast.makeText(context, "Не удалось пройти аутентификацию",
Toast.LENGTH_LONG).show()
            }
        })

    val promptInfo = BiometricPrompt.PromptInfo.Builder()
        .setTitle("Биометрическая аутентификация")
        .setDescription("Используйте отпечаток пальца или камеру для
аутентификации")
        .setNegativeButtonText("Отмена")

.setAllowedAuthenticators(BiometricManager.Authenticators.BIOMETRIC_STRONG)
        .build()

    biometricPrompt.authenticate(promptInfo)
}
```

В кратце разберем этот код. Прежде всего в качестве класса Activity здесь используется не стандартный ComponentActivity, а FragmentActivity:

```
class MainActivity : FragmentActivity() {
```

Дело в том, что класс ComponentActivity (на момент написания статьи) несовместим с BiometricPrompt, который применяется для отображения диалогового окна с подтверждением разрешений. Чтобы обойти эту проблему, нам нужно вместо этого создать подкласс MainActivity от класса FragmentActivity.

В качестве основного компонента, где производятся все действия, определен компонент AuthenticationScreen.

```
@Composable
fun AuthenticationScreen() {
    var supportsBiometrics by remember { mutableStateOf(false) }
    val context = LocalContext.current as FragmentActivity
    val biometricManager = BiometricManager.from(context)

    supportsBiometrics = when (biometricManager.canAuthenticate(
        BiometricManager.Authenticators.BIOMETRIC_STRONG)) {
        BiometricManager.BIOMETRIC_SUCCESS -> true
        else -> {
            Toast.makeText(context, "Биометрия недоступна",
Toast.LENGTH_LONG).show()
            false
        }
    }
```

Для отслеживания доступности биометрии компонент определяет переменную supportsBiometrics. С помощью свойства LocalContext.current компонент получает доступ к локальному контексту - текущему объекту Activity (в нашем случае FragmentActivity) и используют его для получения ссылки на объект BiometricManager. Затем у полученного объекта BiometricManager выполняется вызов метода canAuthenticate(), который проверяет доступность биометрии для текущего пользователя. И если аутентификация недоступна, отображается всплывающее сообщение. Если биометрия доступна, то в supportsBiometrics помещается значение true, и пользователь может пройти биометрическую аутентификацию.

Весь интерфейс компонента по сути состоит из одной кнопки:

```
Button(
    enabled = supportsBiometrics,
    onClick = {
        authenticate(context)
    },
    modifier = Modifier.padding(8.dp)
) {
    Text("Аутентификация", fontSize = 22.sp)
}
```

Прежде всего кнопка доступна, если только доступна биометрия. И в этом случае пользователь может нажать на кнопку, и в этом случае будет выполняться функция authenticate():

```
fun authenticate(context: FragmentActivity) {

    val executor = context.mainExecutor
    val biometricPrompt = BiometricPrompt(
```

```
        context,
        executor,
        object : BiometricPrompt.AuthenticationCallback() {

            override fun onAuthenticationSucceeded(
                result: BiometricPrompt.AuthenticationResult) {
                Toast.makeText(context, "Аутентифкация пройдена",
Toast.LENGTH_LONG).show()
            }

            override fun onAuthenticationError(errorCode: Int, errString:
CharSequence) {
                Toast.makeText(context, "Ошибка при аутентификации: $errString",
Toast.LENGTH_LONG).show()
            }

            override fun onAuthenticationFailed() {
                Toast.makeText(context, "Не удалось пройти аутентификацию",
Toast.LENGTH_LONG).show()
            }
        })
```

Сначала определяется объект BiometricPrompt, который настраивает диалоговое окно биометрического запроса и определяет набор методов обратного вызова аутентификации, которые можно вызывать для уведомления приложения об успехе или неудаче процесса аутентификации:

- onAuthenticationSucceeded(): вызывается при успешной аутентификации
- onAuthenticationError(): вызывается, если в процессе аутентификации произойдет ошибка
- onAuthenticationFailed(): вызывается, если пользователю не удалось пройти аутентификацию

Эти методы необходимо обернуть в объект класса BiometricPrompt.AuthenticationCallback.

И в конце собственно создается окно запроса:

```
val promptInfo = BiometricPrompt.PromptInfo.Builder()
    .setTitle("Биометрическая аутентификация")
    .setDescription("Используйте отпечаток пальца или камеру для аутентификации")
    .setNegativeButtonText("Отмена")
    .setAllowedAuthenticators(BiometricManager.Authenticators.BIOMETRIC_STRONG)
    .build()

biometricPrompt.authenticate(promptInfo)
```

Класс BiometricPrompt.PromptInfo.Builder создает новый экземпляр PromptInfo, настроенный с заголовком, подзаголовком и текстом описания, которые будут отображаться в диалоговом окне. Наконец, вызывается метод authenticate() экземпляра BiometricPrompt, которому передается объект PromptInfo.

В итоге при запуска приложения нам отобразится кнопка, по нажатию на которую отобразится диалоговое окно для ввода отпечатка пальца или сканирования лица. И нам надо будет приложенить палец для сканирования отпечатка, и при успешной аутентификации мы увидим соответствующее сообщение: