**Пошаговое руководство для написания кода в Android Studio**

Добавляем библиотеки

* В Android Manifest:

<uses-permission android:name="android.permission.INTERNET"/> — даём доступ к интернету

* В application:

android:usesCleartextTraffic="true" — даём разрешения дляиспользования HTTP протокола

* В build.gradle(module):

implementation("com.squareup.okhttp3:okhttp:4.12.0")

* В Main Activity:

private lateinit var request: Request

Дальше добавляем и инициализируем [View binding](Работа%20в%20студио.docx)

**ПИШЕМ ФУНКЦИИ**

**1) Функция для сохранения IP:**

private fun saveIP(ip: String){ // saveIP - название функции. В скобках указываем что будем передавать в неё IP при нажатии на кнопку.

val editor = pref.edit() // создаём переменную для хранения и считывания данных из памяти. Но чтобы использовать этот класс, создадим переменную pref для использования класса SharedPreferences. Эту переменную можно создать после переменной binding. Затем чтобы уже получить доступ к памяти, инициализируем эту перменную в onCreate (можно после setContentView): pref = getSharedPreferences( "MyPref", MODE\_PRIVATE). В ковычках указываем таблицу, то есть где будут сохраняться данные. Через запятую указываем мод, т.е. какие прилодения смогут получить доступ к этой таблице (MODE\_PRIVATE — только моё приложение). Дальше обращаемся к этой перменной, получаем этот класс и применяем функцию edit()

editor.putString("ip", ip) // обращаемся к перменной editor и применяем функцию putString, то есть дословно «положить строку в эту перменную». Указываем ключ, по которому записываем ("ip"); позже этот же ключ нужно использовать чтобы получить данные. Потом указываю данные которые хочу получить (ip). То есть какой стринг мы в нашу функцию saveIP передадим, такой и запишется сюда по ключевым словом ("ip").

editor.apply() // теперь нужно применить эти данные, применить эти изменения при помощи функции .apply()

}

**Вот описание работы функции:**

Эта функция saveIP принимает строку IP в качестве параметра и сохраняет ее в общих настройках (shared preferences) под ключом "ip". Вот как она работает:

1. val editor = pref.edit() - Создается объект редактора (Editor) из общих настроек pref. Редактор используется для внесения изменений в общие настройки.

2. editor.putString("ip", ip) - С помощью метода putString добавляется строка IP в общие настройки с ключом "ip". Это позволяет связать IP-адрес с определенным ключом для дальнейшего извлечения.

3. editor.apply() - Вызывается метод apply(), чтобы сохранить внесенные изменения в общих настройках. Метод apply() сохраняет изменения асинхронно, что означает, что они будут применены немедленно без блокировки текущего потока выполнения.

Таким образом, после вызова этой функции с IP-адресом в качестве аргумента, данный IP-адрес будет сохранен в общих настройках под ключом "ip" для последующего использования или извлечения.

**2) Функция для сохранения IP из EditText по нажатию кнопки:**

private fun OnClickSaveIp() = with(binding){ // OnClickSaveIp - название функции. = with(binding) – то есть наша функция будет уже вместе с binding – теперь напрямую можно использовать идентификаторы

saveIpAdderss.setOnClickListener { saveIpAdderss – это id нашей кнопки, по которой будет сохраняться IP. Дальше устанавливаем слушатель нажатий .setOnClickListener{ } – то есть при нажатии на эту кнопку нужно выполнить действия в фигурных скобках (в данном случае вызвать функцию saveIP).

if (ipESP.text.isNotEmpty()) saveIP(ipESP.text.toString()) //проверяем сначала что наш EditText не пустой: ipESP – название поля для ввода IP. Проверяем на пустоту функцией .isNotEmpty(). Если не пустая, то передаём в функцию saveIP то что ну нас записано в поле ipESP. Преобразуем в текст и в строку с помощью функций .text.toString().

}

}

И затем вызываем эту функцию в onCreate (можно после private lateinit var pref: SharedPreferences )

**Вот описание работы функции:**

1. saveIpAdderss.setOnClickListener { ... } - Устанавливает обработчик нажатия на кнопку saveIpAdderss.

2. if (ipESP.text.isNotEmpty()) saveIP(ipESP.text.toString()) - Проверяет, что текстовое поле ipESP не пустое, и если не пустое, вызывает функцию saveIP с текстом из ipESP в качестве аргумента.

Использование with(binding) в данном контексте делает код более читаемым и позволяет уменьшить количество повторяющихся обращений к объекту binding.

**3) Фукнция для взятия из памяти уже сохранённого IP:**

private fun getIP() = with(binding){ // создаём функцию getIP с методом «= with(binding)» для получения доступа к EditText

val ip = pref.getString("ip", "") //проверяем что наш адрес который мы сохранили не равен нулю. Для этого берём из нашего класса pref строку из памяти при помощи функции .getString() и указываем ключ ("ip"), потому что в pref могут быть сохранены разыне объекты string, а мы сохранили IP-адрес именно под ключом “ip”. Потом через запятую указываем значение по умолчанию, потому что вдруг там не было ничего сохранено (Значением по умолчанию будет пустота, т.е. пустые ковычки “”)

if (ip != null){ // делаем проерку еслине арвен нулю

if (ip.isNotEmpty()) ipESP.setText(ip) // проверяем на пустоту, потому что даже если IP не равен null, то по умолчанию там стоит пустота. Если ip не пустая, то установим ip в наш EditText

}

}

Потом сразу после функции OnClickSaveIp() добавляем функцию getIP()

**Вот описание работы функции:**

1. С использованием with(binding) блок кода внутри функции работает с объектом binding (предположительно, объектом привязки, связанным с макетом XML) без необходимости повторять имя объекта binding.

2. val ip = pref.getString("ip", "") - Получает строку с именем "ip" из SharedPreferences (pref) и сохраняет ее в переменной ip. Если в SharedPreferences не существует значения с ключом "ip", то используется пустая строка в качестве значения по умолчанию.

3. if (ip != null) - Проверяет, что значение ip не равно null, чтобы избежать возможного NullPointerException.

4. if (ip.isNotEmpty()) ipESP.setText(ip) - Проверяет, что значение ip не пустое. Если значение не пустое, то устанавливает текст в текстовом поле ipESP (предположительно, поле ввода) равным значению ip.

Таким образом, данная функция getIP() позволяет получить значение из SharedPreferences по ключу "ip" и установить это значение в текстовом поле ipESP, если оно не пустое. Весь блок кода выполняется в контексте объекта binding, что упрощает доступ к элементам пользовательского интерфейса, связанным с данными объекта привязки.

**4) Функция отправки данных по определённому IP-адресу:**

- Эта функция будет передавать данные и принимать их

- Как только функция передала данные, сразу переходит в ждущий режим и ждёт чтобы получить ответ

private fun post(post: String){ // создаём функцию post. При нажатии на кнопку будем передавать некий string (у нас это название светодиодов). Соответственно в качестве аргумента указываем

Thread{ // включаем второстепенный поток. Создается асинхронная задача. Это позволяет выполнять код в отдельном потоке, чтобы не блокировать основной поток (UI поток) при выполнении сетевых операций.

request = Request.Builder().url("http://${binding.ipESP.text}/$post").build() //создаём главный запрос, который будем отправлять в локальную сеть. Это практически то же самое что и просто «http://192.168.1.50/light\_on», просто всё сделано по образцу. Сначала создаём объект request, который мо создали вначале класса, это и есть наш запрос. Далее обращаемся к классу Request из библиотеки okhttp3. Применяем функцию .Builder() - это метод, который позволяет создать объект Request.Builder(), который является построителем для объекта Request. Дальше использем функцию .url — это метод, который устанавливает URL для HTTP-запроса. И как раз HTTP-запрос формируется по шаблону, который в скобках. Чтобы в http-запрос мог вставляться разный ip и разные данные для отправки, мы используем шаблон — “${…}”; .build() - это метод, который завершает процесс построения объекта Request и возвращает готовый объект для отправки запроса.

try { //делаем попытку отправки, потому что могут выйти разыне ошибки

// создадим client в начале: private val client = OkHttpClient(). OkHttpClient указывает на то, что мы хотим создать HTTP-клиент для выполнения сетевых запросов в приложении. Клиенты HTTP позволяют нам взаимодействовать с удаленными серверами, отправлять HTTP-запросы и получать ответы.

var response = client.newCall(request).execute() //создаём специальный класс, который будет ждать данные с МК. Создаём функцию .newCall() с нашим запросом request в скобках (то есть то, что хотим передать), который мы создали выше. .execute() - запустить

if(response.isSuccessful){ // проверяем что наш запрос успешен, что всё инициализировалось и мы получили результат

val resulText = response.body?.string() // создаём переменную для температуры в которую запишем string. Берём мы его из response с помощью метода .body (к нам приходят разыне данные, не только string, но всякие заголовки, header) .string озаначает, что мы берём строку от туда. body может быть равен null, поэтому ставим знак ?. Т.е. если мы получем текст то записываем его сюда. Знак ? отбрасывает оствльные данные

runOnUiThread{ // Так как обновление пользовательского интерфейса должно выполняться в основном (UI) потоке, используется метод runOnUiThread.

binding.tvTemp.text = resulText // здесь мы просто обращаемся к tvTemp и устанавливаем текст из resulText

}

}

} catch (i: IOException){ // дословно: поймать ошибку при Input / Output Exception (при входе или выходе данных)

}

}.start() // запускаем второстепенный поток

}

**Вот описание работы функции:**

Эта функция post(post: String) предназначена для выполнения POST запроса на удаленный сервер. Давайте разберем работу функции по шагам:

1. Создается асинхронная задача с помощью Thread{...}.start(). Это позволяет выполнять код в отдельном потоке, чтобы не блокировать основной поток (UI поток) при выполнении сетевых операций.

2. Создается HTTP запрос с помощью Request.Builder().url("http://${binding.ipESP.text}/$post").build(). В данном случае, формируется запрос типа POST к URL, который состоит из IP адреса, полученного из binding.ipESP.text, и строки $post, переданной в функцию.

3. Выполняется запрос с помощью client.newCall(request).execute(), где client предполагается экземпляр OkHttpClient, используемый для выполнения сетевых запросов.

4. Проверяется успешность выполнения запроса с помощью if(response.isSuccessful). Если запрос был выполнен успешно, данные из ответа считываются с помощью response.body?.string() и результат присваивается переменной resulText.

5. Затем данные resulText отображаются на пользовательском интерфейсе в текстовом поле binding.tvTemp.text с помощью runOnUiThread{ binding.tvTemp.text = resulText }. Так как обновление пользовательского интерфейса должно выполняться в основном (UI) потоке, используется метод runOnUiThread.

6. Обработка возможных исключений типа IOException осуществляется с помощью пустого блока catch (i: IOException) { }. Данная конструкция позволяет обрабатывать ошибки, которые могут возникнуть при выполнении запроса.

Таким образом, эта функция выполняет POST запрос к указанному URL, обрабатывает ответ, и обновляет соответствующий элемент пользовательского интерфейса с полученными данными.

**5) Функция слушателя нажатий для кнопок:**

private fun onClickListenner(): OnClickListener { /создаём функция для слушания нажатий. Указываем, что возвращать эта функция будет OnClickListener. Он добавляется ко всем кнопкам

//после функции getIP() указываем binding класс, т.е. предоставляем доступ к кнопкам:

binding.apply {

lightOnOff.setOnClickListener(onClickListenner()) // lightOnOff – название кнопки.

// здесь можем добавить сколько угодно кнопок по шаблону выше

}

return OnClickListener{ // у всех кнопок будет один слушатель нажатий, поэтому дляопределения на какую кнопку мы нажали, будем использовать id

when(it.id){ //объект it несёт в себе id кнопки, на которую нажали. Проверяем по id какую кнопку нажали

R.id.light\_on\_off -> { post("led\_1") } // если id кнопки light\_on\_off, значит запускаем функцию post со строкой ("led\_1"), а дальше она пойдёт в http и на МК ESP

}

}

}

**Вот описание работы функции:**

Эта функция `onClickListenner()` используется для создания объекта `OnClickListener`, который будет привязан к определенным элементам пользовательского интерфейса. Поясним работу функции по шагам:

1. `private fun onClickListenner(): OnClickListener` - это определение функции `onClickListenner()`, которая возвращает объект типа `OnClickListener`.

2. `return OnClickListener { ... }` - создается экземпляр анонимного класса `OnClickListener`, который реализует метод `onClick()`.

3. `when(it.id) { ... }` - используется конструкция `when` для проверки идентификатора элемента, который был нажат. Переменная `it` представляет собой объект, вызвавший событие клика.

4. `R.id.light\_on\_off -> { post("led\_1") }` - если идентификатор элемента равен `R.id.light\_on\_off`, то вызывается функция `post("led\_1")`. Это означает, что при нажатии на элемент с идентификатором `light\_on\_off`, будет выполнен POST запрос на сервер с параметром "led\_1".

Таким образом, данная функция создает объект `OnClickListener`, который реагирует на клик элемента с определенным идентификатором, и выполняет соответствующее действие (в данном случае, отправляет POST запрос на сервер).