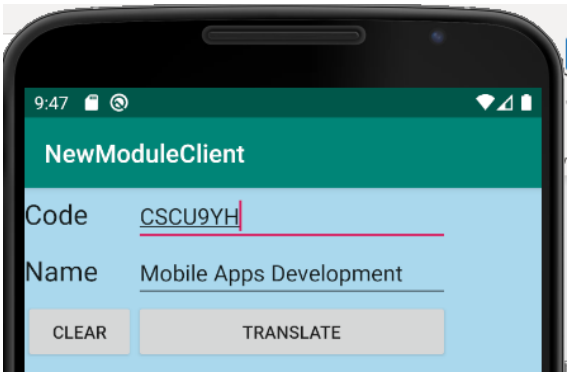
University of Stirling
Computing Science
Mobile App Development

Android Practical 6: Module Provider Database

Checkpoint at the End

In this practical you will learn about Android content providers. The aim is to create a content provider that translates module codes into names (e.g. CSCU9YH is translated into 'Mobile App Development'). You will also create a client activity for this provider.



This lab consists of **two applications** (each with their own activity). So you will need **two projects** for this lab task. The first application contains the provider, and the second contains the client application.

Create the Module Provider

Create a new project Call your project *ModuleProvider*. We will need two classes here, one for the Activity which is automatically generated and a second class for the Provider class. So, to create a new class for the Provider, create a new Kotlin Class file (select *java\uk.ac.stir.cs.provider* and right-click *New > Kotlin File/Class)* and set the *Name* to *ModuleProvider* (that's what is assumed in other code provided on this lab sheet). Open *ModuleProvider.kt* from the Explorer and start editing it.

The class *ModuleProvider* extends the standard *ContentProvider*. You will see that there are constants for the database table name, the provider name, the URI for accessing this, and URI codes corresponding to all modules or just one module. Later on, the *onCreate* method in the provider Activity establishes the link to the content provider object content and also creates a database reference.

In the Provider it is necessary to provide implementations for the *insert*, *update, query*, *delete*, *getType*, and *onCreate* database methods. These provide the interface between client apps and the content provider. So in structure there is similarity to the previous basic database practical code, but the content of these methods make use of Provider specific logic, namely using the URIMatcher to work out if the query is for a specific database record, or all. The SQLQueryBuilder class is another helper class to construct SQL queries.

```
package android.cs.stir.ac.uk.newmoduleprovider

import android.content.*
import android.database.Cursor
import android.database.SQLException
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper
import android.database.sqlite.SQLiteQueryBuilder
import android.net.Uri
import android.text.TextUtils

class ModuleProvider : ContentProvider() {
    lateinit private var contentResolver: ContentResolver
    lateinit private var modulesDatabase: SQLiteDatabase
    lateinit private var uriMatcher: UriMatcher

    override fun onCreate(): Boolean {
        uriMatcher = UriMatcher(UriMatcher.NO_MATCH)
        uriMatcher.addURI(PROVIDER_NAME, "modules", ALL_MODULES)
        uriMatcher.addURI(PROVIDER_NAME, "modules/*", ONE_MODULE)
        val context = context
        contentResolver = context!!.contentResolver
        val databaseHelper = DatabaseHelper(context)
        modulesDatabase = databaseHelper.writableDatabase
        return modulesDatabase != null
    }

    override fun getType(uri: Uri): String? {
        val mimeType: String
        mimeType = when (uriMatcher.match(uri)) {
            ALL_MODULES -> "vnd.uk.ac.stir.cs.cursor.dir/modules"
            ONE_MODULE -> "vnd.uk.ac.stir.cs.cursor.item/modules"
            else -> throw IllegalArgumentException("Invalid URI '$uri'")
        }
        return mimeType
    }

    override fun query(uri: Uri, projection: Array<String>?, selection: String?,
                       arguments: Array<String>?, sortOrder: String?): Cursor? {
        var so = sortOrder
        val sqlBuilder = SQLiteQueryBuilder()
        sqlBuilder.tables = DATABASE_TABLE
        if (uriMatcher.match(uri) == ONE_MODULE) {
            val code = uri.pathSegments[1]
            sqlBuilder.appendWhere("code = '$code'")
        }
        if (so == null || so === "") {
            so = "code"
        }
        val cursor = sqlBuilder.query(modulesDatabase, projection, selection,
            arguments, null, null, sortOrder)
        cursor.setNotificationUri(contentResolver, uri)
        return cursor
    }

    override fun insert(uri: Uri, values: ContentValues?): Uri? {
        val newUri: Uri
        val rowIdentifier = modulesDatabase.insert(DATABASE_TABLE, "", values)
        if (rowIdentifier > 0) {
            newUri = ContentUris.withAppendedId(CONTENT_URI, rowIdentifier)
            contentResolver.notifyChange(newUri, null)
        } else throw SQLException("Failed to insert row into '$uri'")
        return newUri
    }

    override fun delete(uri: Uri, selection: String?, arguments: Array<String>?): Int {
        var count = 0
        when (uriMatcher.match(uri)) {
            ALL_MODULES -> count = modulesDatabase.delete(DATABASE_TABLE, selection, arguments)
            ONE_MODULE -> {
                var code = uri.pathSegments[1]
```

```
                code = "code = '$code'"
                if (!TextUtils.isEmpty(selection)) code += " AND ($selection)"
                count = modulesDatabase.delete(DATABASE_TABLE, code, arguments)
            }
            else -> throw IllegalArgumentException("Unknown URI '$uri'")
        }
        contentResolver.notifyChange(uri, null)
        return count
    }

    override fun update(uri: Uri, values: ContentValues?, selection: String?,
                        arguments: Array<String>?): Int {
        var count = 0
        when (uriMatcher.match(uri)) {
            ALL_MODULES -> count = modulesDatabase.update(DATABASE_TABLE, values, selection,
arguments)
            ONE_MODULE -> {
                var code = uri.pathSegments[1]
                code = "code = '$code'"
                if (!TextUtils.isEmpty(selection)) code += " AND ($selection)"
                count = modulesDatabase.update(DATABASE_TABLE, values, code, arguments)
            }
            else -> throw IllegalArgumentException("Unknown URI $uri")
        }
        contentResolver.notifyChange(uri, null)
        return count
    }

    companion object {
        const val DATABASE_TABLE = "modules"
        const val PROVIDER_NAME = "android.cs.stir.ac.uk.newmoduleprovider.ModuleProvider"

        @JvmField
        val CONTENT_URI = Uri.parse("content://" + PROVIDER_NAME + "/" + DATABASE_TABLE)
        private const val ALL_MODULES = 1
        private const val ONE_MODULE = 2
    }
}

internal class DatabaseHelper(context: Context?) : SQLiteOpenHelper(context, DATABASE_NAME, null,
DATABASE_VERSION) {

    override fun onCreate(database: SQLiteDatabase) {
        database.execSQL(
                "CREATE TABLE " + ModuleProvider.DATABASE_TABLE + "(" +
                        "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                        "code TEXT NOT NULL," +
                        "name TEXT NOT NULL" +
                        ");"
        )
    }

    override fun onUpgrade(database: SQLiteDatabase, oldVersion: Int,
                           newVersion: Int) {
        database.execSQL("DROP TABLE IF EXISTS " + ModuleProvider.DATABASE_TABLE)
        onCreate(database)
    }

    companion object {
        val DATABASE_NAME = "courses"
        val DATABASE_VERSION = 1
    }
}
```

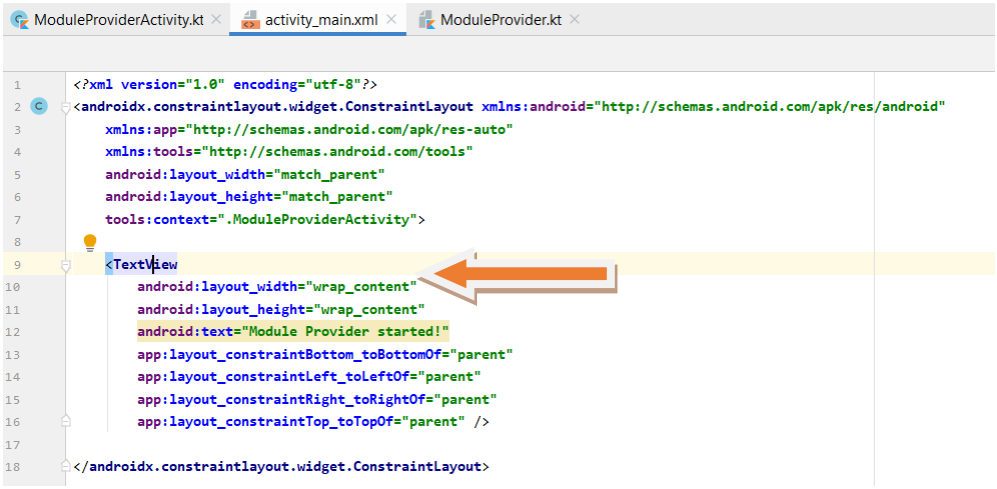
As in the previous lab, a separate DatabaseHelper class provides an interface to the built-in SQLite database. For each content provider URI, the getType method returns the resulting MIME type (Multipurpose Internet Mail Extensions). This names the type of data returned by the content provider when asked for module details; vnd stands for vendor (i.e. developer-defined). The onCreate method creates a content resolver (for accessing content) and creates a reference to the database.

Open AndroidManifest.xml and add the entry for the provider. This should go just after the activity (after its closing tag). The name of the provider will need to match the class name for the Provider class above. The authority needs to be the same whenever you reference the provider, e.g. in the Activity below and also in the client.

```
<provider
    android:name="ModuleProvider"
    android:authorities="android.cs.stir.ac.uk.newmoduleprovider.ModuleProvider"
    android:enabled="true"
    android:exported="true">
</provider>
```

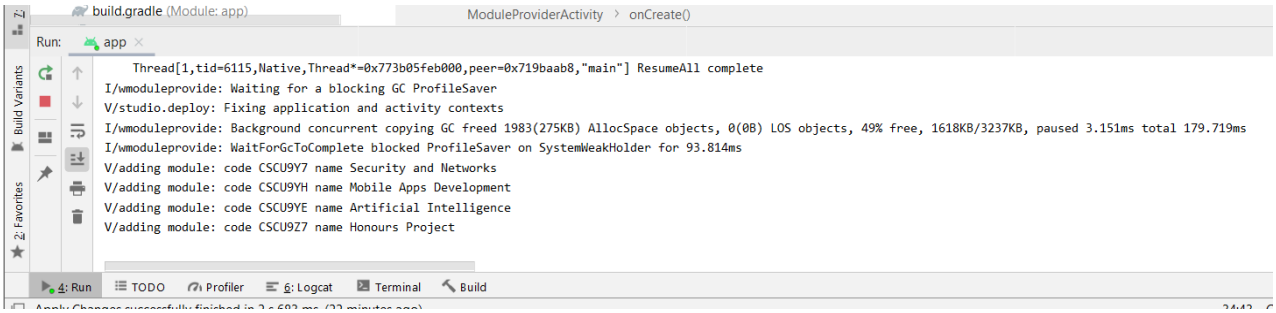
Create the *Module Provider Activity*

Then edit the Layout file for the project to include a *TextView* which includes the text 'Module Provider started'.



In the code that follows below for the provider Activity, the *onCreate* method sets up the database by calling the *addModule* method for each module. **As marked by the code comment in bolt-red, add a number of modules (about 3-5).**

Notice the use of method *Log.v* (short for verbose) which is called to log each module as it is added. This is a useful for debugging Android code (much like inserting print statements in a Java program). The log output appears under the *Run* tab. This should be visible, alternatively use *ALT+6*.



```
class ModuleProviderActivity : AppCompatActivity() {
    private lateinit var cr: ContentResolver

    fun addModule(code: String, name: String) {
        Log.v("adding module", "code $code name $name")
        val values = ContentValues()
        var uri = Uri.withAppendedPath(ModuleProvider.CONTENT_URI, code)
        values.put("code", code)
        values.put("name", name)
        val rows = cr.update(uri, values, "", null)
        if (rows == 0) {
            cr.insert(uri, values)
        }
    }

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        cr = contentResolver

        // delete any existing module table
        cr.delete(ModuleProvider.CONTENT_URI, null, null)

      // add modules
      //  addModule( string: code, string: title)

      // report completion
        setContentView(R.layout.activity_main)
    }
}
```

Once the content provider has been created, run the Activity and check that it reports initialisation of the database.

Create the Module Client

Create an Android project much as you did before. Call your project *ModuleClient*. Then edit the following:

Open *res\layout\activity_main.xml* (or whatever your layout file is called) and edit this to look like the screenshot at the beginning of the practical. The layout is similar to the second practical, with the following rows in the table layout:

- In the first *TableRow* on the left hand side place *TextView*. In the next column place and *EditText* element and give it Id *codeValue* and give it a width of *250dp*.
- In the second *TableRow* on the left hand side place a *TextView*. In the next column place *a EditText* and you may want to set the inputType attribute to *textMultiLine*. Give it the Id *nameValue* and set a width of *250dp*.
- In the third row place a *Button* and give it Id *clearButton* with text '*Clear'*; and then place a second *Button* with Id *translateButton* with text '*Translate'*.

When the Translate button is clicked, the content provider is asked to translate the given module code into its name. A managed query is used as this keeps track of the database cursor automatically. If the module name is missing or invalid (one row is not returned in the cursor), and an error is displayed.

```
package android.cs.stir.ac.uk.moduleclient

import androidx.appcompat.app.AppCompatActivity
import android.net.Uri
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.Toast

class ModuleClientActivity : AppCompatActivity() {
    lateinit var codeText: EditText
    lateinit var nameText: EditText
    val DATABASE_TABLE = "modules"
    val PROVIDER_NAME = " android.cs.stir.ac.uk.newmoduleprovider.ModuleProvider"
    val CONTENT_URI = Uri.parse("content://" + PROVIDER_NAME + "/" + DATABASE_TABLE)

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        codeText = findViewById<View>(R.id.codeValue) as EditText
        nameText = findViewById<View>(R.id.nameValue) as EditText
        nameText.isFocusable = false

        val clearButton = findViewById<View>(R.id.clearButton) as Button
        clearButton.setOnClickListener {
            codeText.setText("")
            nameText.setText("")
        }

        val translateButton = findViewById<View>(R.id.translateButton) as Button
        translateButton.setOnClickListener { view ->
            try {
                val moduleCode = codeText.text.toString()
                val uri = Uri.parse(CONTENT_URI.toString() + "/" + moduleCode)
                val cursor = contentResolver.query(uri, null, null, null, null)
                if (cursor!!.count == 1) {
                    cursor.moveToFirst()
                    val moduleName = cursor.getString(cursor.getColumnIndex("name"))
                    nameText.setText(moduleName)
                    cursor.close()
                } else {
                    nameText.setText("")
                    throw Exception("module code invalid")
                }
            } catch (exception: Exception) {
                Toast.makeText(view.context, "Invalid data - " + exception.message,
Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

Edit the Client Manifest file

Since API level 30, Android applications are more restricted in terms of their "package visibility", that is one application cannot, by default, see other packages from other applications. More information is available at https://developer.android.com/training/basics/intents/package-visibility.

In order for the client app to "see" the provider app we need to make a small change to the **client's Manifest file**. You need to **add a queries tag** at the beginning of the file to match the authorities tag of the provider to allow the client to "see" the content provider in the provider app:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="uk.ac.stir.cs.moduleclient">

    <queries>
        <provider android:authorities=" android.cs.stir.ac.uk.newmoduleprovider.ModuleProvider" />
    </queries>

    <application
        …
```

Run your code and check that it behaves as expected (including error cases such as no module code or an invalid one).

You have now reached a checkpoint. Please record a short screen grab video and upload to Canvas for your checkpoint.

Consider the following questions:

- **Why are different projects used for the provider and the client?**
- **Does the provider really need an associated activity?**