

StorageStorm - Documentation



Graphical Front-End for the NoSQL DBMS based on a
new Data Model

Eman Basic

Table of Contents

[1. Introduction](#)

[2.Tools and Technologies](#)

[3. StorageStorm structure](#)

[3.1. SplashActivity.kt](#)

[3.2. MainActivity.kt](#)

[3.3. DatabaseFragment.kt](#)

[3.4. SearchFragment.kt](#)

[3.5. SearchResultActivity.kt](#)

[3.6. AboutDataObject.kt](#)

[3.7. EditObjectActivity.kt](#)

[3.8. ManageFragment.kt](#)

[3.9. CreateObjectActivity.kt](#)

[3.10. Dialogs](#)

[4. Implementation details](#)

[4.1. Persistent fragment states in the MainActivity.kt](#)

[4.2. Adapters](#)

[4.3. SQLite database](#)

[4.4. BaseDialog.kt](#)

[4.5. AsyncTask](#)

[4.6. Result messages](#)

1. Introduction

The Database Management System ([DMBS](#)) is a software that is able to create, manipulate, retrieve and manage data in a database. Usually, an end-user access to the DBMS is implemented as a Graphical User Interface ([GUI](#)). In this case, it is a mobile application for the [Android operating system](#).

The StorageStorm application is a part of an international bachelor project called “NoSQL DBMS based on a new Data Model” which was a cooperation between students from the [Technical University Graz](#) and the [Peter the Great St.Petersburg Polytechnic University](#).

2.Tools and Technologies

[Android Studio](#)

Android Studio is the official Integrated Development Environment (IDE) that is built on [JetBrains'](#) IntelliJ. It is a professional tool designed to provide all needed functionalities for the full development of Android applications.

[Kotlin](#)

The application is completely written in Kotlin, a language presented by JetBrains and officially announced by Google as the preferred language for Android development. More specifically, it uses Kotlin v1.3.21.

[XML](#)

The Extensible Markup Language (XML) is a programming language used for simple encoding of documents. In Android development, it is used to create the majority of files like Android Manifest or the layout of screens, called [Activites](#).

[Gradle](#)

The gradle file build.gradle is one of the most important files in an Android project. Gradle is an automated build system built on a [Java Virtual Machine](#) (JVM) uses all application source files to create an Android Package Kit (APK) file, a compressed file that contains all the elements that an app needs to be correctly installed on an android device.

[SQLite](#)

SQLite Is a small C-Language library that implements a fast, highly-reliable, self-contained SQL database engine. It provides a persistent local data in Android applications.

[Anko](#)

Anko is a Kotlin library that simplifies the code, makes it clean and easy to read.

[JsonHandleView](#)

Is an open-source library created by Github user [stven0king](#) used for graphical presentation of json files.

[Ikarus API](#)

The Ikarus Application Programming Interface ([API](#)) is a [Java Library](#) that is used to enable the application to communicate with the [Ikarus Database Engine](#) (Ikarus DE).

3. StorageStorm structure

The application consists of Activities, Fragments and Dialogs.

An [Activity](#) is a fundamental building block of Android applications. They are used as the entry point for a user's interaction with an app and they also play an important role in how the user navigates within the app (e.g., with the Back button) or between apps (e.g., with the Recents button).

A [Dialog](#) is a small window that prompts to the user to make a decision or enter some information. A dialog is more like a pop-up and it does not fill the whole screen. Normally, the activity from which the dialog has been started is shown behind the dialog and appears to be in it's shadow.

A [Fragment](#) represents a behavior or a portion of user interface. It is hosted by an Activity and a single activity can host multiple fragments. The fragments are bound to activity's life cycle, and are created/destroyed when the activity is created/destroyed. A life cycle is one of the most important concepts in android development. Other activities/dialogs can also be started from fragments.

Each activity/dialog/fragment has its own XML layout file that defines what the user will see in the app.

Dialogs	Activities	Fragments
BaseDialog.kt EditObjecDialog.kt DeleteObjectDialog.kt AddObjectToCollectionDialog.kt RemoveObjectFromCollectionDialog.kt CreateCollectionDialog.kt DeleteCollectionDialog.kt GetCollectionDialog.kt AddCollectionToCollectionDialog.kt RemoveCollectionFromCollectionDialog.kt	SplashActivity.kt MainActivity.kt SearchResultActivity.kt AboutDataObjectActivity.kt AboutCollectionActivity.kt CreateObjectActivity.kt EditObjectActivity.kt	DatabaseFragment.kt SearchFragment.kt ManageFragment.kt

List of all activities/fragments/dialogs

3.1. SplashActivity.kt

Splash screens typically serve to enhance the look and feel of an application, hence they are often visually appealing. They may also be animated, include graphical or sound effects.

The implementation of the SplashActivity.kt is simple, it is used as the entry point for the app, and all it does is starting the [MainActivity.kt](#), which is the “real” entry point of the app. In larger apps or in games, the splash screen may be used to load important elements in the background before the user starts interacting with the app.



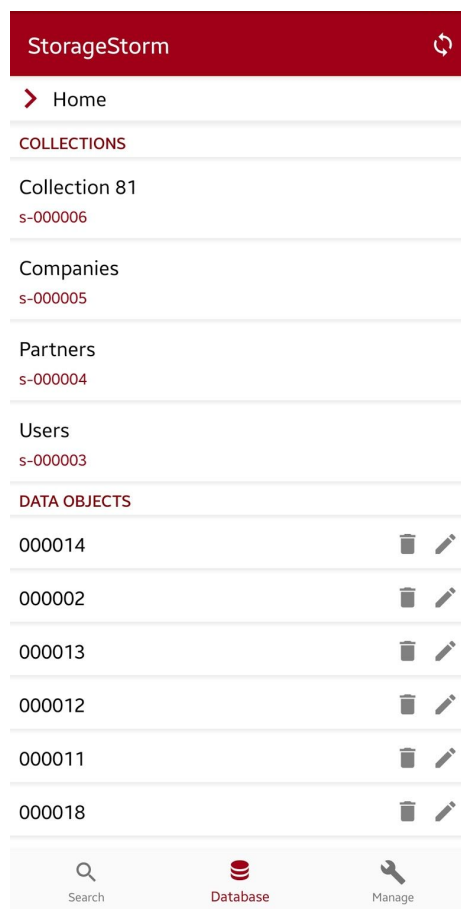
3.2. MainActivity.kt

The MainActivity consists of three fragments: [DatabaseFragment](#), [SearchFragment](#), and [ManageFragment](#).

The only thing that belongs to the MainActivity in this layout is the bottom bar that is used to navigate between fragments, and the toolbar that the fragments share.

A feature worth mentioning is that each fragment is created only when the MainActivity is created, and destroyed only when the MainActivity is destroyed. This approach gives us the benefit of not losing the states of fragments, i.e. if you navigate deeply through the database using the [DatabaseFragment](#), and then switch to the [SearchFragment](#), search for a field in data objects, and then go back to the [DatabaseFragment](#) without losing the last known state.

Pressing the refresh button on the [toolbar](#) sends a [broadcast](#) that the content should be refreshed.



3.3. DatabaseFragment.kt

The DatabaseFragment consists of two [recycler views](#), one for representing the path and the second for displaying the actual content. A [recycler view](#) is a container for the presentation of a large data set of views that can be recycled and scrolled very efficiently.

- Behaviour of the path recycler view

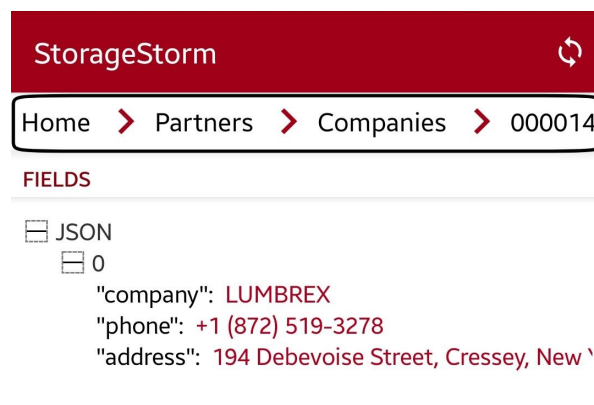
When the app is started, the path is always set to HOME, and this means that all collections and data objects are displayed in the content [recycler view](#).

When the user selects a collection, e.g. collection Users, the path is updated to HOME > Users. This gives the user an overview of the database structure.

If Home or Users is pressed, the content is displayed accordingly.

Pressing the back button results in removing the last element from the path [recycler view](#) and shows the content of the previous element.

If HOME is the only element in the path, a back button press will exit the app.



- Behaviour of the content [recycler view](#)

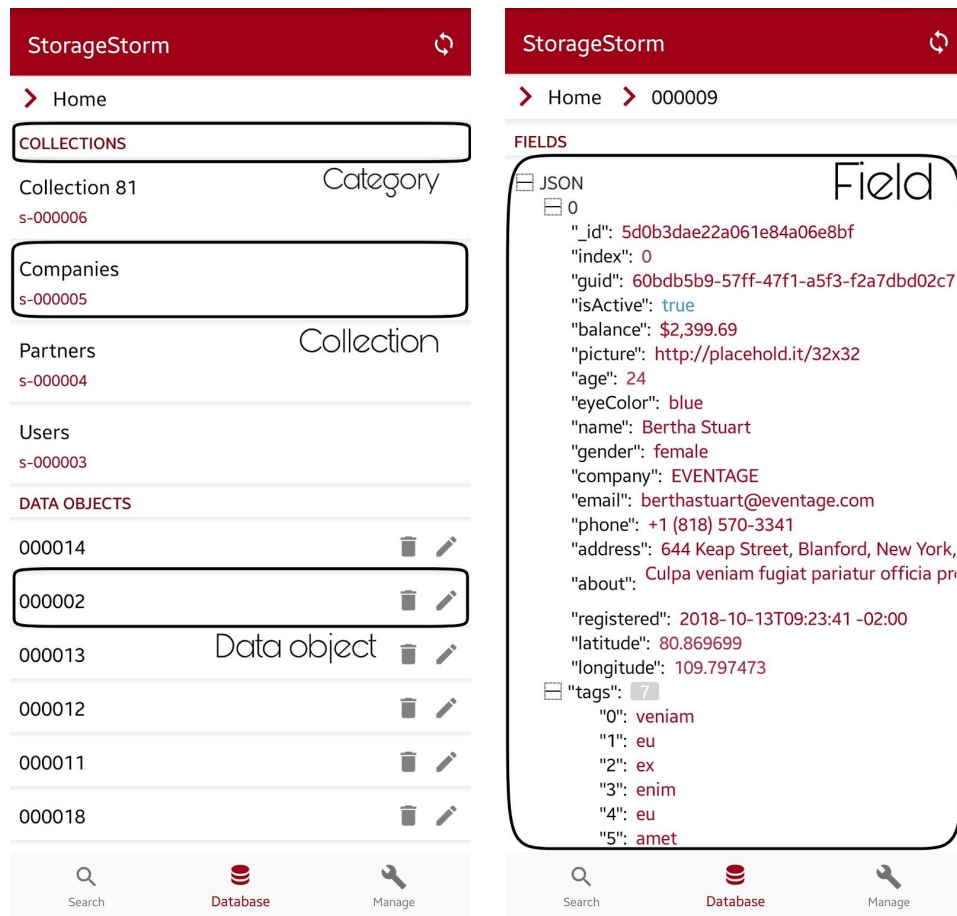
The content [recycler view](#) is responsible for displaying the data. There are four types of possible views:

Category - This is either a COLLECTION, DATA OBJECTS or FIELDS label.

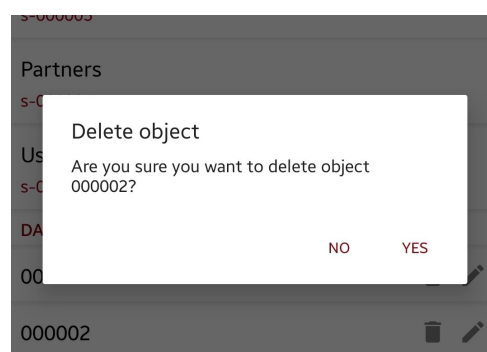
Collection - Displays a collection, it's name and id.

Data Object - Displays the id of the object.

Field - Displays the [JSON](#) fields of an object.



The data object view has 2 additional buttons for deleting and editing an object. The delete button opens a dialog that asks the user to confirm the delete process. The edit button opens the [EditObjectActivity.kt](#).



3.4. SearchFragment.kt

The search fragment gives us the possibility to search for objects keys and values. It consists of a search bar, search button and a history timeline. After each search, the search word as well as the timestamp are stored inside a [SQLite database](#).

The image displays two screenshots of the StorageStorm application's SearchFragment. Both screenshots feature a red header bar with the text 'StorageStorm' and a refresh icon. Below the header is a search bar with the placeholder text 'e.g., name' and a red 'SEARCH' button.

The right screenshot shows the 'HISTORY' section below the search bar, which contains a list of search results. Each result consists of a key and a timestamp. The keys are 'latitude', 'age:18', 'Gmbh', and 'Avenue'. The timestamps are '22.06.2019 00:29' for 'latitude' and 'age:18', and '22.06.2019 00:28' for 'Gmbh' and 'Avenue'. A 'CLEAR HISTORY' link is located to the right of the 'HISTORY' header.

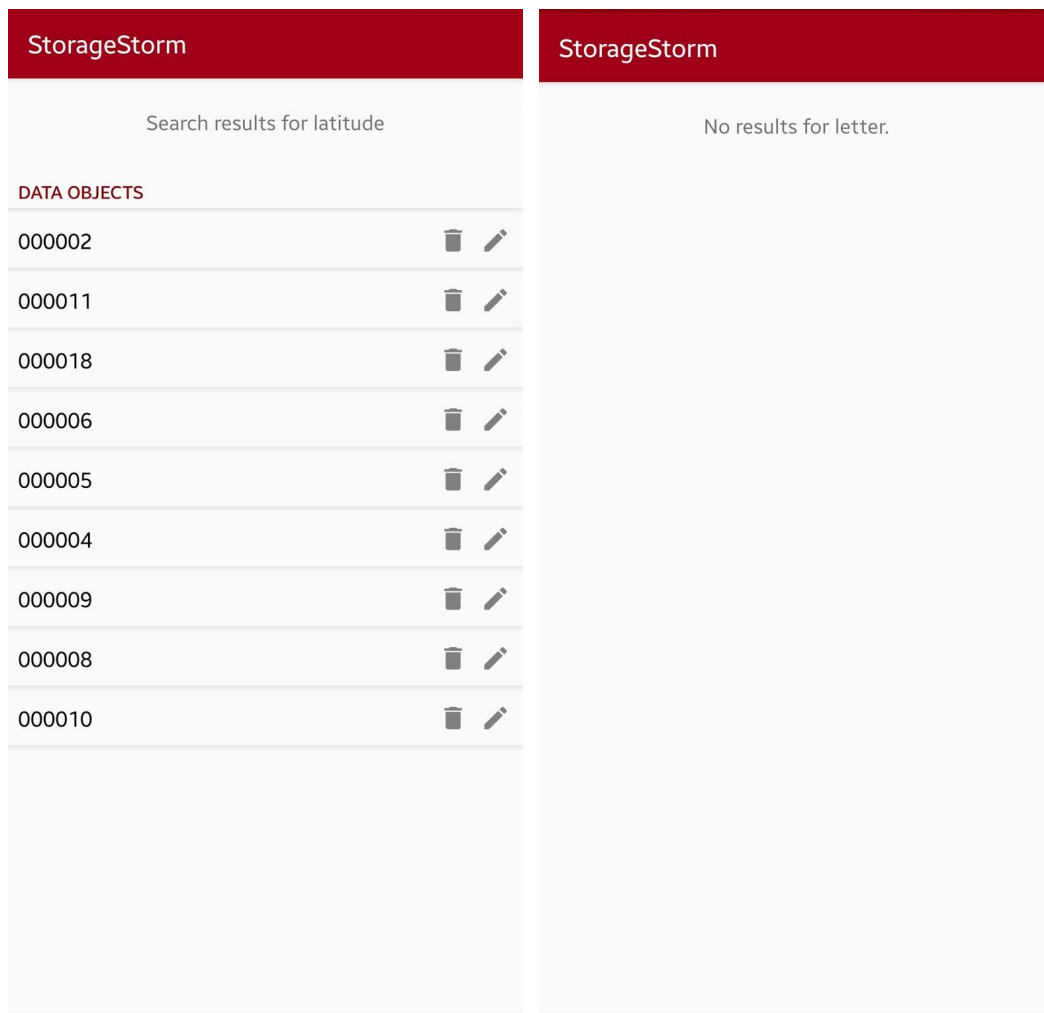
At the bottom of both screenshots is a navigation bar with three icons: a magnifying glass for 'Search', a database icon for 'Database', and a wrench for 'Manage'.

HISTORY		CLEAR HISTORY
latitude	22.06.2019 00:29	
age:18	22.06.2019 00:29	
Gmbh	22.06.2019 00:28	
Avenue	22.06.2019 00:28	

A press on the search button opens the [SearchResultActivity.kt](#), which provides a list of objects that contain the search word.

3.5. SearchResultActivity.kt

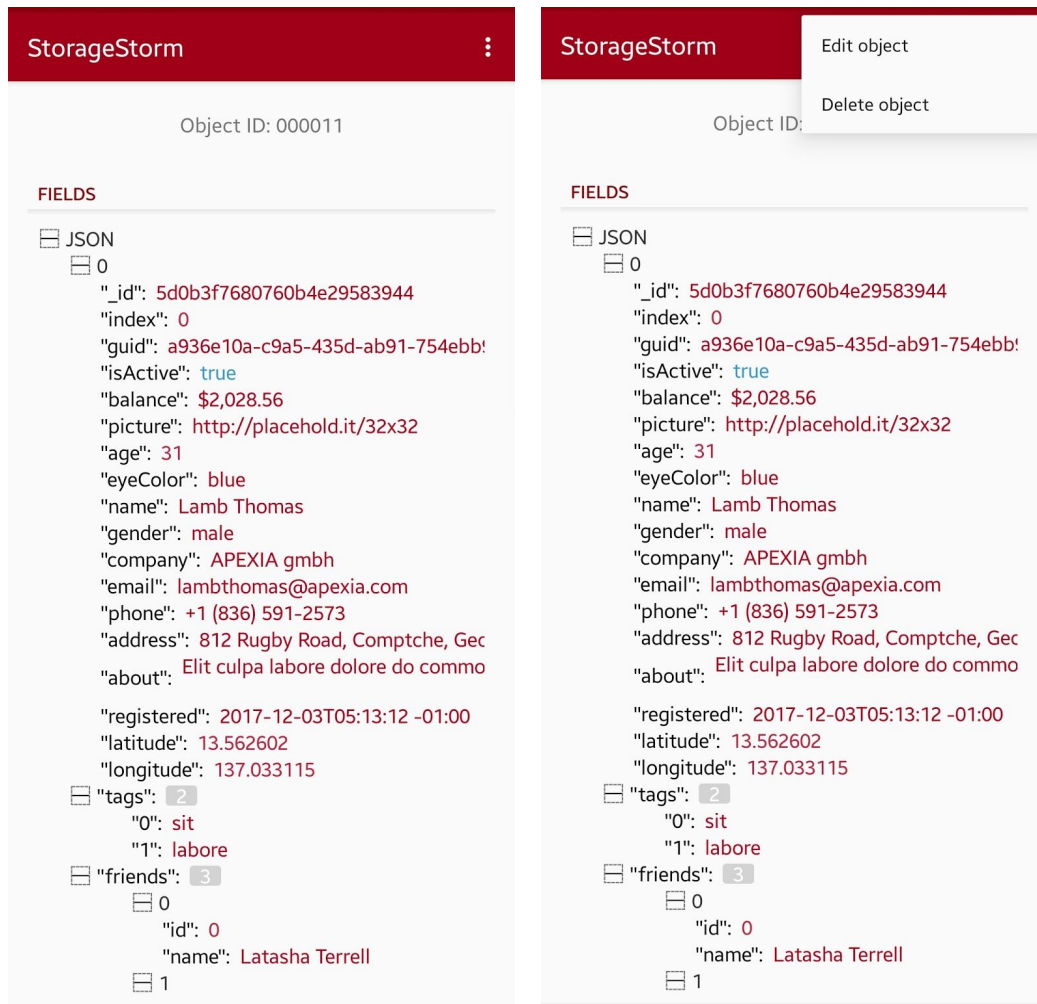
After the user entered a word, e.g. latitude, the SearchResultActivity.kt will display all data objects that contain that word as a key or a value. If there are no results, a corresponding message is displayed.



The data object view is here reused and therefore also gives the possibility to edit or delete the object. A press on the data object opens [AboutDataObjectActivity.kt](#).

3.6. AboutDataObject.kt

The AboutDataObject.kt uses the Field view to display the contents of a data object. The toolbar has two more options for deleting or editing the object.



3.7. EditObjectActivity.kt

The EditObjectActivity.kt gives the possibility to edit an object. An object with invalid [json](#) structure will not be updated and a corresponding message will be displayed.

StorageStorm

Object ID: 000014

CANCEL

SAVE

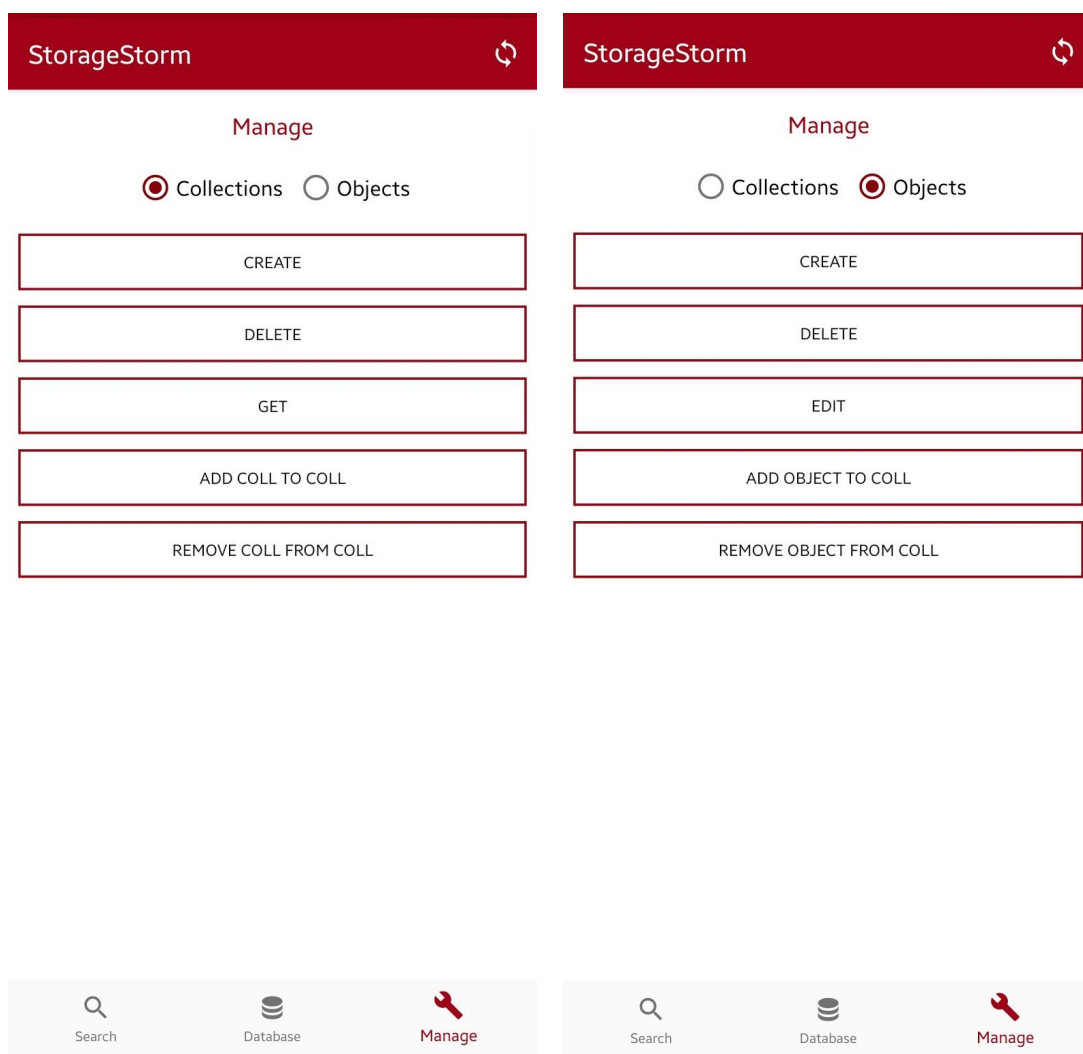
[{ "company": "LUMBREX", "phone": "+1 (872) 519-3278", "address": "194 Debevoise Street, Cressey, New York, 8867" }]

3.8. ManageFragment.kt

This fragment gives the possibility to edit/delete/create objects and collections, as well as to add/remove objects and collections to/from other collections.

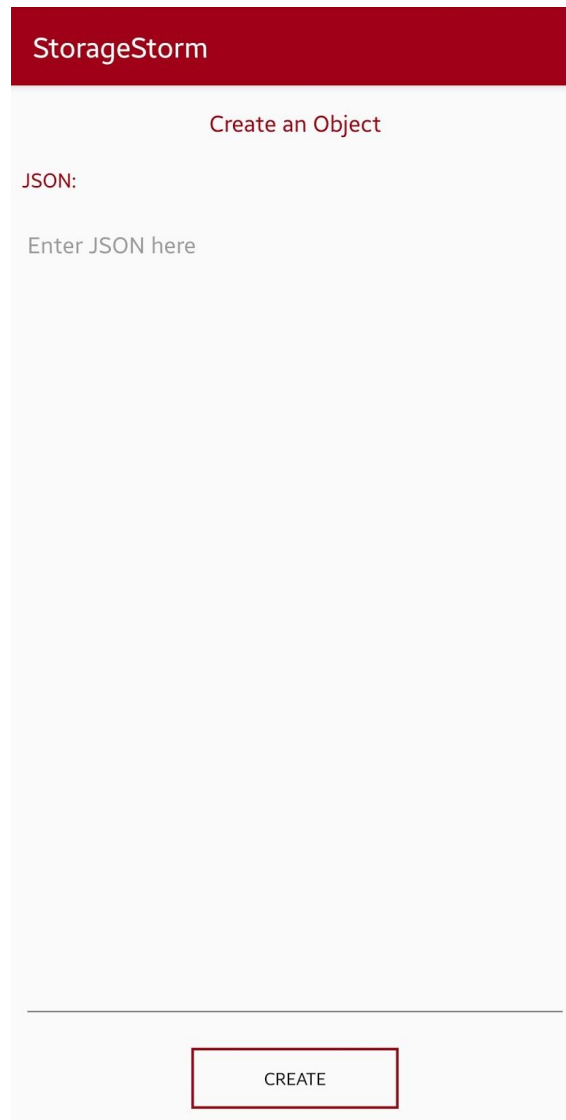
Using radio buttons the user can choose to manage either collections or objects.

Only the “create object” button opens the [CreateObjectActivity.kt](#) activity - all other buttons will open a corresponding dialog.



3.9. CreateObjectActivity.kt

This Activity takes the to be created objects [JSON](#) and creates the object. It consists of a simple text input layout and a create button.



The screenshot displays the 'StorageStorm' application interface. At the top, there is a dark red header bar with the text 'StorageStorm' in white. Below the header, the main content area has a light gray background. At the top of this area, the text 'Create an Object' is displayed in a dark red color. Below this, the label 'JSON:' is shown in dark red. Underneath the label is a large, empty text input field with a light gray border and the placeholder text 'Enter JSON here' in a very light gray. At the bottom of the screen, there is a white rectangular button with a dark red border and the text 'CREATE' in dark red. A thin horizontal line separates the input field from the button.

If the [JSON](#) is invalid, a corresponding message is displayed.

3.10. Dialogs

All dialogs are an extension of the [BaseDialog.kt](#) dialog. It consists of a title [text view](#), two [text views](#), two [text input edit text](#) views and two [buttons](#). Depending on which dialog extends it, the text of the buttons is changed accordingly (e.g., in `CreateCollectionDialog` to `Create`, in `DeleteCollectionDialog` to `Delete`). Besides that, if a dialog requires only one text input view, the other one's visibility is set to gone.

The image displays a grid of 10 dialog box mockups, organized into two columns and five rows. Each dialog box has a dark gray border and a white background.

- Row 1:**
 - Left Dialog:** Titled "Create Collection". It contains two text input fields: "Collection name" with the placeholder text "e.g. Users" and "Head ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "CREATE".
 - Right Dialog:** Titled "Delete a Collection". It contains one text input field: "Collection ID" with the value "s-000001". At the bottom are two buttons: "CANCEL" and "DELETE".
- Row 2:**
 - Left Dialog:** Titled "Remove Object from Collection". It contains two text input fields: "Collection ID" with the value "s-000001" and "Object ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "REMOVE".
 - Right Dialog:** Titled "Remove Object from Collection". It contains two text input fields: "Collection ID" with the value "s-000001" and "Object ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "REMOVE".
- Row 3:**
 - Left Dialog:** Titled "Remove Object from Collection". It contains two text input fields: "Collection ID" with the value "s-000001" and "Object ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "REMOVE".
 - Right Dialog:** Titled "Remove Object from Collection". It contains two text input fields: "Collection ID" with the value "s-000001" and "Object ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "REMOVE".
- Row 4:**
 - Left Dialog:** Titled "Remove Object from Collection". It contains two text input fields: "Collection ID" with the value "s-000001" and "Object ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "REMOVE".
 - Right Dialog:** Titled "Remove Object from Collection". It contains two text input fields: "Collection ID" with the value "s-000001" and "Object ID" with the value "000001". At the bottom are two buttons: "CANCEL" and "REMOVE".

4. Implementation details

4.1. Persistent fragment states in the [MainActivity.kt](#)

To give fragments the functionality of not losing the state, we need to create only one instance of each fragment.

```
private val databaseFragment = DatabaseFragment.newInstance()
private val searchFragment = SearchFragment.newInstance()
private val manageFragment = ManageFragment.newInstance()
var active: Fragment = databaseFragment
```

Using [supportFragmentManager](#) we do not replace the fragment view when a fragment is selected, but rather just hide the currently selected fragment and show the one that is selected.

To show the [databaseFragment](#) as the first fragment on the app start, we will hide other two and leave the databaseFragment visible.

```
supportFragmentManager.beginTransaction()
    .add(R.id.fragmentContainer, searchFragment, "searchFragment")
    .hide(searchFragment)
    .commit()
supportFragmentManager.beginTransaction()
    .add(R.id.fragmentContainer, manageFragment, "manageFragment")
    .hide(manageFragment)
    .commit()
supportFragmentManager.beginTransaction()
    .add(R.id.fragmentContainer, databaseFragment, "databaseFragment")
    .commit()

active = databaseFragment
```

The [MainActivity](#) implements a `BackPressedHandler` interface. Each child fragment implements this interface, and the [MainActivity](#) forwards the action of back button press and leaves the handling to the fragments. If they return false, the [MainActivity](#) will then handle the press itself, which is to exit the app.

Using this approach we make it possible for the [DatabaseFragment](#) to step a level out when browsing through the database, as well as to remove the last item from the path list.

Other fragments will tell the [MainActivity](#) to hide them and show the DatabaseFragment when the back button is pressed.

```
interface BackpressHandler {  
    fun onBackButtonPressed(): Boolean  
}
```

To allow fragments to refresh the content (pressing the refresh button on the MainActivity's toolbar), we forward the `onOptionsItemSelected` to the currently active fragment.

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {  
    if (active.isVisible) active.onOptionsItemSelected(item)  
    return true  
}
```

Which then, for example [DatabaseFragment](#), refreshes the content with calling the `updateView()` method.

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {  
    if (item?.itemId == R.id.menu_sync) {  
        updateView()  
        return true  
    }  
    return false  
}
```

4.2. Adapters

Each [recycler view](#) has an adapter that provides a binding from an app-specific data set (in our case a list) to views that are displayed within a recycler view.

We implement a DatabaseContentAdapter which is reused in the [DatabaseFragment](#) and other activities like [AboutDataObjectActivity](#), AboutCollectionActivity and [SearchResultActivity](#).

```
class DatabaseContentAdapter(
    private val list: MutableList
```

The input list is a list of pairs, where the first item in the pair is a string which indicates the item type, and the second item is the actual object.

The `onCreateViewHolder` method renders different views according to the `viewType`, which is either a `Category`, `Collection`, `DataObject` or a `Field`.

The `getItemViewType` method returns the type of the current item in the list.

The `getItemCount` method returns the size of the list.

The `onBindViewHolder` method does the actual binding of object's data to the corresponding view. To do that, we created an inner class `ViewHolder` that has methods `bindCategory`, `bindCollection`, `bindDataObject` and `bindField` that implement the binding.

Example: `bindCategory` method

```
fun bindCollection(collection: Collection) {
    val tvCollectionTitle = itemView.findViewById<TextView>(R.id.tvCollectionTitle)
    tvCollectionTitle?.text = collection.name

    val tvCollectionID = itemView.findViewById<TextView>(R.id.tvCollectionID)
    tvCollectionID?.text = collection.id

    val textWrapper = itemView.findViewById<RelativeLayout>(R.id.textWrapper)
    textWrapper.setOnClickListener {
        collection.performAction()
    }
}
```

The same way we have an adapter for the content recycler view, we have an adapter for the path recycler view, that works in a similar way.

```
class DatabasePathAdapter(private val pathList: MutableList<Path>, private val context: Context?) :
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, position: Int): ViewHolder {
        return ViewHolder(LayoutInflater.from(context).inflate(R.layout.database_path_row, parent, false))
    }

    override fun getItemCount(): Int {
        return pathList.size
    }

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
        (holder as ViewHolder).bindPath(pathList[position])
    }

    inner class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        private var pathText = itemView.pathText as TextView

        fun bindPath(path: Path) {
            pathText.text = path.name ?: path.ID
            pathText.setOnClickListener {
                path.performAction()
            }
        }
    }
}
```

The [SearchFragment](#) also has an adapter that displays the search history.

```
class SearchHistoryAdapter(
    private val list: List<HistoryEntry>,
    private val context: Context?,
    private val onClick: (entry: String) -> Unit
) :
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, position: Int): RecyclerView.ViewHolder {
        return ViewHolder(LayoutInflater.from(context).inflate(R.layout.history_item_row, parent, false))
    }

    override fun getItemCount(): Int {
        return list.size
    }

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
        (holder as ViewHolder).bindItem(list[position])
    }

    inner class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        fun bindItem(item: HistoryEntry) {
            val tvItem = itemView.findViewById<TextView>(R.id.tvItem)
            val tvDate = itemView.findViewById<TextView>(R.id.tvDate)
            tvItem.text = item.entry
            tvDate.text = item.getDateAsString()
            itemView.setOnClickListener { onClick(item.entry) }
        }
    }
}
```


4.3. SQLite database

The history data is retrieved from the SQLite database.

```
val db = DatabaseHandler((activity as MainActivity).applicationContext)
val historyList = db.history
```

The DatabaseHandler is a class that extends the [SQLiteOpenHelper](#) class. It overrides methods for the creation and for upgrading the database.

```
override fun onCreate(db: SQLiteDatabase?) {
    val createTableQuery = "CREATE TABLE $TABLE_HISTORY ($COL_ENTRY VARCHAR, $COL_DATE INTEGER)"
    db!!.execSQL(createTableQuery)
}

override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {
    db!!.execSQL("DROP TABLE IF EXISTS $TABLE_HISTORY")
    onCreate(db)
}
```

We need three methods for the implementation of the database.

getHistory to retrieve all history data from the database.

```
val history: List<HistoryEntry>
get() {
    val list = ArrayList<HistoryEntry>()
    val selectQuery = "SELECT * FROM $TABLE_HISTORY"
    val db = this.writableDatabase
    val cursor = db.rawQuery(selectQuery, null)
    if (cursor.moveToFirst()) {
        do {
            val entry = HistoryEntry(
                cursor.getString(cursor.getColumnIndex(COL_ENTRY)),
                cursor.getLong(cursor.getColumnIndex(COL_DATE))
            )

            list.add(entry)
        } while (cursor.moveToNext())
    }
    db.close()
    return list
}
```

addEntryToHistory which is called after each user's search.

```
fun addEntryToHistory(entry: HistoryEntry) {  
    val db = this.writableDatabase  
    val values = ContentValues()  
    values.put(COL_ENTRY, entry.entry)  
    values.put(COL_DATE, entry.date)  
  
    db.insert(TABLE_HISTORY, null, values)  
    db.close()  
}
```

clearHistory to clear all data from the database.

```
fun clearHistory() {  
    this.writableDatabase.execSQL("DELETE FROM $TABLE_HISTORY")  
}
```


4.4. BaseDialog.kt

This class is an open class that is extended by all other dialogs. It provides all the elements a dialog needs to have. Each dialog will then adapt the view as needed (change button texts, hide the second [TextInputEditText](#) if only one is needed etc.).

```
open class BaseDialogActivity : AppCompatActivity() {

    private lateinit var tvCancel: TextView
    private lateinit var tvSave: TextView
    private lateinit var tvTitle: TextView
    private lateinit var tvFirstText: TextView
    private lateinit var tvSecondText: TextView
    private lateinit var firstInput: TextInputEditText
    private lateinit var secondInput: TextInputEditText
    lateinit var sharedPref: SharedPreferences

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

        tvCancel = findViewById(R.id.tvCancel)
        tvSave = findViewById(R.id.tvSave)
        tvTitle = findViewById(R.id.tvTitle)
        tvFirstText = findViewById(R.id.tvFirstText)
        tvSecondText = findViewById(R.id.tvSecondText)
        firstInput = findViewById(R.id.firstInput)
        secondInput = findViewById(R.id.secondInput)

        sharedPref = getPreferences(Context.MODE_PRIVATE)
    }
}
```

The variable `sharedPref` is used as a persistent data storage for dialog inputs. We used [SharedPreferences](#) instead of a SQLite database because it is easier to use for single entries and therefore we do not need to create database tables.

We use `sharedPref` to store last user input for each dialog. If user never entered some value before, the default entry starts from s-000001 for collection IDs and 000000 for object IDs.

Example usage in DeleteObjectDialog.

Retrieving data from [SharedPreferences](#):

```
firstInput.setText(sharedPref.getString(Constants.PREF_DELETE_OBJ_OBJ_ID, "000001"))
```

Storing data to [SharedPreferences](#):

```
if (success) {  
    Toast.makeText(it, "$objectID deleted!", Toast.LENGTH_SHORT).show()  
    sharedPref.edit()?.run {  
        putString(Constants.PREF_DELETE_OBJ_OBJ_ID, objectID)  
        apply()  
    }  
    finish()  
}
```

4.5. AsyncTask

Since asynchronous tasks are not allowed to run in the UI thread, the [AsyncTask](#) class is used to perform background operations and publish results on the UI thread. It should ideally be used for short operations.

We use the [Anko](#) library to perform AsyncTask operations in an easy and elegant way. Rather than creating an inner class that extends the [AsyncTask](#) class each time, [Anko](#) requires just calling `doAsync { }` and `uiThread { }` methods.

We use the `doAsync` method to call [Ikarus API](#) methods and get/store data to the [Ikarus Database Engine](#).

This is an example of using [Anko](#) functionality inside the `RemoveObjectFromCollectionDialog`:

```
private fun executeRemove(collID: String, objID: String) {
    if (objID.contains("s-")) {
        Toast.makeText(this, "Please enter valid Object ID", Toast.LENGTH_SHORT).show()
        return
    }
    doAsync {
        try {
            val success = IkarusApi(Constants.UTILITIES_SERVER_URL).removeColl(collID, objID)
            uiThread {
                if (success) {
                    Toast.makeText(it, "$objID deleted from $collID!", Toast.LENGTH_SHORT).show()
                    sharedPref.edit()?.run {
                        putString(Constants.PREF_REMOVE_OBJ_FROM_COLL_COLL_ID, collID)
                        putString(Constants.PREF_REMOVE_OBJ_FROM_COLL_OBJ_ID, objID)
                        apply()
                    }
                    finish()
                } else {
                    Toast.makeText(it, "Please enter valid Collection/Object ID", Toast.LENGTH_SHORT).show()
                }
            }
        } catch (exception: IOException) {
            uiThread {
                Toast.makeText(it, "Network error!", Toast.LENGTH_SHORT).show()
            }
        } catch (exception: NullPointerException) {
            uiThread {
                Toast.makeText(it, "Please enter valid Collection/Object ID", Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

4.6. Result messages

We use the [Toast](#) class to display proper messages across the whole application. It could be either

A success message:

```
uiThread {  
    if (success) {  
        Toast.makeText(it, "$objID deleted from $collID!", Toast.LENGTH_SHORT).show()  
    }  
}
```

An error message:

```
uiThread {  
    Toast.makeText(it, "Network error!", Toast.LENGTH_SHORT).show()  
}
```

Or a wrong user input message:

```
uiThread {  
    Toast.makeText(it, "Please enter valid Collection/Object ID", Toast.LENGTH_SHORT).show()  
}
```

Note that the Toast requires a valid [Context](#) parameter, which is “it” provided from the [Anko's](#) uiThread method.