REPUBLIC OF ALBANIA
UNIVERSITY OF TIRANA
FACULTY OF NATURAL SCIENCES

Master of Science in Informatics

Project

My Crypto App

Mobile Programming

Abstract

With the advent of new mobile technologies, the mobile application industry is advancing rapidly. consisting of several operating systems such as Symbian OS, iOS, Blackberry, etc., Android OS is known as the most widely used, most popular and easiest platform for users. This open source linux kernel based operating system offers a high degree of flexibility due to its personalization features that make it a dominant mobile operating system.
Android applications are programmed in Java language. Google android SDK bring a unique range of software that provides developers with an easy platform to develop android applications. Furthermore, developers can use existing Java IDEs which provides flexibility for developers. Java libraries are dominant in the process of developing third-party applications. Cross-platform approaches ensure that developers do not have to develop platform-dependent applications. With the help of these approaches, an application can be deployed on several platforms without the need for coding changes. However, android is more prone to security vulnerabilities which most users do not take into account.

Content

1. Introduction

In this paper we will discuss building and developing a mobile app for android using all the development trends of Android. The idea and theme of the application we are going to build is related to my area of interest such as -*blockchain* and cryptocurrency(cryptocurrency). I decided to create an application that will contain your "portfolio" of cryptocurrencies and let you know how much they are worth in real value, so convert them into money (fiat money). The important thing for users is that this app will ensure 100% reliability. It will not require any login / registration process. The application will not collect the included data and sensitive data of users by sending them to the server. Sharing information online about money or cryptocurrencies owned requires a high degree of reliability. Data provided by users regarding cryptocurrency investments will only be stored within a local database stored within an Android device. However, to know the value of the portfolio converted to real or physical money, the app will use the internet to get the latest conversion rates.
So, for training purposes or to increase personal and professional skills in programming, the idea of the application is challenging. Technically it is the processing of different ways to work with data. This is one of the most important skills to know to build modern mobile applications. The topic of money for people is so sensitive. To provide even more trust, we will be developing this open source application by making a series of blog posts and making the project code available.

2. What will we use? - Required environment

First, to create this app, we need to know about different cryptocurrency prices at the moment. This data will be provided by the internet as they are relative, constantly changing.

Data API:
CoinMarketCap- one of the most popular websites to get an overview of the cryptocurrency market. This website provides a free API that anyone can use and is perfectly suited to us as a data service provider.

Now, we have a list of the most important trending things in the Android world that fit this project and should be used in it.

Programming language:
Kotlin- an official language on Android. Kotlin is an expressive, concise and powerful language. Best of all, it is interactive with existing Android languages and timed. The introduction of this new language was one of the hottest topics in 2017 for Android.

Integrated Development Environment (IDE):
Android Studio- Official IDE for Android. It provides the fastest tools for building applications on any type of Android device. There are no better alternatives for application development. It is our main choice for an application development IDE.

Project Build Management System:
Gradl- is an advanced general purpose building management system based in Groovy and Kotlin. It supports automatic download and configuration of dependencies or other libraries. It is a recommended system built by Google. It is well integrated within Android Studio, so we will use it.

Architecture:
Android Architecture Components - a collection of libraries that help us create powerful, testable and maintainable applications.
Model - View - ViewModel (MVVM)- is an architectural model. The concept is to separate the data presentation logic from the business logic by moving it to a separate class for a clearer distinction. The Android team is pushing this model as the default choice. It is also an alternative to the MVC and popular MVP models.

Data Persistance:
SQLite Database- is an open source SQL database that stores continuous data in a text file on a device. Android comes with the implementation of the integrated SQLite database. SQLite supports all relational database features.
Shared Preferences- an API from the Android SDK to save and recover application preferences. SharedPreferences are simply sets of constantly stored data values. This allows us to store and retrieve data in the form of value-key pairs.

Libraries:
Android JetPack Components:
AppCompat - is a set of support libraries which can be used to make applications that have been developed with newer versions work with older versions.
Android KTX- a set of Kotlin extensions for Android application development. The goal of Android KTX is to make Android development with Kotlin more concise, enjoyable and idiomatic by utilizing language features such as layout functions / properties, named settings and parameter defaults.

Data Biding - is a support library that allows us to link the UI components in our layouts to the data sources in our application using a declarative format instead of a software way.

Life Cycles - for the management of our activity and the life cycles of the fragments.

Live Data - is an observable data storage class which was created to help solve common Android lifecycle challenges and make applications more usable and testable.

containing - provides an abstraction layer on SQLite to allow easy access to the database while utilizing the full power of SQLite.

View Model- designed to store and manage user interface-related data in a life-cycle-conscious manner. The ViewModel class allows data to survive data configuration changes such as screen rotations.

Others:
Constraint Layout- for building flexible and efficient layouts. The Layout editor uses constraints to determine the position of a UI element within the layout. A constraint represents a connection or extension to another view, the initial presentation.
Recycler View - a flexible and efficient version of ListView.

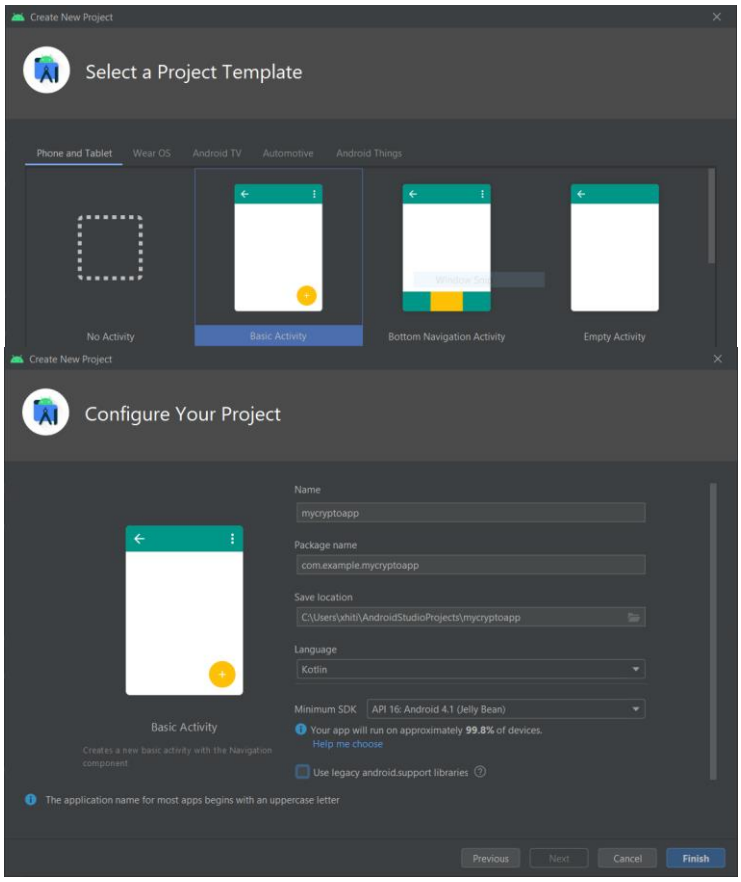
Third parties:
Dagger 2 - this is a completely static dependency, the framework, for both Java and Android.

Retrofit 2- an open source HTTP client for Android and Java. With Retrofit, we can compose HTTP connection easily through a simple, expressive interface just like an API document.

OkHttp - a modern, fast, open source HTTP client that supports HTTP and SPDY.
Gson - an open source Java library to serialize and deserialize Java objects to and from JSON.

3. Project configuration

We will create this project from scratch. So we will start Android Studio, creating a new project (in the project), named "My Crypto App" and select "Basic Activity". At this point, there is nothing special to discuss. Our goal is to make a clean start and avoid any complexity in our minds by adding additional features (e.g., instant app support). We can add everything later if we want, during the development process.



4. SOLID PRINCIPLES

Software is always in a state of change. Any change can have a negative impact on an entire project. So the essential thing is to prevent the damage that can be done while implementing all the new changes. With the "My Crypto App" application we will create new code snippets step by step. We want our project to be quality, but to achieve it, we must first understand the basic principles of creating modern software. They are called SOLID principles.

SOLID is an acronym that helps define the five basic principles of object-oriented programming:

1. **S** - **S**ingle Responsibility Principle
2. **O** - **O**pen-Closed Principle
3. **L** - **L**iskov Substitution Principle
4. **The** - **The**nterface Segregation Principle
5. **D** - **D**ependency Inversion Principle

4.1 Single Responsibility Principle

... A class should have only a single responsibility.

Each class or module should be responsible for only one part of the functionality provided by the application. So basically when he is responsible for only one thing, there should be only one main reason to change it. If our class or module does more than one thing, then we need to divide the functions into separate ones.
To better understand this principle, I would take as an example a knife of the Swiss Army. This knife is well known for its unique feature that can be used for multiple functions as it has in addition to its main blade various other tools integrated inside, such as screwdrivers, a can and many others. The natural question here for you may arise why am I suggesting this knife as an example of sole functionality? But just think about it for a moment, the other main feature of this knife is mobility while being pocket-sized. So even though it offers several different functions, it still fits its main purpose of being small enough to take it with you comfortably. The same rules go with programming. When we create the class or module, it should have a key global purpose, but at the same time we can not overload when we try to simplify everything by sharing functionality. Therefore balance must be maintained.

A classic example might be the method often used onBindViewHolder when building the RecyclerView widget adapter.

```
class MusicVinylRecordRecyclerViewAdapter(
	private val vinyls: List<VinylRecord>,
	private val itemLayout: Int
) : RecyclerView.Adapter<MusicVinylRecordRecyclerViewAdapter.ViewHolder> () {

	override fun onBindViewHolder(holder: ViewHolder, position: Int) {
		val vinyl = vinyls[position]
		holder.itemView.tag = vinyl
		holder.title!!. text = vinyl.title
		holder.author!!. text = vinyl.author
		holder.releaseYear!!. text = vinyl.releaseYear
		holder.country!!. text = vinyl.country
		holder.condition!!. text = vinyl.condition
		holder.genre!!. text = convertArrayListToString(vinyl.genres)
	}
}
```

4.2 Open-Closed Principle

... Software entities should be open for extension, but closed for modification.

This principle says that when we write all the pieces of software like classes, modules, functions, we should make them open for expansion but closed for any modifications. What does it mean? Let's say we create a working class. There should be no need to adjust that class if we need to add a new functionality or make some changes. Instead, we should be able to expand that class by creating its new subclass, where we can easily add all the necessary new features. Characteristics should always be within parameters in a way that a subclass can prevail.

Let's take an example where we are going to create separate FeedbackManager class to show a different kind of personalized message to users.

```
class MainActivity: AppCompatActivity() {
	lateinit hang feedbackManager: FeedbackManager

	override fun onCreate(savedInstanceState: Bundle?) {
		super.onCreate(savedInstanceState)
		setContentView(R.layout.activity_main)
		feedbackManager =FeedbackManager(findViewById(android.R.id.content));
	}
	override fun onStart() {
		super.onStart()
		feedbackManager.showSpecialMessage(CustomToast())
	}
}

class FeedbackManager(hang view: View) {
	fun showSpecialMessage(message: Message) {
		message.showMessage(view)
	}
}
class CustomToast: Message {
	hang welcomeText: String = "Toast message!"
	hang welcomeDuration: Int Toast.LENGTH_SHORT
	override fun showMessage(view: View) {
		Toast.makeText(view.context, welcomeText, welcomeDuration).show()
	}
```

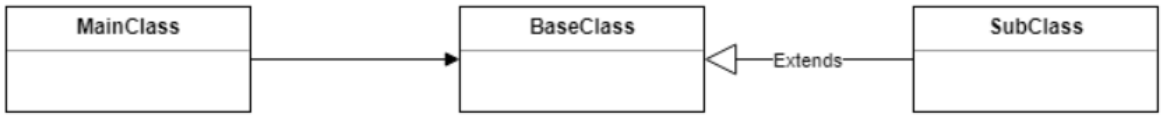
```
}

class CustomSnackbar: Message {
	hang goodbyeText: String = "Snackbar message .."
	hang goodbyeDuration: Int Toast.LENGTH_LONG
	override fun showMessage(view: View) {
		Snackbar.make(view, goodbyeText, goodbyeDuration).show()
	}
}
```

4.3 Liskov Substition Principle

Jects Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

This principle is named after Barbara Liskov – a well-known computer scientist. The general idea of this principle is that objects should be replaced by instances of their subtypes without changing program behavior. Let's say that in our application we have MainClass in which its functionalities depend on BaseClass, which expands to a SubClass. In short, to follow this principle, the code we have in MainClass and the application in general should work smoothly when we decide to switch from BaseClass to SubClass.



To better understand this principle we have a classic example, easy to understand with the legacy between the Square and Rectangle classes.

```
class MainActivity: AppCompatActivity() {
	override fun onCreate(savedInstanceState: Bundle?) {
		super.onCreate(savedInstanceState)
		setContentView(R.layout.activity_main)
		val rectangleFirst: Shape = Rectangle(2,3)
		val rectangleSecond: Shape = Square(3)
		textViewRectangleFirst.text = rectangleFirst.area().()
		textViewRectangleSecond.text = rectangleSecond.area().()
	}
}
```

```
class Rectangle(hang width: Int, hang height: Int) : Shape() {
	override fun area(): Int {
		return width * height
	}
}

class Square(hang edge: Int) : Shape() {
	override fun area(): Int {
		return edge * edge
	}
}

abstract class Shape {
	abstract fun area(): Int
}
```

4.4 Interface Segregation Principle

... Many client-specific interfaces are better than one general-purpose interface.

Regardless of the name, the principle itself is quite easy to understand. He says a client should never be forced to depend on methods or implement an interface that they do not use. A class should be designed to have fewer methods and attributes. When creating an interface we should try to make it not too large, but instead divide it into smaller interfaces so that the interface client knows only about the methods that are relevant.

4.5 Dependency Inversion Principle

… One should "depend upon abstractions, [not] concretions."

The last principle says that high level modules should not depend on low level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. The main idea of the principle is not to have direct dependencies between modules and classes and to try to make them dependent on abstractions (e.g. interfaces). To simplify it even more if we use a class within another class, this class will depend on the injected class, so this violates the idea of principle and we should not do it.
To summarize what we said above about SOLID If we think about all these principles, we can see that they are complementary to each other. Following the SOLID principles will give us many benefits, they will make our application reusable, maintainable, scalable, testable. Of

course it is not always possible to follow all of these principles correctly as it all depends on individual situations when writing code.

6. Mockups & Layouts (UI / UX & XML)

How to start building a new application? So in this section we will introduce the "My Crypto App" application mockups, to discuss how to create them. Also we will build all the UI layouts more or less as it will become our strong foundation by clearly showing us what to code. Finally we will localize our application in different languages and learn how to handle what is written from right to left.

6.1 Mockups

For all models of project mockups we are using Balsamiq Mockups for Desktop app - fast, effective and very easy to use software. Recommended by the best Android mobile app developers with confidence for creating Android apps.
Another important thing to talk about is the visual design of the app. At the time of writing Material Design is the stock visual design recommended by Google for all Android apps. For the My Crypto App we will definitely use the Material Model based on best practices to determine in detailed online instructions.

6.2 Layouts

Now that we have our application frames ready, it's time to build a realistic interface. The Android team is constantly improving the way we can build our interface by adding new features to XML layouts. Using XML would bring less code and the project would look cleaner, more understandable to other developers, and easier to maintain.

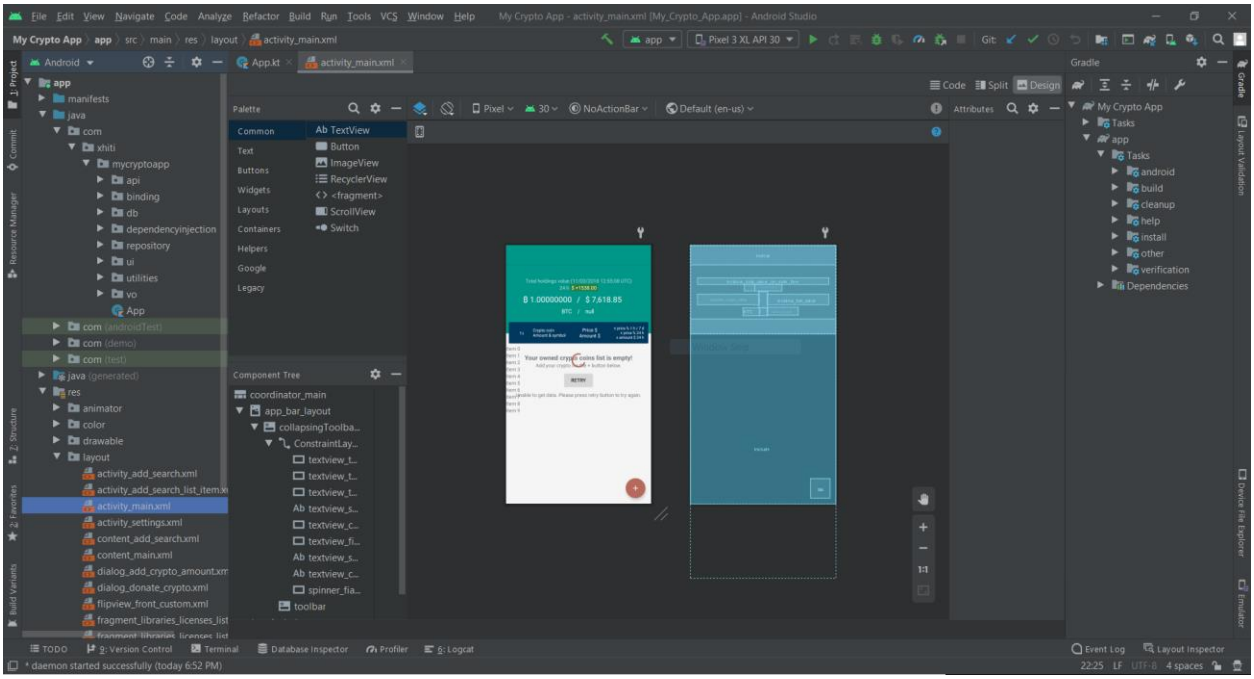
For this application we will use various components that are so common in all modern Android applications:

CoordinatorLayout - a super powerful FrameLayout, which as a main effect is its ability to coordinate animations and image transitions within it.
AppBarLayout - a vertical LinearLayout which implements many of the features of the application bar concept for material designs, namely gestures of movement.

Toolbar - a generalization of action bars for use within application layouts.

FloatingActionButton- a circular button that triggers the main action in the application interface.

For the home page, we will use the combination of all these components to create a nice UI / UX desing, where the user can expand the page to find the total value of the entire cryptocurrency portfolio, to check value change over the last 24 hours but not only.



Of course in addition to these mentioned components there are other important components, which at the moment we have not used in the My Crypto App or are simply not listed here. It is very important to own the whole package of ingredients to have an overview of which one to use in a particular case. As mobile trends change from time to time, the package will expand as well.

Recycler View

The main page of the application should consist of a list of cryptocurrencies that the user personally holds. RecyclerView is best suited for this purpose - a mini application to display a moving list of elements based on large data sets (or frequently changing data). Basically because of its advantages, RecyclerView nowadays became the main recommended ingredient for creating any page in list form. It is a more advanced and flexible version of the simplest ListView component.

Here are the steps to implement RecyclerView:

1. Adding a RecyclerView component.

fragment_main_list.xml

```
<android.support.v7.widget.RecyclerView
        android: id ="@ + id / recyclerview_fragment_main_list"
        android: layout_width ="match_parent"
        android: layout_height ="wrap_content"
        android: background ="@ color / colorForMainListBackground"
        android: clipToPadding ="false"
        android: paddingBottom ="72dp"
        android: paddingTop ="5dp"
        android: scrollbarStyle ="outsideOverlay"
        android: scrollbars ="vertical" />
...
```

Our main activity layout is activity_main.xml. This layout includes the content_main.xml layout which is a snippet. So here we need to add the RecyclerView component.

2. Create the RecyclerView line layout.

fragment_main_list_item.xml

```
        <android.support.v7.widget.CardView xmlns: android
        ="http://schemas.android.com/apk/res/android"
        xmlns: app ="http://schemas.android.com/apk/res-auto"
        xmlns: tools ="http://schemas.android.com/tools"
        android: layout_width ="match_parent"
        android: layout_height ="wrap_content"
        android: layout_marginBottom ="@ dimen / main_cardview_list_item_outer_top_bottom_margin"
        android: layout_marginEnd ="@ dimen / main_cardview_list_item_outer_start_end_margin"
        android: layout_marginStart ="@ dimen / main_cardview_list_item_outer_start_end_margin"
        android: layout_marginTop ="@ dimen / main_cardview_list_item_outer_top_bottom_margin"
        android: foreground ="? android: attr / selectableItemBackground"
        android: clickable ="true"
        android: focusable ="true"
        app: cardBackgroundColor ="@ color / colorForMainListItemBackground">

        <android.support.constraint.ConstraintLayout
                android: layout_width ="match_parent"
                android: layout_height ="wrap_content"
                android: padding ="@ dimen /
                main_cardview_list_item_inner_margin">
```

```
        ...
    <android.support.v7.widget.AppCompatTextView
        android: id ="@ + id / item_name"
        style ="@ style / MainListItemPrimeText"
        android: layout_marginEnd ="@ dimen /
        main_cardview_list_item_text_between_margin"
        android: layout_marginStart ="@ dimen /
        main_cardview_list_item_inner_margin"
        app: layout_constraintBottom_toTopOf ="@ + id /
        item_amount_symbol"
        app: layout_constraintEnd_toStartOf ="@ + id /
        guideline1_percent"
        app: layout_constraintStart_toEndOf ="@ + id / item_image_icon"
        app: layout_constraintTop_toTopOf ="parent"
        app: layout_constraintVertical_chainStyle ="spread"
        tools: text ="@ string / sample_text_item_name" />
        ...
    </android.support.constraint.ConstraintLayout>
</android.support.v7.widget.CardView>
```

For the initial initialization purpose, we will only set the item name for each row and our simplified layout

3. Create the DataAdapter class.

MainRecyclerViewAdapter.kt

```
class MainRecyclerViewAdapter(
    val dataList: ArrayList<String>
) : RecyclerView.Adapter<MainRecyclerViewAdapter.CustomViewHolder> () {
    override fun onCreateViewHolder(
        parent: ViewGroup, viewType: Int): CustomViewHolder {
            val v =
            LayoutInflater.Frome(parent.context).inflata(R.layout.frag
            ment_main_list_item, parent, false)
            return CustomViewHolder(v)
        }

    override fun onBindViewHolder(holder: CustomViewHolder, position: Int) {
        holder.txtName? .text = dataList[position]
    }
    override fun getItemCount(): Int {
```

```
            return dataList.size
        }

        inner class CustomViewHolder(
            itemView: View
        ) : RecyclerView.ViewHolder(itemView) {
            val txtName = itemView.findViewById<TextView> (R.id.item_name)
        }
}
```

Our adapter for now will accept data in string format, but later we will have to create separate class data model as we will have to pass more information than just one string.

4. Connect RecyclerView with custom adapter

MainActivityListFragment.kt

```
class MainActivityListFragment: excerpt() {
    private lateinit hang recyclerView: RecyclerView
    private lateinit hang recyclerAdapter: MainRecyclerViewAdapter
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup ?, savedInstanceState: Bundle?
    ): View? {
        val v: View = inflater.inflata(
            R.layout.fragment_main_list, container, false
        )
        recyclerView = v.findViewById(
            R.id.recyclerview_fragment_main_list
        )
        return v
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        setupList()
    }


    private fun setupList() {
        val date = ArrayList<String> ()
```

```
            date.add("Bitcoin")
            date.add("Etherium")
            date.add("Ripple")
            ...
            recyclerView.layoutManager = LinearLayoutManager(activity)
            recyclerAdapter =MainRecyclerViewAdapter(date)
            recyclerView.adapter = recyclerAdapter
        }
}
```

ListView

In this project ListView will be used for the site where we can add the cryptocurrency we own. ListView is the predecessor of RecyclerView and due to its shortcomings in recent years it was no longer used. The list we are going to create is so simple that the technical limitations of ListView will not cause us any problems.

Let's follow the very similar steps needed to implement ListView:

1. We add a ListView component.

`content_add_search.xml`

```
<ListView
        android: id ="@ + id / listview_activity_add_search"
        android: layout_width ="match_parent"
        android: layout_height ="wrap_content"
        android: scrollbars ="vertical" />
...
```

The first thing we need to do is add a ListView to AddSearchActivity. Open the activity presentation file activity_add_search.xml and you will see that it contains the content_add_search.xml. There we will add the ListView component.

2. Create the ListView row layout.

`activity_add_search_list_item.xml`

```
<android.support.constraint.ConstraintLayout
        xmlns: android ="http://schemas.android.com/apk/res/android"
        xmlns: app ="http://schemas.android.com/apk/res-auto"
```

```
        xmlns: tools ="http://schemas.android.com/tools"
        android: layout_width ="match_parent"
        android: layout_height ="match_parent"
        android: padding ="@ dimen / add_search_list_item_inner_margin">
        ...
        <android.support.v7.widget.AppCompatTextView
        android: id ="@ + id / item_name"
        style ="@ style / AddSearchListItemPrimeText"
        android: layout_marginEnd ="@ dimen /
        add_search_list_item_text_between_margin_2x"
        android: layout_marginStart ="@ dimen /
        add_search_list_item_text_between_margin"
        app: layout_constraintBottom_toBottomOf ="parent"
        app: layout_constraintEnd_toStartOf ="@ + id / item_symbol"
        app: layout_constraintStart_toEndOf ="@ + id / item_image_icon"
        app: layout_constraintTop_toTopOf ="parent"
        tools: text ="@ string / sample_text_item_name" />
        ...
</android.support.constraint.ConstraintLayout>
```

In the same way as before in the case of RecyclerView, we will only set the item name for each row. The layout of the lines given here is simplified.

3. Create the DataAdapter class.

AddSearchListAdapter.kt

```
class AddSearchListAdapter(
        context: Context,
        private val dataSource: ArrayList<String>
) : BaseAdapter() {
        private val inflater: LayoutInflater =
        context.getSystemService(Context.LAYOUT_INFLATER_SERVICE) neither
        LayoutInflater

        override fun getView(
                position: Int,
                convertView: View ?, parent: ViewGroup?
        ): View {
                val view: View
                val holder: CustomViewHolder
```

```
            f (convertView ==null) {
            view = inflater.inflata(
                  R.layout.activity_add_search_list_item, parent, false
            )
            holder = CustomViewHolder()
            holder.nameTextView = view.findViewById(R.id.item_name)
            view.tag = holder
      } else {
            view = convertView
            holder = convertView.tag neither CustomViewHolder
      }
            val nameTextView = holder.nameTextView
            nameTextView.text = getItem(position) neither String
            return view
      }
      override fun getItem(position: Int): Any {
            return dataSource[position]
      }
      override fun getItemId(position: Int): Long {
            return position.toLong();
      }
      override fun getCount(): Int {
            return dataSource.size
      }

      inner class CustomViewHolder {
            lateinit hang nameTextView: AppCompatTextView
      }
}
```

Similarly to our RecyclerView our ListView adapter for now will only accept string data to get the item name and display visually on the screen for us. Later we will use the special class data model. As for this part, we want to build a very simple list showing only cryptocurrency titles, instead of creating our own custom adapter, we can use a default ArrayAdapter.

4. Connect ListView with custom adapter.

`AddSearchActivity.kt`

```
class AddSearchActivity: AppCompatActivity() {
        private lateinit hang listView: ListView
        override fun onCreate(savedInstanceState: Bundle?) {
                super.onCreate(savedInstanceState)
                setContentView(R.layout.activity_add_search)
                setSupportActionBar(toolbar2)
                supportActionBar ?.setDisplayHomeAsUpEnabled(true)
                val date = ArrayList<String> ()
                date.add("Bitcoin")
                date.add("Etherium")
                date.add("Ripple")
                date.add(Bitcoin Cash)
                date.add("Litecoin")
                ...
                val adapter = AddSearchListAdapter(this, date)
                listView =findViewById(R.id.listview_activity_add_search)
                listView.adapter = adapter
        }
}
```

SearchView

On the same page with all the cryptocurrencies listed in Simple ListView, we also need to add SearchView. Search will be useful functionality for any user who wants to find specific cryptocurrencies by just searching its name. For this part we will not build the functionality fully but will simply implement its visual part.

Follow these steps to add SearchView to your project:

1. We declare the search configuration in XML

`searchable.xml`

```
<searchable
        xmlns: android ="http://schemas.android.com/apk/res/android"
        android: label ="@ string / app_name"
        android: hint ="@ string / search_hint">
</searchable>
```

The configuration search file must be added to the directory named xml. Here we can specify attributes for the SearchView component which determines how it behaves.

2. We create a new activity which will become our research activity.

We will create a new empty activity that implements AppCompatActivity () and name it AddSearchActivity.

3. We specify the activity created in the Android manifest file to be searched.

AndroidManifest.xml

```
<manifesto     xmlns: android ="http://schemas.android.com/apk/res/android"
               package ="com.xhiti.mycryptoapp">
       <application
              android: icon ="@ mipmap / ic_launcher"
              android: label ="@ string / app_name"
              android: roundIcon ="@ mipmap / ic_launcher_round"
              android: theme ="@ style / AppTheme">
              ...
              <activity
                     android: name =".AddSearchList.AddSearchActivity"
                     android: parentActivityName =".MainList.MainActivity"
                     android: theme ="@ style / AppTheme.NoActionBar">
                     <intent-filter>
                            <action android: name ="android.intent.action.SEARCH" />
                     </intent-filter>
                     <metadata      android: name ="android.app.searchable"
                     android: resource ="@ xml / searchable" />
              </activity>
       ...
       </application>
</manifesto>
```

We will allow the Android system to handle the search process, so we add search action and metadata to the AddSearchActivity activity element.

Metadata has a name and source which is associated with the search configuration file located in the res / xml directory.

4. We create the search menu.

menu_search.xml

```
<menu  xmlns: android ="http://schemas.android.com/apk/res/android"
       xmlns: app ="http://schemas.android.com/apk/res-auto">
       <item  android: id ="@ + id / search" android: icon ="@ drawable /
       ic_search"
              android: title ="@ string / action_search"
              app: actionViewClass ="android.support.v7.widget.SearchView"
              app: showAsAction ="ifRoom | collapseActionView" />
</menu>
```

Inside the res / menu file we will create a menu resource file.

5. We add the search menu to the activity.

AddSearchActivity.kt

```
class AddSearchActivity: AppCompatActivity() {
       override fun onCreateOptionsMenu(menu: Menu?): Boolean {
              menuInflater.inflata(R.menu.menu_search, menu)
              val searchManager = getSystemService(
                     Context.SEARCH_SERVICE
              ) neither SearchManager
              val searchView = menu ?.findItem(
                     R.id.search
              )? .actionViewneither SearchView
              searchView.setSearchableInfo(
                     searchManager.getSearchableInfo(componentName)
              )
              searchView.maxWidth = Integer.MAX_VALUE
              return true
       }
}
```

We will add the Android search widget as an action menu.

Settings

In creating and developing a modern Android app, it is highly recommended to include the settings page and give the user the app settings. The inclusion of settings in the application gives users the "power" to control some of the functionality of the application, to be in control of how the application behaves. So we will create the settings page for the My Crypto App as well.

1. Create the XML file of preferences (settings).

We will create the XML file of the preferences screen which should be placed in the res / xml directory.

2. We create the preferences fragment

We need to create a new blank snippet - SettingsFragment, which should extend PreferenceFragment (). This snippet will create preferences from the XML source that we just created. Later, this snippet will contain all the methods needed to create XML parameters and provide settings when changing settings.

```
SettingsFragment.kt
class SettingsFragment: PreferenceFragment() {
	override fun onCreate(savedInstanceState: Bundle?) {
		super.onCreate(savedInstanceState)
		addPreferencesFromResource(R.xml.pref_main);
	}
}
```

3. We create the activity of preferences

With the settings snippet we create a new activity - AppCompatPreferenceActivity, which extends PreferenceActivity (). This class ensures compatibility on all devices and versions.

4. We create the settings activity

```
SettingsActivity.kt
class SettingsActivity: AppCompatPreferenceActivity() {
	override fun onCreate(savedInstanceState: Bundle?) {
		super.onCreate(savedInstanceState)
		setupActionBar()
		fragmentManager.beginTransaction().replace(android.R.id.content,
		SettingsFragment()).commit()
	}
```

```
        private fun setupActionBar() {
                supportActionBar ?.setDisplayHomeAsUpEnabled(true)
        }
        override fun onMenuItemSelected(featureId:Int, item: MenuItem): Boolean{
                val id = item.the facility,
                f (id == android.R.id.home) {
                        f (!super.onMenuItemSelected(featureId, item)) {
                                NavUtils.navigateUpFromSameTask(this)
                        }
                        return true
                }
                return super.onMenuItemSelected(featureId, item)
        }
}
```

5. We place items and settings in the main menu

`menu_main.xml`

```
<menu  xmlns: android ="http://schemas.android.com/apk/res/android"
        xmlns: app ="http://schemas.android.com/apk/res-auto"
        xmlns: tools ="http://schemas.android.com/tools"
        tools: context ="com.xhiti.mycryptoapp.MainList.MainActivity">
        <item  android: id ="@ + id / action_settings"
                android: orderInCategory ="100"
                android: title ="@ string / action_settings"
                app: showAsAction ="never" />
</menu>
```

6. Start the activity of the created settings when the settings are selected from the menu.

`MainActivity.kt`

```
class MainActivity: AppCompatActivity() {
        override fun onCreateOptionsMenu(menu: Menu): Boolean {
                menuInflater.inflata(R.menu.menu_main, menu)
                return true
        }
        override fun onOptionsItemSelected(item: MenuItem): Boolean {
```

```
                return when (item.the facility,) {
                        R.id.action_settings -> {
                                startActivity(Intent(
                                        this@MainActivity, SettingsActivity
                                        ::class.WEEK)
                                );
                        return true
                }
                else -> super.onOptionsItemSelected(item)
        }
    }
}
```

7. We specify the activity of the settings created in the Android manifest file.

`AndroidManifest.xml`

```
<manifesto    xmlns: android ="http://schemas.android.com/apk/res/android"
              package ="com.xhiti.mycryptoapp">
        <application
              android: allowBackup ="true"
              android: icon ="@ mipmap / ic_launcher"
              android: label ="@ string / app_name"
              android: roundIcon ="@ mipmap / ic_launcher_round"
              android: supportsRtl ="true"
              android: theme ="@ style / AppTheme">
              <activity
                    android: name =".Settings.SettingsActivity"
                    android: label ="@ string / title_activity_settings"
                    android: parentActivityName =".MainList.MainActivity">
                    <metadata
                    android: name ="android.support.PARENT_ACTIVITY"
                    android: value ="com.xhiti.mycryptoapp.MainList.MainActivity" />
              </activity>
        </application>
</manifesto>
```

8. Architecture and components of architecture

The most important thing to focus on when starting to build a new application is to think about its architecture. There are plenty of architectural models for building applications. The most popular are the classic three-level architectures such as:

- MVC: Model-View-Controller
- MVP: Model-View-Presenter
- MVVM: Model-View-ViewModel

6.1 Model-View-ViewModel Architecture (MVVM)

The MVVM model is the third way of replication, which became the recommended architecture model by the Android team with the release of Android Architecture Components. This is why we will focus on learning this model above all. Also we will use exactly that for the app "My Crypto Currencies". Let's take a look at its separate layers of code:

Model - abstracts the data source. ViewModel works with Model to retrieve and store data.

View - informs ViewModel about user actions.

ViewModel - exposes important data streams for the View.

The difference compared to the MVC model is that in the MVVM ViewModel does not hold a View reference as it does with the Presenter. In MVVM, ViewModel exposes the flow of events to which different Views can be linked while in the MVC case, the Presenter directly tells View what to display. Let's take a look at the MVVM scheme:

6.2 Android architecture components

Android Architecture components are a collection of libraries that help us create and develop powerful, testable, and well-maintained applications.
It consists of the following components:

LiveData

LiveData is a data "storage" class. It follows the model of the observer. LiveData is life cycle aware which means it only updates application component monitors (activity, snippet, etc.) that are in an active life cycle state. The LiveData class returns the last value of the data. When the data changes, it returns the updated value. LiveData fits best with ViewModel.

containing

Our application must store continuous data of various cryptocurrencies used by the user. This should be stored inside the local database that is stored inside the Android device privately. For storing structured data in a private database we will use the SQLite database which is often the best and recommended choice. In order to create the SQLite database for our application, we will use Room - a persistent library created by the Android team which is basically an add-on to SQLite.

There are basically 3 main components in a room:

1. Entities - the component represents a class that holds a row of databases. For each entity, a database table is created to hold the data they represent.
2. DAO (Data Access Object) - the main component which is responsible for determining the methods that enter the database.
3. Database - the component which is a holding class that uses the notation to define the list of entities, the list of DAOs, the version of the database and serves as the main entry point for the basic link.

Let's follow these simple steps to configure Room in our app:

1. We create an entity.

```
@Entity
date class Cryptocurrency (valname: String, valrank: lot,valamount:
Double,@PrimaryKey
 valsymbol: String,valprice:
Double,valamountFiat:Double,valpricePercentChange1h:
Double,valpricePercentChange7d: Double,valpricePercentChange24h:
```

```
Double,valamountFiatChange24h: Double)
```

2. We create DAO.

```
@Dao
interface MyCryptocurrencyDao {@Query("SELECT * FROM Cryptocurrency")fun
getMyCryptocurrencyLiveDataList(): LiveData <List<Cryptocurrency>>
 @Insert
 fun insertDataToMyCryptocurrencyList(data:List<Cryptocurrency>)}
```

3. We create, configure the database and create repository

The design architectures we discussed in this section should be used as informed guidelines, but not as strict rules. In the third part we started building a new application with best practices applied. Let's summarize everything we have already managed to do:

My Crypto App and every single page has its ViewModel, which will adapt to any configuration changes and protect the user from any data loss.
The user interface of the application is of the reactive type, which means that it will be updated immediately and automatically when the data changes at the back. This is done with the help of LiveData. Our project has less code as we link to the variables in our code directly using Data Biding. Finally, our application stores user data locally within the device as the SQLite database which was conveniently created with the Room component. The application code is structured by features and the whole project architecture is MVVM - model recommended by the Android team.

9. Injection of dependency

Injection of dependencies will significantly improve the code in our project by making it more modular, flexible and testable. We will use Dagger 2 - the most popular open source dependency injection framework for Android.

What is Injection of dependencies?

To explain the injection of dependencies we must first understand what "dependency" means in relation to programming. A dependency is when one of the objects depends on the concrete implementation of another object. We can identify a dependency in our code whenever we instantiate one object within another.

What is Dagger 2?

Dagger is a completely static open source dependency for both Java and Android. The Dagger 2 is considered to be one of the most efficient injection dependencies ever built. Dagger 2 is

written in Java and the code generated by its notation processor will also be Java code. However it works with Kotlin language without any problem or modification as remember that Kotlin is fully interactive with Java. All of this means that this dependency will require more work to configure and specifically to learn it, but will provide performance enhancement with the security of compiling time.

Injection of dependency for MVVM with Kotlin.

How to configure Dagger 2 with ViewModels, Activities and Fragments?

1. To get started we need to activate Kotlin and add special Dagger 2 dependencies.

`build.gradle (Module: app)`

```
apply plugin: 'kotlin-kapt'
implementation "com.google.dagger: dagger: $ versions.dagger"
implementation "com.google.dagger: dagger-android: $ versions.dagger"
implementation "com.google.dagger: dagger-android-support: $ versions.dagger"
, ch"com.google.dagger: dagger-compiler: $ versions.dagger"
, ch"com.google.dagger: dagger-android-processor: $ versions.dagger"
```

2. We create our own App class.

`App.kt`

```
class App: Application() {
	override fun onCreate() {
		super.onCreate()
	}
}
```

3. Update the manifest file to activate the App class.

`AndroidManifest.xml`

```
<manifesto xmlns: android ="http://schemas.android.com/apk/res/android"
		package ="com.xhiti.mycryptoapp">
	<application
		android: name = ". App"
		android: allowBackup = "true"
```

```
android: icon = "@ mipmap / ic_launcher"
android: label = "@ string / app_name"
```

4. Now let's create a new package called dependencyinjection

5. We create the AppModule class module that will provide dependency throughout our application.

AppModule.kt

```
@Module(includes = [ViewModelsModule ::class])
class AppModule() {
	@Singleton
	@Provides
	fun provideContext(app: App): Context = app
	@Singleton
	@Provides
	fun provideCryptocurrencyRepository(context: Context):
	CryptocurrencyRepository {
	return CryptocurrencyRepository.getInstance(
		AppDatabase.getInstance(context).cryptocurrencyDao())
	}
}
```

6. We create the ViewModelsModule class module which will be responsible for providing ViewModels throughout your application.

ViewModelsModule.kt

```
@Module
abstract class ViewModelsModule {
	@Binds
	@IntoMap
	@ViewModelKey(MainViewModel ::class)
	abstract fun bindMainViewModel(mainViewModel: MainViewModel): ViewModel
	@Binds
	@IntoMap
	@ViewModelKey(AddSearchViewModel ::class)
	abstract fun bindAddSearchViewModel(addSearchViewModel:
	AddSearchViewModel): ViewModel
	@Binds
	abs
```

7. Create a personalized ViewModelKey markup class.

`ViewModelKey.kt`

```
@MustBeDocumented
@Target(
        AnnotationTarget.FUNCTION,
        AnnotationTarget.PROPERTY_GETTER,
        AnnotationTarget.PROPERTY_SETTER
)
@Retention(AnnotationRetention.RUNTIME)
@MapKey
annotation class ViewModelKey(val value: KClass<out ViewModel>)
```

8. Create the ViewModelFactory class.

`ViewModelFactory.kt`

```
@Suppress("UNCHECKED_CAST")
@Singleton
class ViewModelFactory @Inject constructor(
        private val viewModelsMap: Map<Class<out ViewModel>,
        @JvmSuppressWildcards Provider<ViewModel>>
) : ViewModelProvider.Factory {
        override fun <T: ViewModel> create(modelClass: Class<T>): T {
                var creator: Provider<out ViewModel>? = viewModelsMap[modelClass]
                if (creator == null) {
                        for (entry in viewModelsMap.entries) {
                                if (modelClass.isAssignableFrom(entry.key)) {
                                        creator = entry.value
                                        break
                                }
                        }
                }
                if (creator == null) {
                        throw IllegalArgumentException(
                                "Unknown model class $ modelClass"
                        )
                }
                try {
```

```
                return creator.get() as T
            } catch (e: Exception) {
                throw RuntimeException(e)
            }
        }
}
```

9. Create the ActivityBuildersModule class module.

`ActivityBuildersModule.kt`

```
@Module
abstract class ActivityBuildersModule {
        @ContributesAndroidInjector(
                modules = [MainListFragmetBuildersModule ::class])
        abstract fun contributeMainActivity(): MainActivity
        @ContributesAndroidInjector
        abstract fun contributeAddSearchActivity(): AddSearchActivity
}
```

10. We create the mainListFragmetBuildersModule class module.

`MainListFragmetBuildersModule.kt`

```
@Module
abstract class MainListFragmetBuildersModule {
        @ContributesAndroidInjector()
        abstract fun contributeMainListFragment(): MainListFragment
}
```

11. Create the AppComponent interface

`AppComponent.kt`

```
@Singleton
@Component(modules = [
        AndroidSupportInjectionModule ::class,
        AppModule ::class,
        ActivityBuildersModule ::class
```

```
])
interface AppComponent {
        @Component.Builder
        interface Builder {
                @BindsInstance
                fun application(application: App): Builder
                fun build(): AppComponent
        }
        fun inject(app: App)
}
```

12. We generate graph object.
13. Create the Injectable interface
14. We create an AppInjector helper class
15. Configure the App class we created earlier
16. Configure MainActivity in order to inject ViewModel
17. Configure MainListFragment in order to inject ViewModel

10. RESTful Web Services

These days almost every Android app connects to the internet to receive / send data. We need to have knowledge of RESTful Web Services as their correct application is essential knowledge when creating a modern application. We will combine multiple libraries at the same time to get the result of the work.

What is Retrofit, OkHttp and Gson?

Retrofit is a REST client for Java and Android. The library makes it relatively easy to retrieve and upload JSON (or other structured data) through a REST-based service. In Retrofit we configure which converter is used for data serialization. Usually to serialize and deserialize objects from JSON we use an open source Java library - Gson. To make HTTP Retrofit requests use the OkHttp library. OkHttp is a pure HTTP / SP2 client responsible for any low-level network operations, caching, requests and response manipulations.

In contrast, Retrofit is a high-level REST abstraction built on OkHttp. Retrofit is strongly associated with OkHttp and uses it intensively.

Our first goal is to get all the lists of cryptocurrencies using Retrofit from the Internet. We will use the special OkHttp "tapping" class for the CoinMarketCap API We will retrieve a JSON data result and then convert it using the Gson library.

Glide is a fast and efficient open source media management and image uploading framework for Android that uses media decoding, disk memory, and merging resources into a simple, easy-to-use interface. Glide's main focus is on making moving any type of image list as smooth and fast as possible, but it is also effective for almost any occasion when we need to take, resize, and display a sized image big.
In the My Crypto App we have several list screens where we have to show multiple logos or images of cryptocurrencies retrieved from the internet immediately and still provide smooth user experience for the user.

Kotlin Coroutines

As we build this application, we will face situations where we will execute time consuming tasks, such as writing data to or reading from a database, retrieving data from the network, and more. All of these common tasks require more time than allowed by Android.
Coroutines are a feature of Kotlin that converts asynchronous calls to long-term tasks, such as database or network access, into the following code. The return value of a function will provide the result of the asynchronous call. Sequentially written code is usually easier to read, and may even use linguistic features such as exceptions. So we will use coroutines in this application, where we have to wait until a result is available from a long task and that we continue executing.

References:

[1] Android Programming for Beginners: Build in-depth, full-featured Android 9 Pie apps starting from zero programming experience, 2nd Edition
[2] https://developer.android.com/kotlin
[3] https://www.udemy.com/course/the-complete-android-developer-bootcamp/
[4] https://developer.android.com/jetpack/guide?gclid=CjëKCAjëvMqDBhB8EiwA2iSmPEQ2ZHDpksMUHQjRRkdUIjDOzCekjnEdNNksRl7MyxolGTHDhONkRoCb2IcAvd
[5] https://www.jetbrains.com/help/idea/restfulwebservices.html?gclid=CjwKCAjwvMqDBhB8EiwA2iSmPInyw9w9amOyNyop9c47jdoighlCxzyX5IFnKApX7ëdvDNmIw

Mexhit Kurti

Faculty of Natural Sciences
Master of Science in Informatics
Mobile Programming
2021