Android study

Juliane Marubayashi

2022-04-19

Contents

Attention before reading: this document is not original. It is deeply based on the resources. Thus, I don't have any credit on the biggest part of this text.

1 Lecture 1 - Basic Android

The android framework for building applications was developed in C and wrapped in java. The java virtual machine was also written in C and optimized for small devices and cores. The framework itself was based in reuse and extension patterns.

The **application framework** is a **high level library in java** suitable to the creation of user android applications. In other words, it's the basic structure underlying the system (Android API). The framework also creates the higher-level Android management services (Java).

- Activity manager
- Content manager
- Resource manager
- Location manager
- Notification manager

1.1 Application components

The android applications can contain serveral independent components:

- Activities
- Services
- Broadcast Receivers
- Content providers

1.1.1 Activities

- Contain a portion of the user interface
- Execute a well-defined task inside the application
- Are subclasses of the android.app.Activity class
- Are usually composed by a hierarchy of Views

1.1.2 Services

- Don't have user interface, like the activities
- Can execute in background for indeterminate period
- It's possible to estabilish a connection with the service and communicate through a well-defined programming interface

1.1.3 Broadcast

It handles notifications and the applications react to then.

- Can receive and react to notification received from the system or other applications
- Applications can initiate a notification 'broadcast'

1.1.4 Content providers

Basically allow the comunication of data between applications.

- Make available to other applicationo a data collection maintaned by this application
- Define an interface to access, add and update the supported data types

1.2 Component activation

1.2.1 Activities and Services

Activities and services are activated through an **Intent**.
The **intent** identify the component or specify actions for activities and services.
A explicit intent contains the:

- Class reference of the desitnation
- Specification of an action, category, data (in the form of url) and possibly extra information.

Once the specifications are given, the android will try to find a matching component capable of executing ht eaction in the data (or data type) specified. When activities and services are declared in the **manifest** they can specify the *intent-filters*, describing their accepting intents.
An example is calling the activity using an intent:

```
val otherActivity: Intent = Intent(thisActivity, OtherActivity::class.java);
thisActivity.startActivity(otherActivity);
```

1.2.2 Broadcast receivers

The intents for activating broadcast receivers indentify a **message** to be delivered to matching receivers. The message is specificied using: action, category, data and extra info.

```
sendBroadcast(Intent);   // Sends the intent.
onReceive(Context, Intent); // If a broadcast receiver was installed and matches the intent,
    this method will be called.
```

1.2.3 Content providers

When declared in the manifest must have an 'authority'. The 'authority' must recognize a name for its data collection.
The content providers supports the CRUD operations on the specified collection.
They are usually activated by the method **getContentResolver** and the **ContentResolver** can insert, update, delete and query an information.

```
ContentResolver resolver = getContentResolver();
Uri uri = Uri.parse("content://<authority>/<data-collection>[/<item>]");
resolver.query(Uri uri, String[] projection, Bundle queryArgs, CancellationSignal
    cancellationSignal);
resolver.delete(Uri uri, Bundle extras);
resolver.insert(Uri uri, ContentValues values);
resolver.update(Uri uri, ContentValues values, Bundle extras);
```

1.3 Intent (start an activity example)

All the intents have a name (action) and can gave nire data associated (e.g uri, category, extra info).
The intents can also be **explicit** with a class reference (inside an app).
There're many predefined intents.
This first example matches the android activity in the Dialer application, that allows the user to make a phone call, declared in an intent-filter, that can handle this action.

```
val intent = Intent(Intent.ACTION_DIAL);
startActivity(intent);
```

This second example, matches an activity that makes a phone call, but for a given number.

```
val intent = Intent(Intent.ACTION_CALL);
Uri.parse("tel:555-555-555").also{intent.date = it};
startActivity(intent);
```

Creating an intent to start an activity externally the app:

```
Intent LaunchIntent =
    getActivity().getPackageManager().getLaunchIntentForPackage(CALC_PACKAGE_NAME);
startActivity(LaunchIntent);
```

1.3.1 Intents data

Intents **can transport data between components**. In other words, if the application is currently running in an activity, we can initiate another activity and parse the actual information to it by using intents.

The **data** property can be used for any kind of a Uri. The calling and new app components uses the Intent **data** property of the Uri type, by calling **setData()** and **getData()** methods to set it.

The Extra internal field is used for arbitrary data types.

It is a **Bundle** : set of (name, value) pairs organized in a hash table.

- The value can be a String, simple type, or an array.
- Can also be any Serializabe or Parcelable (more efficient) object
- The values are inserted with some **putExtra(String name, ... value)** method and retrieved with **get...Extra(String name)**.

```
override fun onCreate(savedInstanceState: Bundle?){
    super.onCreate(savedInstanceState);
    ...
    /* Activities have an intent property */

    val fName = intent.getStringExtra("firstName");
    val lName = intent.getStringExtra("lastName");
}
```

Disadvantages of using Bundle

When used to pass information between Android components the bundle is serialized into a binder transaction. The total size for all binder transactions in a process is 1MB. If you exceed this limit you will receive this fatal error: `Failed binder transaction`.
It's recommend that you keep the data in these bundles as small as possible because it's a shared buffer, anything more than a few kilobytes should be written to disk.

1.3.2 Activity retuning data

Specially invoked activities **can return data**. This can be done by calling a new activity with **startActivityForResult(...)**.

Besides the intent, it has a requestCode(Int) as a parameter. The new activity should create a **result Intent**, fill it with the result data, and call **setResult**, passing this intent before finishing.

Called activity sending value:

```
...
val resultIntent = Intent();
resultIntent.putExtra("some_key", "String data");
setResult(Activity.RESULT_OK, rsultIntent);
finish();
```

Original activity receiving value:

```
override fun onActivityResult(requestCode, resultCode,: Int, data: Intent){
    super.onActivityResult(requestCode, resultCode, data);
    when(requestCode){
        MY_CHILD_ACTIVITY ->
            if(resultCode == Activity.RESULT_OK){
                val resultValue = data.getStringExtra("some_key");
                ...
            }
    }
}
```

1.4 Colletion of elements

In android we might want to display a collection/list of elements to the user and this list might suffer some updates. For this, we must use the `ArrayAdapter` class. If we want to personalize the adapter, we must override this class.

ArrayAdapter Class Implementation

```
class MainActivity : AppCompatActivity() {
    private val adapter by lazy { RestaurantAdapter() }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        supportActionBar?.setIcon(R.drawable.rest_icon)
        supportActionBar?.setDisplayShowHomeEnabled(true)
        setContentView(R.layout.activity_main)
        app.adapter = RestaurantAdapter(this, app.rests)

        val list = findViewById<ListView>(R.id.listview)
        list.adapter = app.adapter
    }


    inner class RestaurantAdapter : ArrayAdapter<Restaurant>(this@MainActivity,
         R.layout.list_row, rests) {
        override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
            val row = convertView ?: layoutInflater.inflate(R.layout.list_row, parent, false)
            val r = rests[position]
            row.findViewById<TextView>(R.id.title).text = r.name
            row.findViewById<TextView>(R.id.address).text = r.address
            val symbol = row.findViewById<ImageView>(R.id.symbol)
            when (r.type) {
                "sit" -> symbol.setImageResource(R.drawable.ball_red)
                "take" -> symbol.setImageResource(R.drawable.ball_yellow)
                "delivery" -> symbol.setImageResource(R.drawable.ball_green)
            }
            return row
        }
    }
}
```

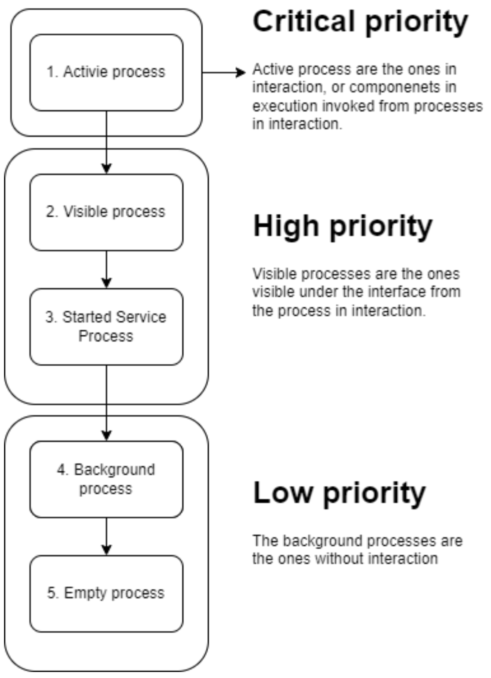
2 Lecture 2 - Applications, activities, interface

2.1 Processes

Each application, including all its components runs in a single process. The application object from the framework is always in memory (singleton object);

2.2 Priorities

Processes are organized in priority levels depending on the state of its components. When there is some lack of resources processes are automatically terminated.



1. Activie process
2. Visible process
3. Started Service Process
4. Background process
5. Empty process
Critical priority
Active process are the ones in interaction, or componenets in execution invoked from processes in interaction.
High priority
Visible processes are the ones visible under the interface from the process in interaction.
Low priority
The background processes are the ones without interaction

Figure 1: Process priorities

2.3 Activities

Usually show an user interface (using ViewGroups and Views) and inherit from android.app.Activity. The activities execute a specific task in the application and the android always mantains a stack of activities.

Let's suppose that one activity inits another one. The first activity will go to the stack. When the back button is pressed, the first activity is retrieved from the stack.

An activity might be:

- active (runnning) - In interaction
- paused - Visible;
- Stopped - Non visible;
- Inactive (destroyed) - Without an instance object;

The android call lifecycle methods when there's a state transition.

2.3.1 In interaction vs Paused

Let's suppose that we are running an active and then a dialog box pops up. Considering that the dialog box is managed by another activity, we can say that the previous activity is paused, but visible. The current activity (i.e the dialog box) is the active one.

2.3.2 Starting an activity

Before starting, take a look on the diagram below. This is the complete state diagram for android.

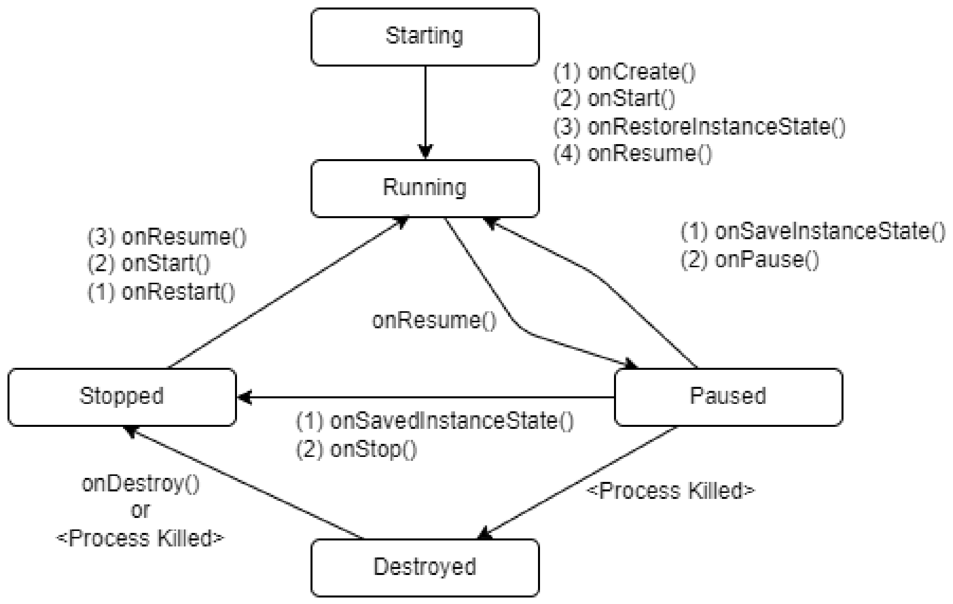


Figure 2: Activities State Diagram

Firstly the android system call life-cycle callbacks starting with the **onCreate()**.

```
// Always override
override fun onCreate(savedInstanceState: Bundle?){
    super.onCreate(savedInstanceState)
    val bt = findViewById<Button>(R.id.some_button)
    bt.setOnClickListener(this)
}
```

We should associate the listeners to interacion events in onCreate().

2.3.3 Activity Navigation

There are two criterias to the navigation:

- Based on the past activated activity and the activity stack. This is usually used when the user presses the **back button**.
- Based on a defined hierarchy. We can declare an activity parent in the manifest file for each activity (except the one that starts the application); and a person can use the Up button in the action bar to navigate.

2.4 Resources

The resources can be:

- drawable - bitmaps, graphics, shapes, etc
- anim - XML specifying animations between 2 configurations (tweens)
- color - colors (AARRGGBB) for many elements of the interface according to state (pressed, focused, selected, ...)
- layout - screen organization
- menu - options and context menus specifications
- values - value collections with a name: string, arrays, colors, styles, dimensions,...
- xml - other XML files read with getXML()

2.5 Events and listeners (IoC pattern)

2.5.1 Inversion of Control (IoC)

IoC is the inversion control principle. Actually it's a principle, not a design pattern. This principle defines a basic characteristic of frameworks and it was popularized in 2004 by Robert C. Martin and Martin Fowler.

Imagine that we have a code like this:

```java
public static void main(String[] args) {
    while (true) {
        BufferedReader userInputReader = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.println("Please enter some text: ");
        try {
            System.out.println(userInputReader.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The code is really simple: it will try to read the user input for undefined time and print in the console. The main function/file has controls the program flow.

Now consider that we've decided to use a GUI to do the same logic. The **framework is responsible for calling the user code.** The framework is an extendable structure that provides the developer a set of specific points for injecting segments of the code.

IoC in short

In the case of frameworks that stick to the open/closed principle by providing an extendable API, the role of the developer using the framework boils down to defining their own set of custom classes, either by implementing one or multiple interfaces provided by the framework or by inheriting from existing base classes. In turn, instances of the classes are directly instantiated from and called by the framework.

2.5.2 IoC in android

In android the framework needs to listen the user code and execute it. Everytime a user makes an interaction, a message goes to a message queue, which is executed by a main thread (e.g UI thread). The main therad runs a loop that reads the message from the queue and makes a callback (life-cycle) or calls an event linstener (handler).

Here, the framework has the program flow.

2.6 Events generated by the interface

Many views generate events, such as clicking in a button, changing focus, etc. We need to install listeners to these evets, by implementing the interfaces that describe them. Them implemented methods are called when the event occurs, such as:

- onClick()
- onLongClick()
- onFocusChange()
- onKey()
- onTouch()
- onCreateContextMenu()

Defining a listerner is easy and can be defined in an activity:

```
// Listener defined in an activity
class ExampleActivity: Activity(), OnClickListener{
    ...
    val button = findViewById<Button>(R.id.corky);
    button.setOnClickListener(this);

}
// Implement the OnClickListener interface
override fun onClick(v: View){
    // Do something
}
```

As an anonymous object listener:

```
override onCreate(savedValues: Bundle?){
    ...
    // Capture our button from layout
    val button = findViewById<Button>(R.id.corky)
    // Register the onClick listener with the implementation
    button.setOnClickListener(object: View.OnClickListener) {
        override fun onClick(v: View?){
            // do something
        }
    }
}
```

This last approach is no longer considered a good practivce, because it can create confusion and deeper dependencies between layouts and code.

We can also use a lambda:

```
    ...
    button.setOnClickListener(v->btClick(v))
}
fun btClick(v: View){
    // do something
}
```

Disadvantages on IoC

A disadvantage of this pattern is that is one task takes too long, it will make the application slow. Because it's only one thread.

2.7 Menus

There're two types of menu in android: the **option menus (associated to activities)** and the **context menu (associated to views)**.

2.8 State preservation

Standard activities in an application save some of their internal state whenever the activity is in its way to the destroyed state. If the activity is active again it will restore its state before the next time it is displayed. This facility can be extended for non-standard activities (i.e., graphical) or other type of state.

Activities call onSaveInstanceState(Bundle) and onRestoreInstanceState(Bundle) when they need to save and restore state (it is saved on memory). These methods can be overridden in the derived class.

```
override fun onSaveInstanceState(state: Bundle) {
    state.putFloat("Score", mCurrentScore)
    state.putInt("Level", mCurrentLevel)
    super.onSaveInstanceState(state)
}
override fun onRestoreInstanceState(state: Bundle) {
super.onRestoreInstanceState(state)
mCurrentScore = state.getFloat("Score")
mCurrentLevel = state.getInt("Level")
}
```

2.9 Rotation

When certain situation occurs that cause a change in device configurations (rotating the devive, extending or hiding a logical keyboard, docking and undocking, changing the locale or language), Android **destroys** and **re-creates** the running and paused activities, the next time are viewd.

This could be necessary to load new resources for the user interface, more adapted to the new situation.

Device rotation is a very common situation, and every user expects that all applications support this change, eventually adapting the interface to portrait and landscape.

When the activity is desstroyed, all internal variables loses their values, and the activity default mechanism default of saving state based on the **onSaveInstanceState()** and **onRestoreInstanceState()** methods only saves and restores some of the internal contents of the views (but not all).

Overriding this methods (e.g onRestoreInstanceState() and onSaveInstanceState()) we save information to the same bundle. But this bundle is saved in memory as long as the activity remains in the stack. If the process is popped from the stack (e.g click the back button) the Bundle is lost.

2.10 Feature and permissions

Declared in the manifest when an application needs a certain hardware characteristic that can be unavailable. The features are defined in the Android class PackageManager.

To use certain API functionalities the application needs permission from the user. We can install it, by writing the permission in the manifest.

3 Lecture 3 - Fragments

Activities were created to occupy the enterire screen and that's ok for small devices, but for large devices such as tablets, we could use the space in a better way.

Fragments can be seen as sub-activities, as they may respond to back-buttons and have their own **life-cycle**. However, a fragment lifecycle is related to the activity lifecycle. One activity can have many fragments and the point is that they all use the same thread (the main one).

3.1 Fragment Lifecycles

Now let's explain the sequence of commands called in a fragment life-cycle.

- `onInflate(activity: Activity, attrs: AttributeSet, savedInstanceState: Bundle)` : Called in the beggining of the activity, when the fragment sets its content layout. The AttributeSet contains the attributes defined in the activity layout. They should be parsed and saved. The fragment is not yet attached to the activity.

- `onAttach(activity: Activity)`: Now the activity is attached to the fragment. In other words, it's possible to get the activity with the function `getActivity()` inside the fragment. It's also possible to get the arguments with `getArguments()`, but only the arguments set until this moment.

- `onCreate(savedInstanceState: Bundle)`: **It's when the fragment starts the process of creation** called at the beginning of the owner activity onCreate() callback. Usually, the **activity View hierarchy is not yet inflated**. You can create here another thread to do lengthy data loading operations.

- `onCreateView(inflater: LayoutInflater, container: ViewGroup, savedInstanceState: Bundle)`: This function should return the inflated View of this fragment. If the container is null, then null must be returned. The container is the ViewGroup in the activity layout that will display the fragment. Do not attach the fragment to this container. Some specialized fragments (e.g., ListFragment) do not need this callback.

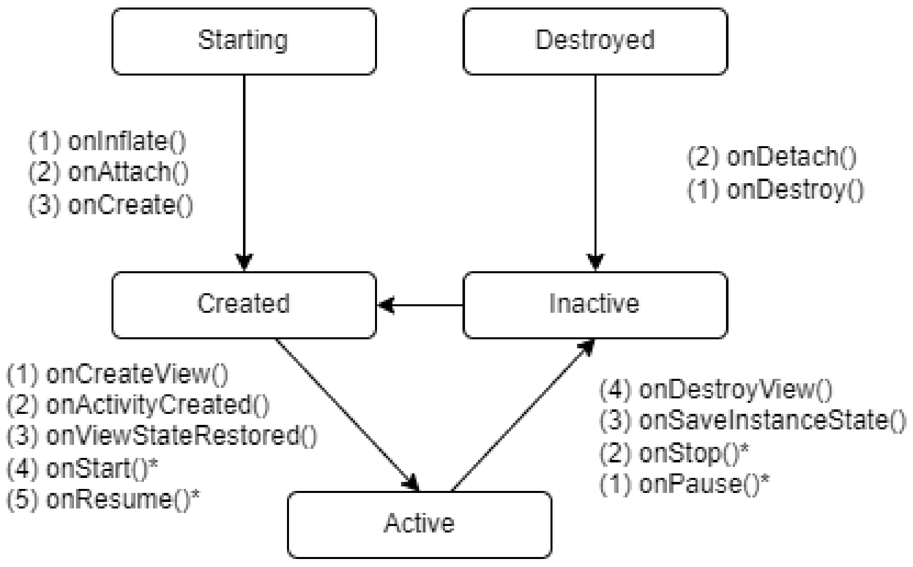


Figure 3: Fragment State Diagram

- `onActivityCreated(savedInstanceState: Bundle)`: here the onCreate() method of the activity is now complete. The complete interface is now built, including other present fragments.
- `onStart() and onResume()`: tied with the activity corresponding callbacks.
- `onPause()`: the first to be called (the fragment can be put on the fragment back stack). You should stop playing sounds, related to this fragment, here.
- `onStop()` : tied with the onStop() callback of the activity. A stopped fragment can go straight to the onStart() callback.
- `onDestroyView()`: when the fragment is being killed or saved, this will be called. Here its View hierarchy is already detached from the activity layout.
- `onDestroy()`: is called when the fragment is no longer in use (but still existing in the activity).
- `onDetach()`: here the fragment does not belong anymore to the activity and the interface resources are already freed.
- `onSaveIntanceState(outstate: Bundle)`: called somewhere before onDestroy(). It should save internal state in the provided Bundle. That Bundle is passed to the entry callbacks.

3.2 Fragment creation

The fragment creation can be **STATIC** or **DYNAMIC**.

- **STATIC**: Whenever the activity inflates its layout and has a ¡fragment¿ element in it, the system constructs the Fragment
- **DYNAMIC**: If Fragments are built in the code, then a static factory method must be created in the fragment class. It's recommended to have a factory for the fragment, since fragments doesn't have a constructor.

Fragment Factory

```
fun MyFragment newInstance(index: Int) {
    val f = MyFragment()
    val args = Bundle()
    args.putInt("index", index)
    f.arguments = args
    return f
}
```

Bundled arguments

The arguments for a fragment inialization must be received in the factory must be passed as parameter and then *Bundled*. Bundled arguments are **preserved in the rotation** and are available as property.

3.3 Fragment Manager (supportFragmentManager)

Activities and fragments can manager other active fragments. This can be done by using the `FragmentManager` object. This object is available at the property `supportFragmentManager`.

It can:

- Find fragments
- Manipulate the fragment back stack
- Save and restore references to fragments and fragment internal state
- Add, replace, remove, hiding and showing can be done and must be executed as a transaction.

Example:

Fragment transaction

```
/**
 * Changes the view to a fragment that shows the items inside the history.
 */
private fun changeToBasketHistoryProducts(){
    val basketHistoryProducts = BasketHistoryProducts.newInstance(intent.extras!!)
    val fragmentManager = this.supportFragmentManager
    val fragmentTransaction = fragmentManager.beginTransaction() // Starting transaction.
    fragmentTransaction.add(android.R.id.content, basketHistoryProducts)
    fragmentTransaction.commit() // Should end with commit

}
```

3.4 Implementing a fragment

Fragment implementation

```
// In the parent activity
class ParentActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val frag = MyFragment.newInstance(intent.extras!!)
    ...
  }
  ...
}

// In the Fragment class
class MyFragment : Fragment() {
  var mIndex = 0;

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    mIndex = arguments?.getInt("index") ?: 0
  }

  override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
      savedInstanceState: Bundle?): View? {
```

```
    if (container == null)
      return null

    // Don't tie this fragment to anything through the inflater
    val v = inflater.inflate(R.layout.details, container, false)
    v.findViewById<TextView>(R.id.text1).text = Planets.PROPS[mIndex]
    return v
  }

  // Fragment factory
  companion object {
    @JvmStatic
    fun MyFragment newInstance(index: Int) {
      val f = MyFragment()
      val args = Bundle()
      args.putInt("index", index)
      f.arguments = args
      return f
    }
  }
}
```

NOTE: The method of building dialogs from the Dialog class doesn't take care automatically of lifecycle events, like the destruction and re-creation. You should embed the `DialogFragment`.

4 Lecture 4 - NFC

There're three modes that the NFC can work:

- Read and write tags (good, API 10);
- Peer-to-Peer (beam/limited, API 14);
- Card emulator (real/emulated SE, API 19);

The android hardware can include a **secure element** which is similar to a smartcard running JavaCardOS. For more information about JavaCardOS you can read:
https://www.oracle.com/java/technologies/java-card/javacard1.html

4.1 NdefMessage class

Android supports NFC messages by using `NdefMessage` class.

Using NdefMessage class

```
val mimeType = "application/my.mime.type" // a custom mime type
val payload = "This is a TNF_MIME_MEDIA"
val mimeRecord = NdefRecord.createMime(mimeType, payload.getBytes(Charsets.UTF_8))
val newMessage = NdefMessage(arrayOf(mimeRecord))
```

The constructor `NdefMessage` accepts a list of records (`NdefRecord`) and these methods can be obtained through `getRecords()` method.

The `NdefRecord` constructor is: `NdefRecord(tnf: Short, type: ByteArray, id: ByteArray, payload: ByteArray)`.

The type, id and payload are conditioned by the TNF field.

- **Tags**: TNF values of TNF_ABSOLUTE_URI or TNF_WELL_KNOWN are usually used, with corresponding values of type, id, and payload.
- **Device to device messaging**: the TNF value is usually TNF_MIME_MEDIA with a custom media type in type, an id of 0 bytes, and a payload.

The `NdefRecord` class has get methods for the TNF, type, id, and payload constituents. It has also static `NdefRecord` methods for building some kinds of records, like `createMime()`.

4.2 Setup manifest

The following code must be added:

```
<uses-permission android:name="android.permission.NFC" />
<uses-sdk android:minSdkVersion="10"/>
<uses-feature android:name="android.hardware.nfc" android:required="true" />
```

Not finished...AAAAAAHHH!

5 Lecture 5 - Android HTTP and Asynchronous processing

5.1 HTTP request

Calling webservices uses HTTP protocols.
To execute this, **a separate thread is necessary** as well as a manifest permission:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

After API 27, only HTTPS is allowed by default. Use `android:usesCleartextTraffic="true"` in the <application> manifest tag for allowing HTTP.

Using HttpURLConnection

```
private class AddUser(val address: String, val uname: String) : Runnable {
    override run() {
        val url: URL? = null
        val urlConnection: HttpURLConnection? = null
        try {
        // Configure -------------------------------------------
        url = URL("http://" + address + ":8701/Rest/users")
        urlConnection = url.openConnection() as HttpURLConnection
        urlConnection!!.setDoOutput(true)
        urlConnection!!.setDoInput(true)
        urlConnection!!.setRequestMethod("POST")
        urlConnection.!!setRequestProperty("Content-Type", "application/json")
        urlConnection!!.setUseCaches(false)

        // Payload ---------------------------------------------
        val outputStream = DataOutputStream(urlConnection!!.getOutputStream())
        val payload = "\"" + uname + "\""
        outputStream.writeBytes(payload)
        outputStream.flush()
        outputStream.close()

        // Response --------------------------------------------
        val responseCode = urlConnection!!.getResponseCode()
        if (responseCode == 200)
        val response = readStream(urlConnection!!.getInputStream(); // ... and transmit to UI
        }
        catch (Exception e) { . . . // treat the exception
        }
        finally {
            if(urlConnection != null) urlConnection.disconnect()
        }
    }
}
```

Using threads to invoke

```
val addUser = AddUser(address, name)
val thr = Thread(addUser)
thr.start();
```

5.2 Threads

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is in charge of dispatching events to the appropriate user interface widgets and for interacting with components from the Android UI toolkit. As such, the main thread is also sometimes called the UI thread.

Android UI toolkit is not thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Activity.runOnUiThread(Runnable) is a way to access the UI thread from another thread to solve this problem.

5.3 Communication between threads

The problem with the java communication systems between threads (e.g pipes, blocking queue, shared memory, etc) is that these mechanisms are blocking. Since the communication between threads are mainly done between the UI thread and background threads, an android mechanism was created to handle this communication.

5.3.1 Handlers

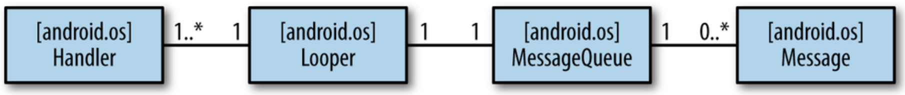


Figure 4: Thread handler overview

- `android.os.Looper`: A message dispacher associated with one and only one cosumer thread.
- `android.os.Handler`: Consumer thread message processor, and the interface for a producer thread to insert messages into the queue. A Looper can have many associated handlers, but they **all insert messages into the same queue**.
- `android.os.MessageQueue`: Unbounded linked list of messages to be processed on the consumer thread. Every Looper - and Thread - has at most one MessageQueue.
- `android.os.Message`: Message to be executed on the consumer thread.

Send messages between threads

```
// Cosumer threads
MyHandler(val uiActivity: MyActivity) {
    override handleMessage(m: Message) {
        val s = m.getData().getString("msgstr")
        // ... uiActivity.doSomething(s)
    }
}

// Producer threads
MyRunnable(val uiHandler: Handler) {
    fun run() {
        ...interact()...
    }
    fun interact() {
        val m = uiHandler.obtainMessage()
        m.setData(createBundleFromStr("something"))
        // boolean sendMessage(Message msg)
        uiHandler.sendMessage(m) // Send message to the queue.
    }
}

// In the activity
val myHandler = MyHandler(this)
val worker = Thread(MyRunnable(myHandler))
worker.start()
```

The code above follows the schema:

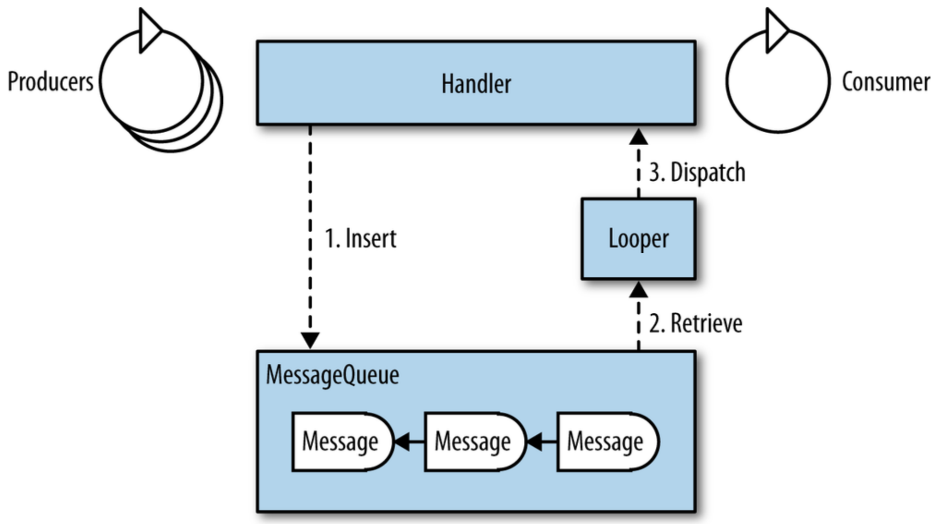


Figure 5: Sending message flow schema

5.3.2 Messages as Tasks (Post)

The Handler inserts messages in the message queue in various ways depending on the message type.

- Task messages are inserted through methods that are prefixed post;
- Data insertion methods are prefixed send, like we saw in the previous section.

To add a task in a queue we can do:

```
boolean post(Runnable r)
```

Post an no parameters

```
private val handler = Handler()

private fun mainProcessing() {
    val thread = Thread(doBackgroundThreadProcessing, "Background")
    thread.start()
}

private val doBackgroundThreadProcessing = Runnable() {
    override run() {
        [ ... Time consuming operations ... ]
        handler.post(doUpdateGUI) // send a task to the queue.
    }
}

// Runnable that updates GUI on the UI thread
val doUpdateGUI = Runnable() {
    override run() {
        [ ... Open a dialog or modify a GUI element ... ]
    }
};
```

5.4 Async Task

The message passing mechanism between thread is a **nonblocking** consumer-producer pattern.

Creating and running Async tasks

```
// AsyncTask<[Input Parameter Type], [Progress Report Type], [Result Type]>
private class MyAsyncTask() : AsyncTask<String, Int, Int> {
    // Is also executed by the UI thread when the background task call publishProgress.
    There is parameter passing between these methods.
    override onProgressUpdate(progress: Int) {
        // [... Update progress bar, Notification, or another UI element ...]
    }

    // Is executed by the UI Thread when doInBackground finishes.
    override onPostExecute(result: Int) {
        // [... Report results via UI update, Dialog, or notification ...]
    }

    // Is executed by a background task when AsyncTask is excuted.
    override doInBackground(parameter: String ): Int {
        val myProgress = 0
        // [... Perform background processing task, update myProgress...]
        publishProgress(myProgress)
        // [... Continue performing background processing task ...]
        // Return the value to be passed to onPostExecute
        return result
    }
}

// Executing
MyAsyncTask().execute("inputString");
```

6 Lecture 6 - Broadcast, Services and Notifications

6.1 Broadcast

Are application components that can receive 'intents' from other components.
We can use the broadcast to send information to another app and inside the same app.
Broadcast receivers can be declared in **the manifest** or **registered dynamically**.
They also can have an associated ACTION or cross-application explicit intent.
Are invoked using `sendBroadcast()`, which can be invoked by any other component or other application.
To receive the broadcast message, one must use the `BroadcastReceiver`. It's necessary to override the `onReceive` class, because this class doesn't have any user interface.
Let's create a `BroadcastReceiver` and declare it in the manifest:

```
...
<application>
...
<receiver android:name=" ... " >
<intent-filter>
<action android:name=" ... " />
</intent-filter>
</receiver>
...
```

In the activity, we gonna send the a broadcast message:

```
Intent bi = ... // should match the name of the action in the manifest.
sendBroadcast(bi);
```

An example is:

Manifest definition

```
<manifest>
    <application>
        ...
        <receiver android:name=".MyReceiver">
            <intent-filter>
                <action android:name="org.feup.intents.test" />
            </intent-filter>
        </receiver>
        ...
    </application>
    ...
</manifest>
```

Broadcast activity

```
public class MyActivity extends Activity {
    ...
    private void invokeReceiver() {
        Intent broadcast = new Intent ( "org.feup.intents.test");
        broadcast.putExtra("somename", "Hello");
        sendBroadcast(broadcast);
    }
    ...
}
```

The receiver

```
// This class can be instatiated in any component.
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String msg = intent.getStringExtra("somename");
        //Do something
    }
}
```

6.2 Services

As we have seen before the services are invisible to the user and are not an asynchronous execution itself. The **Service runs in the UI thread**, so it can degrade responsiveness and cause ANRs (application not responding), even though it does not interact directly with the UI. Still, the `Service` in combination with asynchronous executor is a powerful tool for background task execution.

6.2.1 Why use a Service for asynchronous execution?

- *Decouple lifecycles of components and threads*: Even after the component that started the thread finishes itself, the thread will continue running. Threads may keep references to Java objects so that they cannot be garbage collected until the thread terminates.
- *Lifecycles of the hosting process*: If the runtime terminates the process, all of its threads are terminated and not restarted by default when the process is restored. A process with no active components is likely yo be eligible for termination, since a task with no components contains low rank. For example, an `Activity` stores user data to a database in a background thread while the user navigates back leaves an empty process if there are no other components running. This increases the risk of process termination, aborting the background thread before it can persist the data.

Thus, a Service can mitigate both the risk for memory leaks and the risk of having tasks terminated prematurely. **Services has high priority**, but still less, than activities.

6.2.2 Creation and Execution

```
    public class EatService extends Service {
        @Override
        public void onCreate() { /* Initialize component */ }

        @Override
        public void onDestroy() {/* Clean up used resources*/}

        @Override
        public IBinder onBind(Intent intent) { /* Return communication interface */ }
    }
```

The only mandatory method is onBind, which returns the communication interface to the client to perform remote call services.

To **start** the service, one can call startService(Intent) and to stop it's possible to call stopService(Intent) or Service.stopSelf().

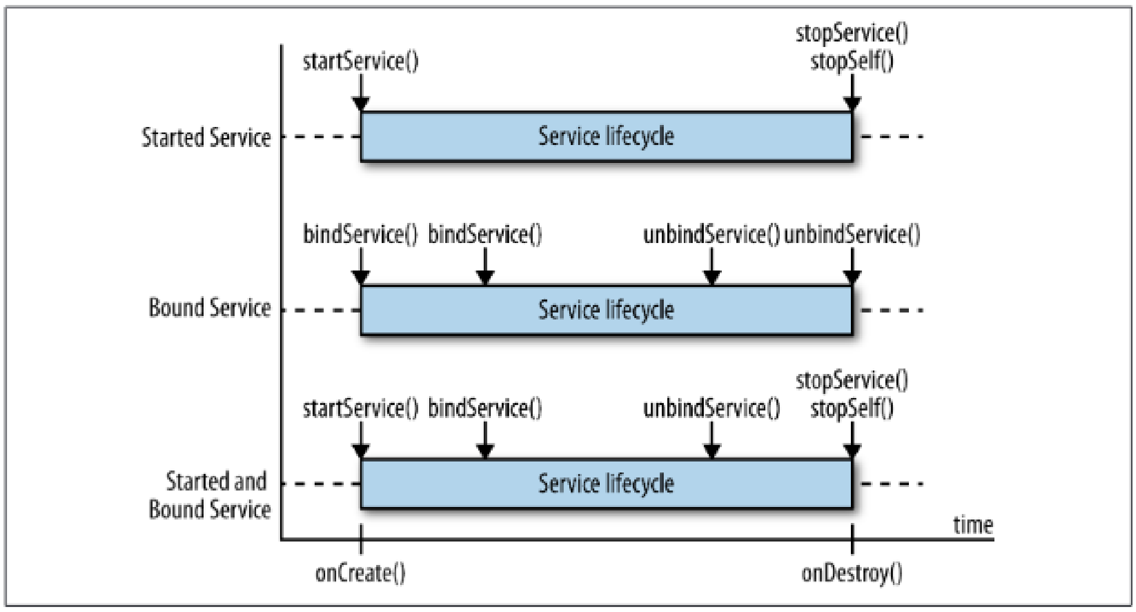


Figure 6: Service lifecycle

6.2.3 Started service

Components invoke Context.startService(Intent) to send start requests to a Service, which can be invoked by multiple components and multiple times from every components during lifecycle. The first start request creates and starts the Service, whereas consecutive start requests just pass on the Intent to the started Service so that the data conveyed in the Intent can be processed.

Services always have to implement onBind, but started services—which do not support binding—should provide a trivial implementation that just return null:

```
public class StartedEatService extends Service {
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) { ... }

    @Override
    public IBinder onBind(Intent intent) { return null; }
}
```

Started services must implement an `onStartCommand` method that handles start requests. The method is invoked each time a start request (`Context.startService`) from a client component is ready to be processed.

Start requests are delivered sequentially to onStartCommand and remain pending in the runtime until preceding start requests are processed or offloaded from the UI thread.

Problems of onStartCommand

onStartCommand is executed on the UI thread, so you should spawn background threads within the method to execute long-running operations, not only to preserve responsiveness but also to enable concurrent execution of multiple start requests.

`onStartCommand` is the key method for implementing started Services and intiating asynchronous task execution. The arguments are:

- *Intent*: Data to be used for the asynchronous execution; e.g., a URL to a network resource that shall be retrieved.
- *Delivery method*: A flag reflecting the history of the start request. This argument may contain other flags in future versions of Android. Possible values are currently 0, START_FLAG_REDELIVERY (value of 1), or START_FLAG_RETRY (value of 2).
- *StartID*: A unique identifier provided by the runtime for this start request. If the process is terminated and restarted, onStartCommand is called with the same start ID.

The return value of `onStartCommand` and the second argument (the delivery method flag) let you control what happens after your `Service` is terminated. It tells the runtime weather to restart the `Service` and resubmit the `Intent` argument, in case the runtime has to terminate the process for lack of resources and then restart it.

6.2.4 Options for restarting

ike any Android application, your Service may be terminated by the runtime if there are too many processes running on the device. In fact, as a background process, your Service has a greater chance of being killed than many other processes.

This section covers termination by either the runtime or a client, not a clean termination through `Service.stopSelf`.

The values that the `onStartCommand` may return are:

- `START_STICK`: The `Service` will be restarted in a new process whether or not here are any requests pending. The resources are brought to memory again, but with a `NULL` intent, because `onStartCommand` is invoked with a `null` value for the `Intent` argument. However, the `Service` will receive any pending start requests that remained undelivered when the previous `Service` process was terminated. The pending start requests are delivered with the `START_FLAG_RETRY` set in the call's second argument.
- `START_NOT_STICK`: Like `START_STICK`, except that the `Service` will be restarted only if there were pendind start requests when the process was terminated. An `Intent` will always be passed. Then the resources are not brought back to memory until a new `startService` is executed.
- `START_REDELIVER_INTENT`: The `Service` will be restarted and reves both pendind requests and requests that were previously started and had no change to finish. The pending requests are delivered with the `START_FLAG_RETRY` set in the second argument, whereas the previously started requests are redelivered with the `START_FLAG_REDELIVERY` set. The resources are brought to memory again with the last process intent.

6.2.5 Example

Service code template

```java
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
public class MyService extends Service {
    @Override
    public void onCreate() {
        // TODO: Actions to perform when service is created.
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null; // mandatory but should return null for
        // non remote call services
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // Usually launch a background thread to do processing.
        return Service.START_NOT_STICKY; // or other value
    }
    @Override
    public void onDestroy() {
        // TODO: Actions to perform when service is destroyed
    }
}
```

Manifest

```xml
    <service android:name=".MyService"/>
```

Calling the service

```java
// Implicitly start a Service
Intent myIntent = new Intent(MyService.ORDER_PIZZA);
myIntent.putExtra("TOPPING", "Margherita");
startService(myIntent);

// Explicitly start a Service in the same process
startService(new Intent(this, MyService.class));
```

Stoping the service

```java
// With the same intent
stopService(new Intent(MyService.ORDER_PIZZA));

// Stop a service with the service name (same proc).
ComponentName service = startService(new Intent(this, MyService.class));
...
stopService(new Intent(this, service.getClass()));

// Stop a service explicitly in the same process
Class serviceClass = Class.forName(service.getClassName());
stopService(new Intent(this, serviceClass));
```

6.3 Intent service

It's a special purpose Service subclass that creates a single worker thread. The intent received on `onStartCommand()` is passed to the method that the worker thread executes.

The `IntentService` is suitable for when you want to offload tasks easily from the `UI thread` to a background thread with sequential task processing, giving the task a component that is always active in order to raise the process rank.

Successive calls on `onStartCommand()` are queued. You only have to override and implement `onHandleIntent()`.

IntentService implementation

```java
public class MyService extends IntentService {
    public MyService() {
        super("MyService");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        // Do the work in this single worker thread
        // and return
    }
}
```

6.4 Result Service

Mechanism to return a result to an `Activity` (or other component activated by an `Intent`) from other component or thread (using a `Handler()`).

It is created on the destination with `onReceiveResult()` overridden. As this class is `Parcelable` their objects can be passed in Intent. The recipient sends results using `send()`, triggering a call to `onReceiveResult()`.

6.4.1 Example

Recipient component (an activity)

```java
//recipient Activity
public class MainActivity extends AppCompatActivity {
    // some variables (Activity state)
    ....
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Intent aService = new Intent(this, MyService.class);
        aService.putExtra(MyService.RESULT, new ResultReceiver(new Handler()) {
            @Override
            protected void onReceiveResult(int code, Bundle data) {
                super.OnReceiveResult(code, data);
                .... // if code OK, use data and other Activity state
                ....
            }
        });
    }
}
```

A service sending results

```java
public class MyService extends Service {
    public final static String RESULT = "RemoteResult";
        ...
    @Override
    public int onStartCommand(Intent i, int flags, int sId) {
        ResultReceiver rec = i.getParcelableExtra(MyService.RESULT);
        ...
        Bundle data = new Bundle();
        data.putString("value", "some data");
        ...
        rec.send(1, data);
        return Service.START_NOT_STICKY;
    }
    ...
}
```

6.5 Remote Call Services (RPC)

Their functionality is invoked using RPC. Usually, they are standalone in their own processes.

Remote call services are activated (brought to memory and `onCreate()` invoked) through `bindService()` and can be freed when the last bound client calls `unbindService()`.

When a service is ready to be called through its interface a callback `onServiceConnected()` is called on the client.

There is also a `onServiceDisconnected()` callback on the client that is called when the service is not available (motivated by a crash or reclaimed by Android).

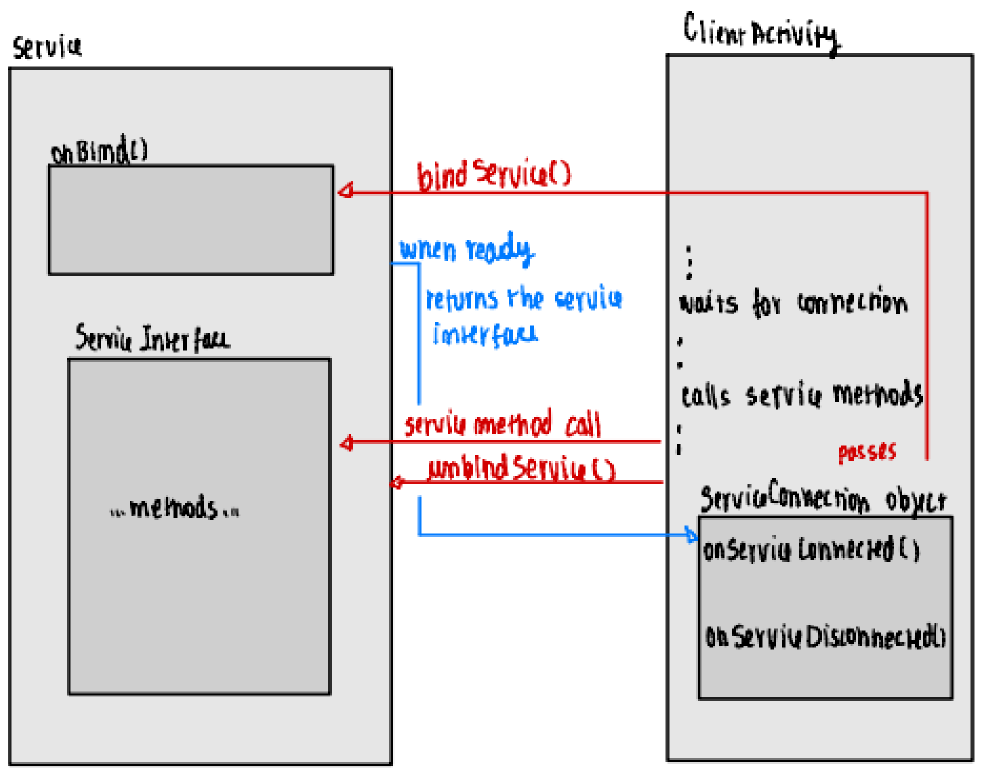


Figure 7: RPC flow

6.5.1 Example

Service interface is defined in the AIDL file

```
// This file is IStockQuoteService.aidl
package com.androidbook.services.stockquoteservice;
    interface IStockQuoteService {
    double getQuote(String ticker);
}
```

Service must implement the interface

```
public class StockQuoteService extends Service {
    public class StockQuoteServiceImpl extends IStockQuoteService.Stub {
        @Override
        public double getQuote(String ticker)
            throws RemoteException {
                return 20.0;
            }
        }
    @Override
    public IBinder onBind(Intent intent) {
        return new StockQuoteServiceImpl();
    }
}
```

Client calling the service

```
...
bindService(new Intent(IStockQuoteService.class.getName()), serConn, Context.BIND_AUTO_CREATE);
...
private ServiceConnection serConn = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        stockService = IStockQuoteService.Stub.asInterface(service);
        callBtn.setEnabled(true);
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
        callBtn.setEnabled(false);
        stockService = null;
    }
};
...
try {
    double val = stockService.getQuote("ANDROID");
    Toast.makeText(this, "Value from service is " + val, Toast.LENGTH_SHORT) .show();
} catch (RemoteException ee) {
}
```

6.6 Notifications

Are shown in the status bar. More details listed in the extended status drawer. They can produce sound, vibration and light leds.

They are created using a system service:

Creating notifications

```
String svcName = Context.NOTIFICATION_SERVICE;
NotificationManager notificationManager;
notificationManager = (NotificationManager) getSystemService(svcName);
```

Are specified in a Notification object through a Build class:

Build class

```
// A small icon, a title and a text and mandatory (many other features)
// get the Notification object using the build() method
Notification notf = new Notification.Builder(this)
.setContentText(message) // the main text of the notification
.setContentTitle(title) // the first line (title)
.setSmallIcon(R.drawable.nticon) // icon on bar and notification
```

```
.setWhen(System.currentTimeMillis()) // for ordering
.setPendingIntent(PendingIntent pi) // Activity to launch on tap
.build(); // returns the notification object
notf.flags |= Notification.FLAG_ONGOING_EVENT; // cannot be cleared
```

Sent using the `notify()` method of the service.
Notifications are customizable.

6.7 Alarms

Calls an application component periodically or after a specified time interval and uses another system service:

Creating an alarm

```
String svcName = Context.ALARM_SERVICE;
AlarmManager alarms;
alarms = (AlarmManager) getSystemService(svcName);
```

We can use the methods `set()`, `setRepeating()` or `setInexactRepeating()` to create alarms.

```
int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;
long timeOrLengthOfWait = 10000;
String ALARM_ACTION = "ALARM_ACTION";
Intent intentToFire = new Intent(ALARM_ACTION);
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intentToFire, 0);
alarms.set(alarmType, timeOrLengthOfWait, pendingIntent);
```

7 Lecture 7 - Sensors

7.1 Location Providers

A device may have several providers. Location can be obtained directly from satellites (GPS). Or can be derived from Wi-fi access points information, mobile communication tower's location.

The devices are condensed in the provider: GPS_PROVIDER ("gps"), NETWORK_PROVIDER ("network"). Also usually exists a PASSIVE_PROVIDER (another app)

GPS is accurate, has more info, more power consumption, more delay, needs line of sight to satellites (weak signals).

Network is less accurate, has less consumption, less delay, interiors.

Android manifest must declare permissions:

- ACCESS_COURSE_LOCATION
- ACCESS_FINE_LOCATION

7.2 LocationListener

Interface declaring methods (callbacks) where information is delivered after requesting a location.

LocationListener interface

```
public interface LocationListener {
    void onLocationChanged(Location location);
    void onProviderDisabled(String provider);
    void onProviderEnabled(String provider);
    // Not called in api 29.
    // status: OUT_OF_SERVICE, TEMPORARILY_UNAVAILABLE,AVAILABLE
    // satelities: nr. of satelities in gps.
    void onStatusChanged(String provider, int status, Bundle extras);
}
```

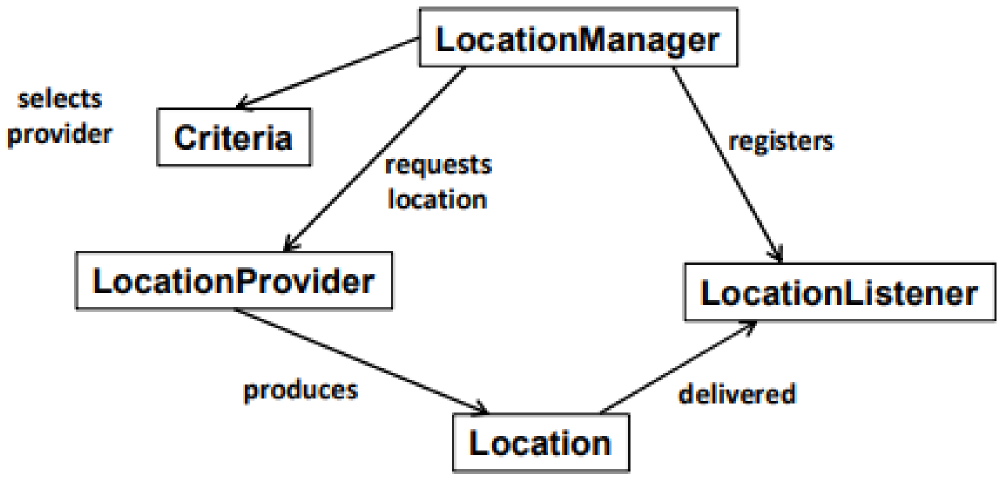


Figure 8: Location Manager Diagram

7.3 LocationManager

System service for requesting locations and selecting a provider. Obtained in an Activity:

```
locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);
```

Locations should be requested only when the activity is active and cancelled when it stops being in foreground. So requests can be done in the `onResume()` callbackof the activity, from the LocationManager object, with:

```
// minTime: minimum time or distance for a new location
// criteria: selection of a provider
// listener: class implementing the listener
// looper: null for the current thread
requestLocationUpdates(long minTime, float minDistance, Criteria criteria,LocationListener
    listener, Looper looper);
```

In `onPause()` we should cancel the request and stop the provider:

```
removeUpdates(LocationListener listener)
```

It is also possible to request a single location or an alert of proximity to agiven location expressed in latitude or longitude. `requestSingleUpdate(...)  and addProximityAlert(...)`.

7.4 Location

Locations are delivered to LocationListener

- They bring latitude and longitude;
- The provider that has generated it;
- The time it was generated;

If the provider have the information, it can also contain:

- The altitude of the location;
- The accuracy of the location;
- The bearing if the device is moving
- The speed also if it is moving

- The number of satellites used to obtain the location

Some convenience methods allow:

- To know the bearing from this location to another (geodesic)
- The distance between locations (along a geodesic)
- Obtain a convenient String representation

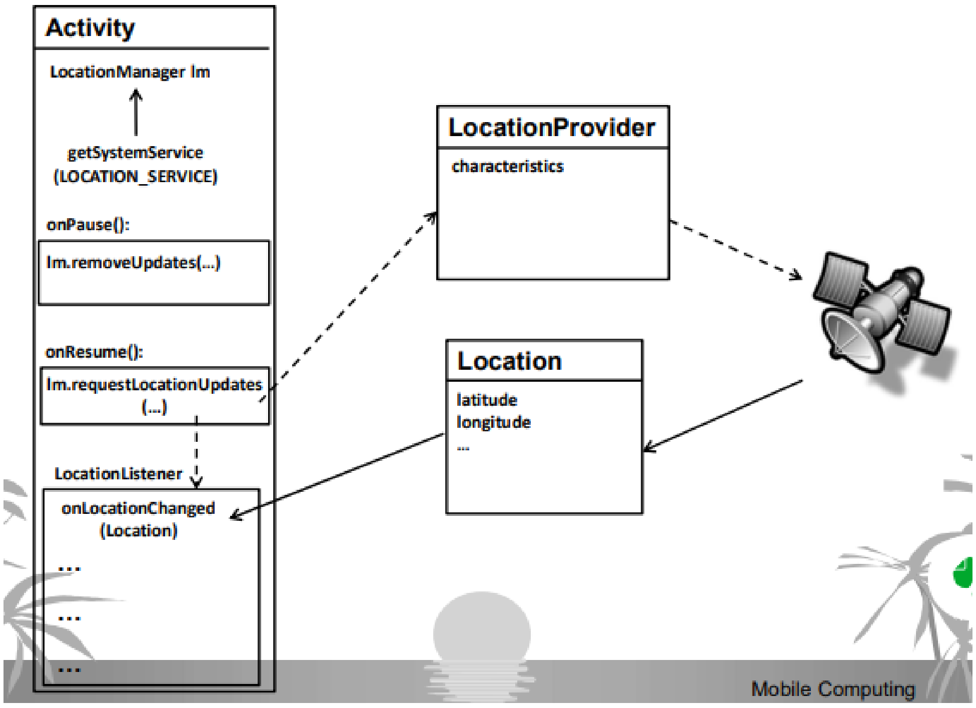


Figure 9: Receiving location information

7.5 Sensors

Android has support for several sensors and a common API to get their measurements. Sensors available include:

- movement sensors;
- orientation sensors;
- environment sensors;

Sensors can be divided into:

- physical sensors giving actual measurements of some quantity;
- synthesized sensors fusing and processing the measurements of other physical sensors to calculate another quantity;

Sensors can be characterized by:

- Range and resolution (minimum, maximum and step value);
- Rate of measurement (nr. of measurements per time unit);
- Power consumption;

7.5.1 Sensor type supported

Android defines a constant for each of the sensor types supported bythe operating system, in the Sensor class. But each device may haveonly a few of those sensors.

physical types

```
movement:
   TYPE_ACCELEROMETER (3D)
   TYPE_GYROSCOPE (3D)
orientation:
   TYPE_MAGNETIC_FIELD (3D)
environment:
   TYPE_AMBIENT_TEMPERATURE
   TYPE_LIGHT
   TYPE_PRESSURE
   TYPE_PROXIMITY
   TYPE_RELATIVE_HUMIDITY
```

synthetized types

```
TYPE_GRAVITY (3D)
TYPE_LINEAR_ACCELERATION (3D)
TYPE_ORIENTATION (3D)
TYPE_ROTATION_VECTOR (3 or 4D)
TYPE_SIGNIFICANT_MOTION (Trigger)
```

7.5.2 Sensor on a device

It's possible for a device to have more than one sensor of the same type. It's uncommon for physical sensors, but can happen with synthesized sensors (more than one implementation).

When an application requests the sensors of a certain type (with `getSensorList(type)`), the system returns an array of sensors.

The `SensorManager` is the starting point and can be obtained from the `Activity` with:

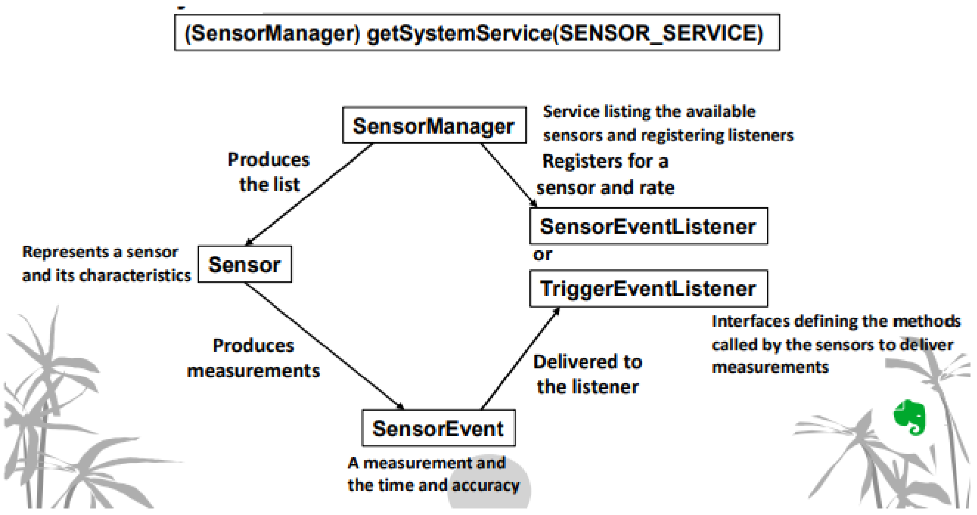

(SensorManager) getSystemService(SENSOR_SERVICE)
SensorManager
Service listing the available sensors and registering listeners
Produces the list
Registers for a sensor and rate
SensorEventListener
or
TriggerEventListener
Represents a sensor and its characteristics
Sensor
Interfaces defining the methods called by the sensors to deliver measurements
Produces measurements
Delivered to the listener
SensorEvent
A measurement and the time and accuracy

Figure 10: Sensor API classes

7.5.3 Movement and Orientation

Movement quantities are presented in the device coordinate system. Orientation measures are in the earth coordinate system (except `TYPE_ORIENTATION` and `getOrientation()` measurements).

`TYPE_ORIENTATION` and `getOrientation()` measurements earth coordinate system.

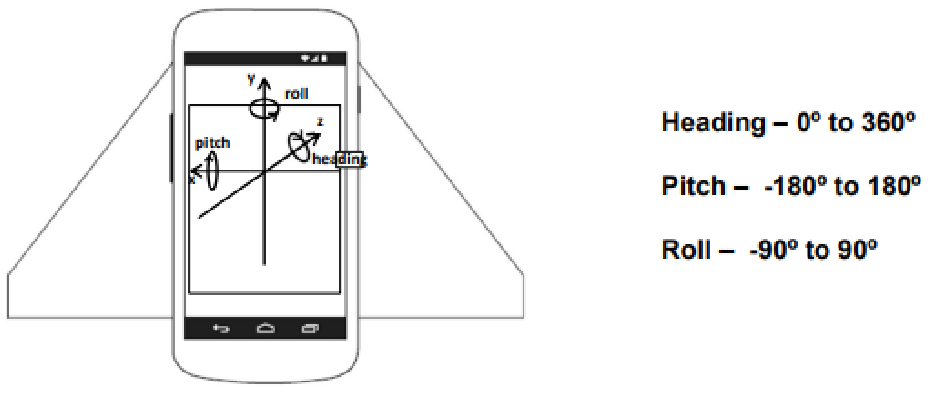


Figure 11: Orientation measurements

7.5.4 SensorManager

Activities get it from the system and it knows the sensors available on the device. We can get a list of all or of a single type of sensors and it's possible to have more than one sensor of a given type (specially of fusion/synthesized sensors).

It can register the `SensorEventListener` for one or more sensors. It defines some measure transformation methods:

- `getRotationMatrixFromVector()` uses the `ROTATION_VECTOR` sensor and computes a rotation matrix;
- `getRotationMatrix()`: computes Inclination and Rotation matrices from gravity and geomagnetic fields
- `getInclination()` (from the Inclination matrix)
- `getOrientation()` (from the Rotation matrix)
- `getAltitude()` From the atmospheric pressure here and at sea level

7.5.5 SensorEventListener

Interface declaring methods (callbacks) where measurements are delivered after registering it for a sensor. Measurements are represented by an instance of `SensorEvent`.

```java
public interface SensorEventListener {
    // The event has sensor, values[], timestamp, accuracy
    void onSensorChanged(SensorEvent event);
    void onAccuracyChanged(Sensor sensor, int accuracy);
}

// The accuracy can be one of SENSOR_STATUS_ACCURACY_HIGH, SENSOR_STATUS_ACCURARY_MEDIUM,
// SENSOR_STATUS_ACCURACY_LOW, SENSOR_STATUS_ACCURARY_UNRELIABLE.

// For sensors producing an event at a time (like the SIGNIFICANT_MOTION detector)
// use the abstract class:
public abstract class TriggerEventListener {
    public void onTrigger(TriggerEvent event);
}
```

7.6 Noise and signal processing

Many sensors produce several kinds of noise:

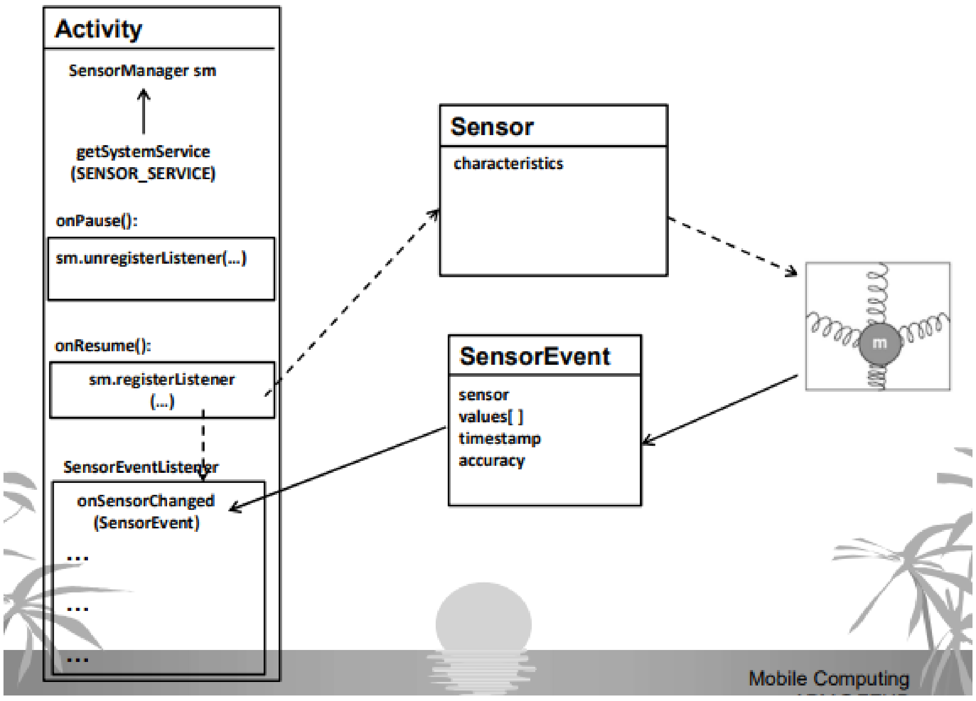


Figure 12: Receiving sensor measurement

- High frequency variations with significant amplitudes
- Low frequency deviations (drifts)

Some simple frequency domain filters are useful:

- **Low pass filters**: Weighted smoothing, simple moving average, simple moving median;
- **High pass filters**: inverse low pass filter;
- **Band pass filters**: Simultaneous low and high pass;
- Kalman filters

8 Lecture 8 - soft keyboard, graphics and media

8.1 Soft keyboard

Devices can have hard keyboards or only a directional pad (arrows plus select).
Many of the soft keyboard properties can be set from the device 'Settings'.
However, `EditText` views can modify the keyboard using the attribute `android:inputType`. It allows different keys (i.e numeric, email, etc.).
Using the attribute `android:imeOptions` allows different bottom-right keys instead of 'return'. Example: Next, Send, Done,...

8.2 Action Events

Pressing the bottom-right key raises the `EditorAction` event. A listener can be defined in `EditText` views with `setOnEditorActionListener()`.
The bottom right menu is a button to show the keyboard, basically.
You can dismiss the keyboard in the handler. By default, the `Done` key does that. Or you can use the code in the handler:

Code to hide keyboard

```
InputMethodManager mgr = (InputMethodManager) getSystemService(INPUT_METHOD_SERVICE);
mgr.hideSoftInputFromWindow(view.getWindowToken(), 0);
```

8.3 2D graphics on the screen - Canvas

To draw in android we use the `Canvas` API. Canvas API is a drawing framework that is provided in Android, with the help of which we can create custom shapes like rectangle, circle, and many more in our UI design.

With the help of this API, we can draw any type of shape for our app. The drawing of the different shapes is done using Bitmap.

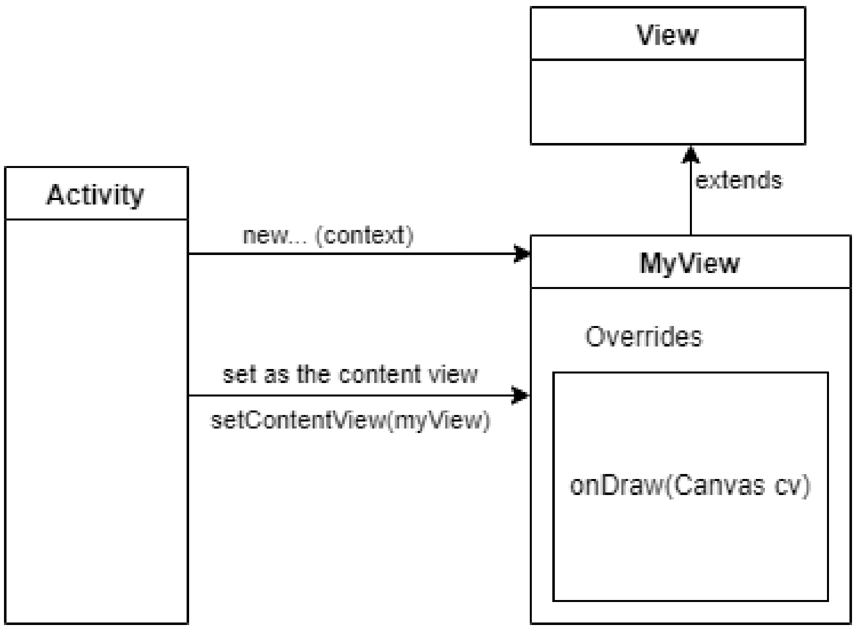


Figure 13: Draw Canvas Setup

The `Canvas` instance defines a lot of primitives. To draw something we use functions like `draw...`. These functions need an instance of `Paint`. It defines the characteristics of the drawings, like color, line style and width, fonts and sizes, etc.

Many geometric shapes are defined through a `Path` instance. They use the `canvas.drawPath()` function.

Othre graphic elements are `Drawable` instances: Bitmaps, Shapes, NinePatches, etc.

Some graphic elements can be defined in xml resources and directly used or 'inflated': Colors, Gradients, Shapes, ...

8.4 Full custom Views

This section is mainly an example of how to draw in views. Full custom Views need to override several methods from the View class.

- They can be used in XML layouts;
- Parameters from the layout are passed in the constructor;
- You can create your own event listeners and property accessors and modifiers;
- You need to override the `onMeasure()` method for proper behavior, when this View is integrated inside a layout;
- You need also to override `onDraw()` with your customized drawing, based on this View properties;

8.4.1 Example

```
public class GraphicsView extends View {
    private static final String QUOTE = "Now is the time for all " +
    "good men to come to the aid of their country.";
    private final Path circle;
    private final Paint cPaint;
    private final Paint tPaint;

    public GraphicsView(Context context) {
        super(context);
        circle = new Path();
        circle.addCircle(150, 150, 100, Direction.CW);
        cPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        cPaint.setStyle(Paint.Style.STROKE);
        cPaint.setColor(Color.LTGRAY);
        cPaint.setStrokeWidth(3);
        tPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        tPaint.setStyle(Paint.Style.FILL_AND_STROKE);
        tPaint.setColor(Color.BLACK);
        tPaint.setTextSize(20f);
        setBackgroundResource(R.drawable.background);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawPath(circle, cPaint);
        canvas.drawTextOnPath(QUOTE, circle, 0, 20, tPaint);
    }
}
```

```
public class Graphics extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new GraphicsView(this));
    }
}
```

background.xml on res/drawable

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
<gradient
    android:startColor="#FFFFFF"
    android:endColor="#808080"
    android:angle="270" />
</shape>
```

8.5 Playing audio

The Android framework encapsulates a complex media player. Can be used through the framework class `MediaPlayer`. It can work asynchronously (playing independently of the application) and works as a state transition machine object.

Supports a lot of audio formats: wav, aac, mp3, wma, amr (speech), ogg, midi.

For a very simple operation, we must call in order:

- `release()`: If the object of the MediaPlayer is not null. It is used to release the resources which are associated with MediaPlayer object.
- `create()`: Will create a media player object. We need to specify the resource ID (in res/raw) or a URI: `mPlayer = MediaPlayer.create(this, R.raw.baitikochi_chuste);`.

- `start()`: To start playing. It returns immediatly.

Creates media player on action

```
...
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    int resId;
    switch (keyCode) {
        case KeyEvent.KEYCODE_F:
            resId = R.raw.f;
            break;
        default:
            return super.onKeyDown(keyCode, event);
    }

    // Release any resources from previous MediaPlayer
    if (mp != null) {
        mp.release();
    }
    // Create a new MediaPlayer to play this sound
    mp = MediaPlayer.create(this, resId);
    mp.start();
    // Indicate this key was handled
    return true;
}
```

```
public class Audio extends Activity {
    private MediaPlayer mp;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        setVolumeControlStream(
        AudioManager.STREAM_MUSIC);
    }
    ...
}
```

xml

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Press the F key"
    />
</LinearLayout>
```

8.6 Playing video

A video inside a file accessible to your application can be played within a `VideoView`.
Formats supported include MP4, H.263 (3GP), H.264 (AVC).
Inform the `VideoView` about the video file path with `setVideoPath()`. Start playing with the `start()` method.
Position is not maintained when the device is rotated. Use `getCurrentPosition()` and save it in `onSaveInstanceState()`.

Restore the video position with the `VideoView seekTo()`.

```
public class Video extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Fill view from resource
        setContentView(R.layout.main);
        VideoView video = (VideoView) findViewById(R.id.video);
        // Load and start the movie
        video.setVideoPath("/mnt/sdcard/samplevideo.3gp" );
        video.start();
    }
}
```

XML

```
    ...
<FrameLayout
    xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <VideoView
        android:id="@+id/video"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_gravity="center" />
</FrameLayout>
```

Manifest

```
...
<activity android:name=".Video"
android:label="@string/app_name"
android:theme=
"@android:style/Theme.NoTitleBar.Fullscreen" >
...
```

8.7 Camera in preview mode

To display video directly from the camera we need a `SurfaceView` in an `Activity` layout. We need also to orchestrate the camera activation with that `SurfaceView` and the `Activity` life-cycle.

Steps

```
    0. Put a SurfaceView in the Activity layout
    1. [In onCreate()]
    get the SurfaceView from the layout (findViewById())
    get a SurfaceHolder from the SurfaceView (save it on variable)
    add a SurfaceHolder.Callback object (with the callbacks) to the SurfaceHolder
    2. [In onResume()]
    open the Camera (static open() method) and save it
    if the camera was already configured go to 4.
    3. [In the surfaceChanged callback (inside the SurfaceHolder.Callback object)]
    setPreviewDisplay( )
    -- configure the camera
    getParameters()
    -- modify some Parameters
    setParameters()
    4. startPreview() (we see the preview in the screen)
    5. [In onPause()]
```

```
stopPreview()
release the Camera (release())
```

8.8 3D graphics in Android

3D graphics are the projection of objects and light on a plane. The plane is the viewport and is mapped to the screen.

The piece of space projected on the viewport is the view frustum (a piece of the pyramidal field of view).

8.8.1 OpenGL

OpenGL is a big library for 3D graphics programming. Independent of graphics hardware, designed in 1992 for graphical workstations, there is a lighter version for mobile devices.

For using OpenGL ES in Android we use a special view derived from `GLSurfaceView`.

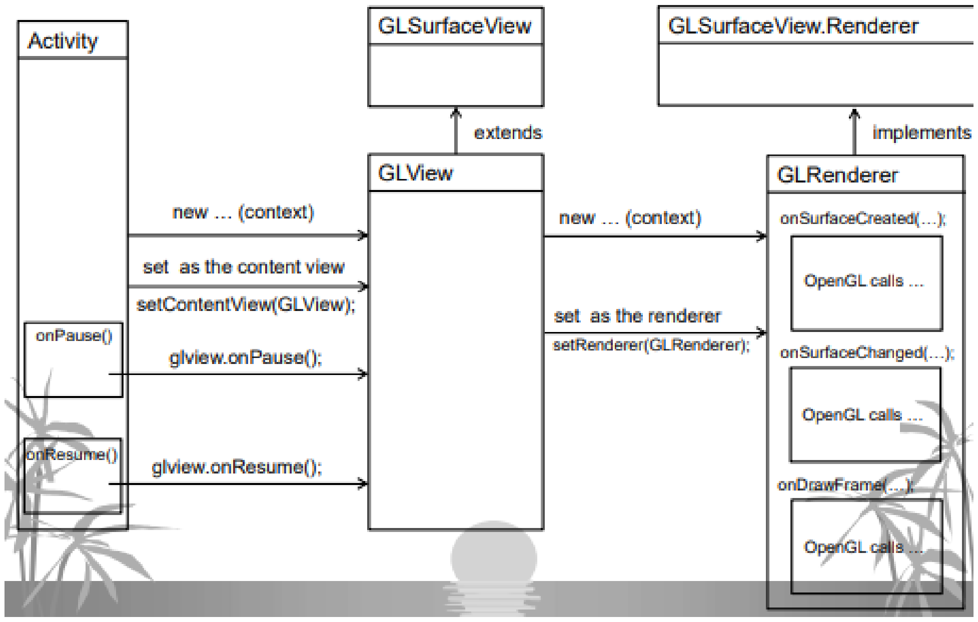


Figure 14: OpenGL diagram

8.9 Touch Events

Many Android devices have only as input the touch screen and gestures.

Many of the events generated by touch are transformed in high level ones like:

- Click
- LongClick
- list item
- select
- key

But we can intercept them at a lower level using the `OnTouchListener` (and its `onTouch()` method). Since we are in the section about graphics, the touch events

The `View` and most of its subclasses generate `onTouch` events. Registered with `setOnTouchListener()`. When the listener is called it receives the `View` that caused it and a `MotionEvent` instance describing it.

8.10 MotionEvent

MotionEvent objects provide information about the touch.

- getAction(): returns in the lower 8 bits a code for theaction: DOWN, UP, MOVE, OUTSIDE, ...
- In the higher 8 bits it gives a 'finger' number startingwith 0 (in and after Android 2.2 multitouch issupported).
- getPointerCount() returns the number of active 'fingers'.
- getPointerId(i), getX(i), getY(i) allows us to extract thenumber and position of each active 'finger'.

Long touch events

```
private void dumpEvent(MotionEvent event) {
    String names[] = { "DOWN" , "UP" , "MOVE" , "CANCEL" , "OUTSIDE" ,
    "POINTER_DOWN" , "POINTER_UP" , "7?" , "8?" , "9?" };
    StringBuilder sb = new StringBuilder();
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    sb.append("event ACTION_" ).append(names[actionCode]);
    if (actionCode == MotionEvent.ACTION_POINTER_DOWN || actionCode ==
         MotionEvent.ACTION_POINTER_UP) {
        sb.append("(pid " ).append(action >> MotionEvent.ACTION_POINTER_ID_SHIFT);
        sb.append(")" );
    }
    sb.append("[" );
    for (int i = 0; i < event.getPointerCount(); i++) {
        sb.append("#" ).append(i);
        sb.append("(pid " ).append(
        event.getPointerId(i));
        sb.append(")=" ).append((int) event.getX(i));
        sb.append("," ).append((int) event.getY(i));
        if (i + 1 < event.getPointerCount())
            sb.append(";" );
        }
        sb.append("]" );
        Log.d(TAG, sb.toString());
    }
```

Results

```
event ACTION_DOWN[#0(pid 0)=135,179]
event ACTION_MOVE[#0(pid 0)=135,184]
event ACTION_MOVE[#0(pid 0)=144,205]
event ACTION_MOVE[#0(pid 0)=152,227]
event ACTION_POINTER_DOWN(pid 1)[#0(pid 0)=153,230;#1(pid 1)=380,538]
event ACTION_MOVE[#0(pid 0)=153,231;#1(pid 1)=380,538]
event ACTION_MOVE[#0(pid 0)=155,236;#1(pid 1)=364,512]
event ACTION_MOVE[#0(pid 0)=157,240;#1(pid 1)=350,498]
event ACTION_MOVE[#0(pid 0)=158,245;#1(pid 1)=343,494]
event ACTION_POINTER_UP(pid 0)[#0(pid 0)=158,247;#1(pid 1)=336,484]
event ACTION_MOVE[#0(pid 1)=334,481]
event ACTION_MOVE[#0(pid 1)=328,472]
event ACTION_UP[#0(pid 1)=327,471]
```

8.11 Higher level gesteures

The onTouch listener can pass the MotionEventdata to gesture detectors (Android has two).

- **GestureDetector**: Can detect and trigger events corresponding to one finger gestures. Down, Fling, LongPress, Scroll, ShowPress, SingleTap, DoubleTap.
- **ScaleGestureDetector**: Detects the pinch two finger gesture. Generates three events during the gesture: ScaleBegin, Scale, ScaleEnd.

9 Lecture 9 - Introduction to flutter

9.1 Widgets

All the UI in flutter is made by a tree of Widgets.
Almost all widgets are StatelessWidgets or StatefulWidgets.

- immutable;
- constructor calls the `build()` method, that returns the widget;
- `build()` cannot be called again;
- new instance must be created in order to redraw the widget;

Widgets receive a `BuildContext` in the `build()` method . It is a reference to the location of a Widget in the UI tree and it can contain properties concerning the widgets rendering.

9.2 StatefulWidgets

- Have an associated state object;
- State object is mutable and redraws the immutable widget through the `build()` method;
- The `StatefulWidget` derived class should override at least the `createState()` method, that returns the associated state object ;
- The associated `State` class should override the `build()` method that returns the `Widget` (created the first time or redrawn).

```
class MyWidget extends StatefulWidget {
    @override
    _MyWidgetState createState() => _MyWidgetState();
    }

    class _MyWidgetState extends State<MyWidget> {
    sometype value = initvalue;

    @override Widget build(BuildContext context) {
        return Container( ...
        // the UI of this widget
        );
    }
}
```

9.2.1 Stateful lifecycle

- `createState()`: Is immediately called after construction;
- `initState()`: Called after creation if overridden;
- `build()`: To create or redraw a widget tree dependent on the state Automatically called if state changes (using `setState()` or `didUpdateWidget()`);
- `setState()`: Should be called with a function parameter that changes the state and makes a rebuild.

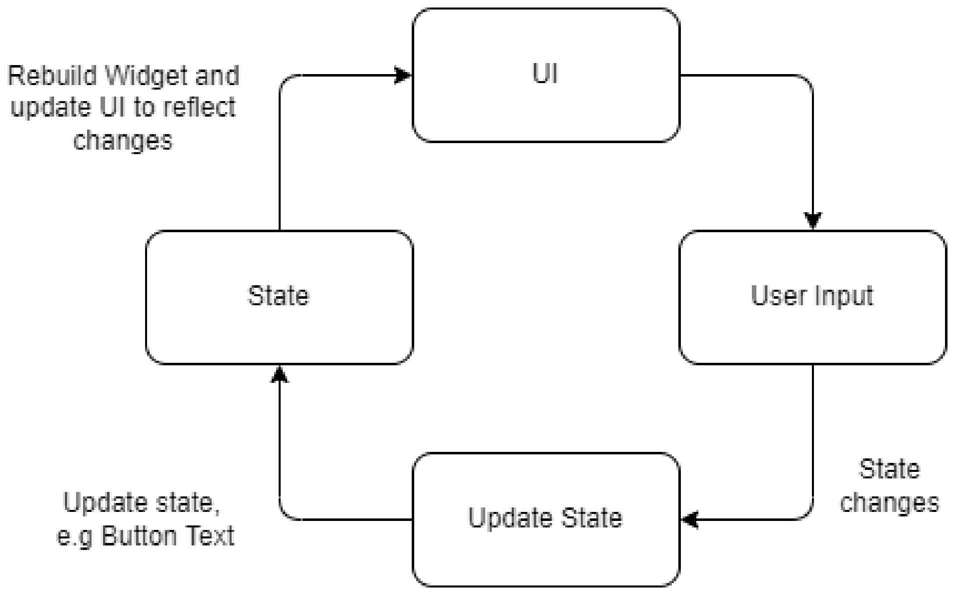

UI
Rebuild Widget and update UI to reflect changes
State
User Input
State changes
Update State
Update state, e.g Button Text

Figure 15: Stateful Widget lifecycle

Advantages of stateful widgets over android API

Supposing that you want to "show a collection of items to a user", or show a list that might be updated, in android API, you would have to use an `ArrayAdapter` or `BaseAdapter`. As explained in the **Basic Android** chapter, if we want to draw a specific view of the list, we must override the ArrayAdapter and create our own view.
In Flutter, this task is simplified, by using the `StatefulWidgets`, the collection/list of elements is treated as another normal Widget. Everytime the information inside the Widget is updated, it's redrawed once again. No overrides are necessary and the code is simplified.

10 Lecture 10 - Flutter pages, navigation, drawing and external calls

10.1 Pages

Page or screen is a Widget tree aka a Route.
The App widget defines the home route and others, from Home, routes, and initialRoute properties (on the App constructor).
The home is the initial screen (widget tree) shown. Alternatively, a set of routes (pages) can be defined and an initialRoute specified.

10.2 Navigation

The Navigator widget allows the replacement of a page by another using a stack discipline. It is possible to create a set of navigation routes previously in the app, or build each one when we want to navigate to it. Initially the App has already a Navigator. We can use it or create a new independent one with new routes.
The Navigator defines a set of static methods that manipulate the stack of pages (routes):

- `push(context, Route)` and `pop(context)`
- `pushName()`, `removeRoute()`, `replace()`,...

10.2.1 Prebuild route table

It's possible to prebuild a route table in the App widget and put it in the routes property:

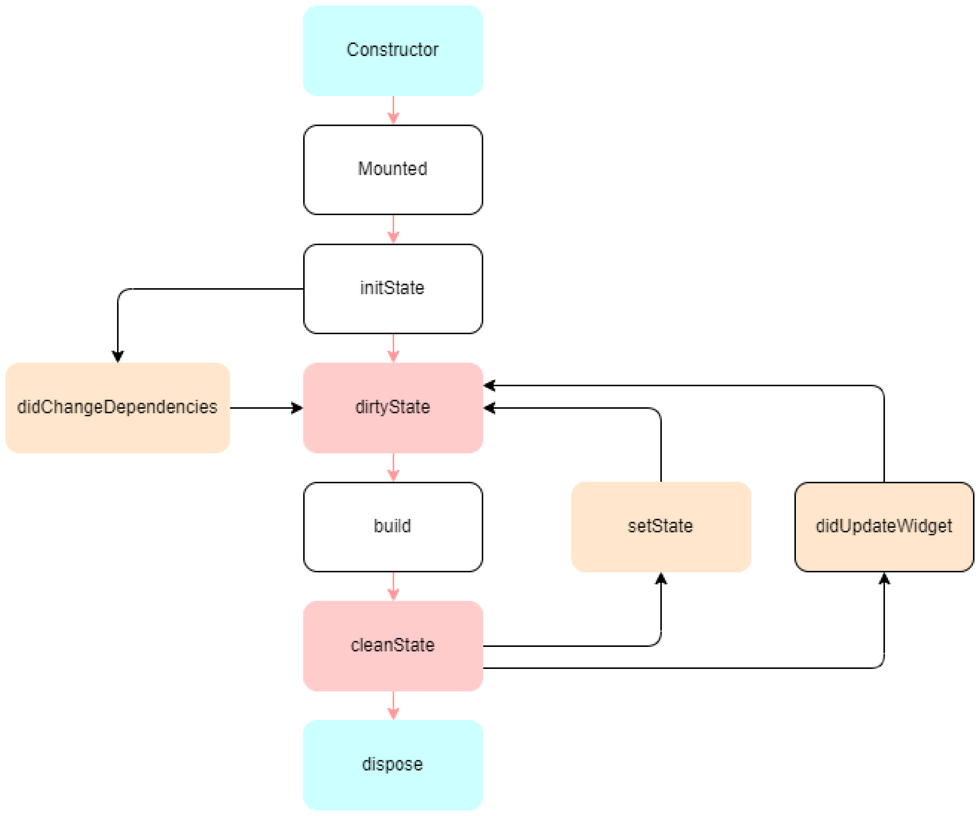


Figure 16: State object lifecycle

```
void main() {
    runApp(MaterialApp(
        home: MyAppHomePage(), // becomes the route named '/'
        routes: {
        '/a': (BuildContext context) => MyPage1(title: 'page A'),
        '/b': (BuildContext context) => MyPage2(title: 'page B'),
        '/c': (BuildContext context) => MyPage3(title: 'page C'), }
    ));
    }
```

10.2.2 Create pages dynamically

New screens or pages (widgets trees) can be built from a PageRoute (derived from) e.g., the MaterialPageRoute, requires a WidgetBuilder (a function) that builds the tree from context.

```
Navigator.push(context,
MaterialPageRoute(builder: (context) => MyPage1(title: "page A")));
```

10.2.3 Navigation forward and backward

Navigate forward

```
// within a widget
...
Navigator.pushNamed(context, '/b');
```

Navigate backward

```
// within the navigated page
...
Navigator.pop(context);
```

10.2.4 Passing data to a route

Class to be passed

```
class ScreenArguments {
   final String title;
   final String message;
   ScreenArguments(this.title, this.message);
}
```

Class to create page

```
class HomePage extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
      return ...
   ...
   onPressed: () {
      Navigator.pushNamed(context, '/page2',
      arguments: ScreenArguments('Title', 'Message'));
   }
   ...
   }
}
```

Page

```
// App root widget
...
return MaterialApp(
   routes: { '/page2': (context) => SecondPage() },
   home: HomePage() // route name: '/'
)
...
class SecondPage extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
      final ScreenArguments args = ModalRoute.of(context).settings.arguments;
      return ...
         ... args.title ...
         ... args.message ...
   }
}
```

10.2.5 Return a value using pop

When we do a `Navigator.pop(context)`, `pop()` accepts a second parameter that is a result. It can be of any type.

This result is returned by the `Navigator.push(...)` that created the popped page. The result comes in a `Future`, resolved only when the page is popped.

Example

```
// caller
final result = await Navigator.push(context, MaterialPageRoute(builder: (context) =>
    SomePage()));
```

```
// New page - SomePage
Navigator.pop(context, 'Some message');
```

10.3 External HTTP Calls

Flutter does not have a direct API for web requests. But there are several packages in the package repository (pub.dev).

One of the simplest is the http package (https://pub.dev/packages/http) For using external packages, it is necessary to import them in code, and to declare them in the project pubspec.yaml file. Finally, it must be downloaded.

The declaration and installation can be done from the command line, or from the used IDE:
```
$ flutter pub add http
```

For example, to define a function to perform a REST GET request:

```
import 'package:http/http.dart';
...
Future<String> getResponse() async {
final response = await http.get(Uri.http('data.fixer.io', '/api/latest',
    { 'access_key': '<your API key>',
        'base': 'EUR',
        'symbols': 'USD,GBP'
    }
));
if (response.statusCode == 200)
    return response.body;
else
    throw Exception('HTTP failed');
}
```

11 Lecture 11 - Flutter app architecture

11.1 MVC's architecture

The MVC architecture holds:

- **Models** - Holds the application state and simple operations to access/filter/manipulate that data. These operations are also called the Domain Logic.
- **Views** - the UI and pages build with the Flutter widgets. This layer may also contain the "view controllers" with handlers for local interactions.
- **Controller** - contains the Application Logic composed by the business operations that constitute the functionalities of the application. These operations can be organized in **Commands** invoked by the other layers, but mainly is result of UI interactions.
- **Services** - contain the operations in the outside world. Usually, they are invoked by Commands that can retrieve data and inject it on Models, or in the other direction.

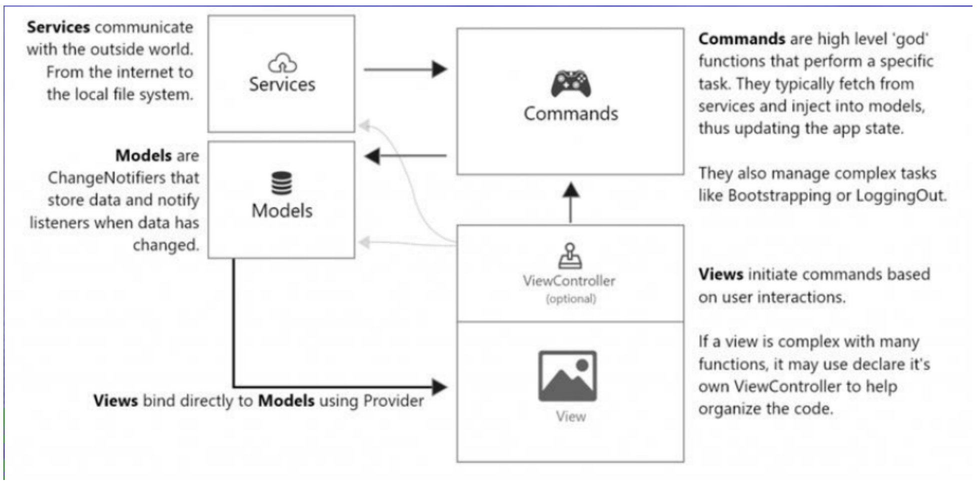

Services communicate with the outside world. From the internet to the local file system.
Services
Commands
Commands are high level 'god' functions that perform a specific task. They typically fetch from services and inject into models, thus updating the app state.
They also manage complex tasks like Bootstrapping or LoggingOut.
Models are ChangeNotifiers that store data and notify listeners when data has changed.
Models
ViewController (optional)
Views initiate commands based on user interactions.
If a view is complex with many functions, it may use declare it's own ViewController to help organize the code.
View
Views bind directly to Models using Provider

Figure 17: MVC diagram

The implementation of this architecture requires access to the `Models` in every place in the App. For doing that, a special widget is provided in an external package. It associates `Models` and `Services` with the application context. The package is called `Provider`.

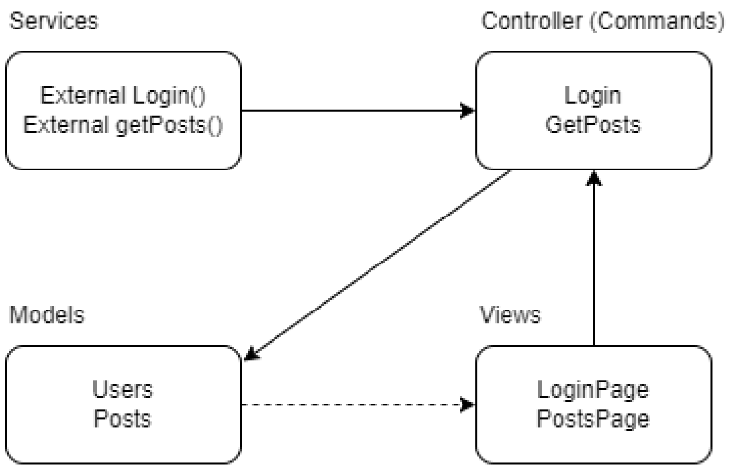


Figure 18: MVC diagram example

11.2 Redux Architecture

The Redux architecture specifies that all the application state is centralized in the Store block (it can contain several Models). is a state management architecture library that successfully distributes data across widgets in a repetitive manner. It manages the state of an application through a unidirectional flow of data.

In this example, data generated in the main widget is needed in sub-widget 8. Ordinarily, this data passes through sub-widget 2 to sub-widget 6 and then, finally, it reaches sub-widget 8. This is also the case for widgets that need data generated or saved in the state of any widget that's higher up in the hierarchy.

With Redux, you can structure your application so that the state is extracted in a centrally-located store. The data in this centralized store can be accessed by any widget that requires the data, without needing to pass through a chain of other widgets in the tree.

Any widget that needs to add, modify, or retrieve the data in a state managed by the Redux store would have to request it with the appropriate arguments.

Likewise, for every change made to the state, the dependent widgets respond to the change either through the user interface or any other configured means.

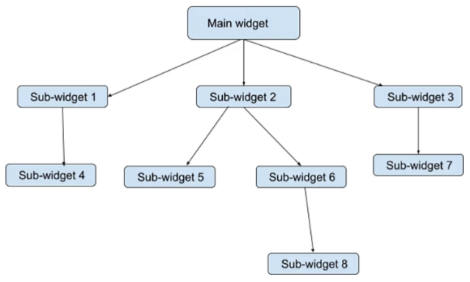


Figure 19: An unidirectional flow of data

11.2.1 Why is it important?

In a medium or large-scale application with many widgets, when a child widget needs data, it is common to manage the data from the main.dart file.

This data could be distributed through the constructors of widgets as arguments until the data gets to the recipient widget, this could lead to a long chain of data transfer through widgets that don't need this data.

Not only can it be cumbersome and difficult to pass data through the constructors, it can also affect the performance of an application. This is because when you manage data from the main widget - or any root widget - the entire widget tree rebuilds whenever a change occurs in any of its child widgets. You only want to run the build method in the widget that requires the changed data.

11.2.2 How it works

In Redux there's a `Store` which holds a `State` object that represents the state of the whole application. Every application event (either from the user or external) is represented as an `Action` that gets dispatched to a `Reducer` function. This `Reducer` updates the `Store` with a new `State` depending on what `Action` it receives. And whenever a new `State` is pushed (the state is immutable, thus it can't be modified, just replaced) through the `Store` the `View` is recreated to reflect the changes.

With Redux most components are decoupled, making UI changes very easy to make. In addition, the only business logic sits in the `Reducer` functions. A `Reducer` is a function that takes an `Action` and the current application `State` and it returns a new `State` object, therefore it is straightforward to test because we can write a unit test that sets up an initial State and checks that the `Reducer` returns the new and modified `State`.

11.2.3 Middleware

what happens when the application has to perform some asynchronous operation, such as loading data from an external API? This is why people came up with a new component known as the `Middleware`.

`Middleware` is a component that may process an `Action` before it reaches the `Reducer`. It receives the current application `State` and the `Action` that got dispatched, and it can run some code (usually a side effect), such as communicating with a 3rd-party API or data source. Finally, the `Middleware` may decide to dispatch the original `Action`, to dispatch a different one, or to do nothing more.

11.2.4 Resume according to slides

The `View` (flutter widget tree) generates `Actions` from user interactions, that specify external requests and/or state modifications. The external requests are filtered out by a `Middleware` block. The other `Actions` go to a `Reducer` function.

The `Reducer` function modifies the Store, according to the specified `Action`. `Views` have access to the `Store`, and whenever this is modified it triggers a `View` redrawing, taking into account the app state inside the `Store`.

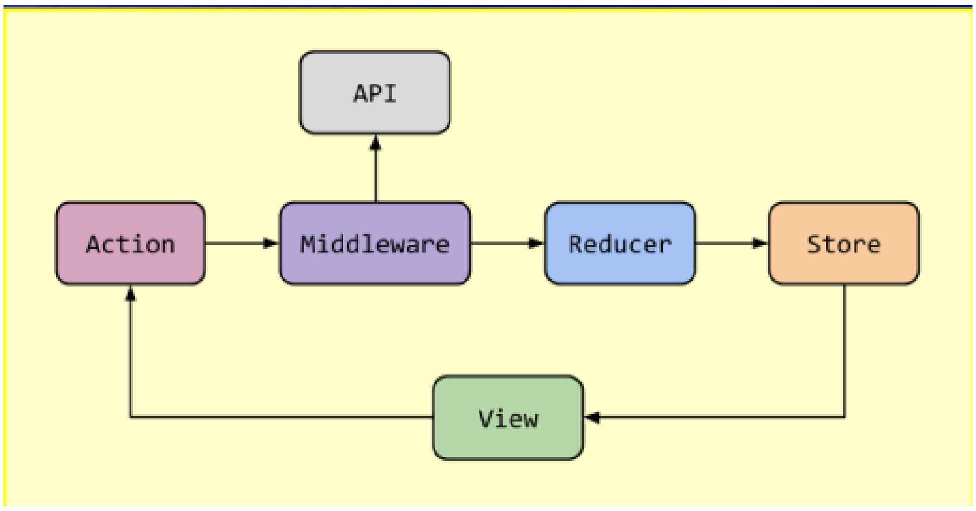


Figure 20: Redux architecture diagram

12 Lecture 12 - Flutter native channels

We can use a function called `invokeMethod('method_name', parameters)` to call native functions from a specific system.

```
const channel = MethodChannel('API_name');
final ResultType response = await channel.invokeMethod('method\_name', parameters);
```

The channel can also define Callback methods to be called from native code.

12.1 Catching native events

Can be done using an `EventChannel` between the Dart side and the native side.

On the Dart side a receiver should be established defining callbacks called when an event is generated in the native side.

The native side establishes a `StreamHandler` attached to the channel, together with an `EventSink`, capable of generating events that can carry data. It's possible to also generate error events.

The appropriate Dart side callback is called whenever an event is generated.

13 Lecture X - MVVM

Model-view-viewmodel (MVVM) is a software that helps to cleanly separate business and presentation logic of an application from the its user interface (UI), making it easier to test, maintain and evolve. It can also code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts in the app. This is a famous pattern and used in Xamarim and it is similar to the **MVP** (Model View Presenter) pattern, however the drawbacks of the MVP has been solved.

13.1 The MVVP Pattern

The pattern is divided in three parts: the model, the view and the view model.

At a high level, the view "knows about" the view model, and the view model "knows about" the model, but the model is unaware of the view model, and the view model is unaware of the view. Therefore, the view model isolates the view from the model, and allows the model to evolve independently of the view.

The separate code layers of MVVM are:

- **Model**: this layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

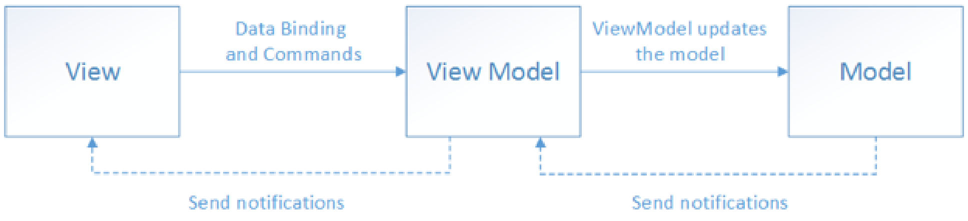


Figure 21: The MVVM pattern

- **View**: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.
- **ViewModel**: It exposes those data streams which are relevant to the View. Moreover, it serve as a link between the Model and the View.

13.2 Advantages and disadvantages

Advantages:

- Enhance the reusability of code.
- All modules are independent which improves the testability of each layer.
- Makes project files maintainable and easy to make changes.

Disadvantages:

- This design pattern is not ideal for small projects.
- If the data binding logic is too complex, the application debug will be a little harder.

13.3 Implementation and DataBinding

There are 2 ways to implement MVVM design pattern in Android projects:

- Using the DataBinding library released by Google.
- Using any tool like RxJava for DataBinding.

Google releases the Data Binding Library for Android that allows the developers to bind UI components in the XML layouts with the application's data repositories. This helps in minimizing the code of core application logic that binds with View. Further, Two - way Data Binding is done for binding the objects to the XML layouts so that object and the layout both can send data to each other.

14 Resources

IOC: https://www.sitepoint.com/three-design-patterns-that-use-inversion-of-control/
Threads: https://www.oreilly.com/library/view/efficient-android-threading/9781449364120/ch04.html
Audio Media: https://www.tutlane.com/tutorial/android/android-audio-media-player-with-examples
Canvas API: https://www.geeksforgeeks.org/how-to-use-canvas-api-in-android-apps/
Xamarim MVVM: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm
Xamarim MVVM: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/
Reducer: https://blog.logrocket.com/flutter-redux-complete-tutorial-with-examples/