

UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA



FACOLTÀ DI INGEGNERIA INFORMATICA
CORSO DI MOBILE PROGRAMMING

Relazione progetto

“WORLDBANK”

Gruppo FPR-IOL2018

formato da:
Ferraro Daniele
Rufini Alessio
Petricca Geremia

Anno accademico 2018/2019

Indice dei contenuti

1 SCOPO DEL DOCUMENTO.....	1
2 SPECIFICHE PROGETTUALI.....	1
3 ANALISI DEI REQUISITI.....	3
3.1 RICERCA DATI.....	3
3.1.1 Caso d'uso richiesto.....	4
3.1.2 Casi d'uso eccezionali.....	8
3.2 CARICAMENTO DATI.....	9
3.2.1 Caso d'uso richiesto.....	10
3.2.1 Casi d'uso eccezionali.....	12
3.3 CONSIDERAZIONI SUI REQUISITI.....	13
4 SCELTE PROGETTUALI ED IMPLEMENTATIVE.....	14
4.1 MULTI-THREAD:.....	14
4.2 ACQUISIZIONE DATI REMOTI.....	14
4.2.1 Connessione al server World Bank Data.....	15
4.2.2 Utilizzo GSON.....	15
4.3 GESTIONE DELLE NOTIFICHE.....	18
4.3.1 Mancanza di connessione a Internet.....	18
4.3.2 File JSON non valido.....	20
4.3.3 Gestione delle dialog.....	22
4.4 GESTIONE CICLO DI VITA DELLE ATTIVITÀ.....	25
4.4.1 Callback del ciclo di vita dell'activity corrente.....	25
4.4.2 Meccanismo StartActivity.....	28
4.4.3 Gestione Database.....	30
4.4.4 Model-View-Controller.....	32
4.5 GESTIONE GRAFICO.....	33
4.5.1 Utilizzo Libreria esterna MPChartAndroid.....	33
4.5.2 Salvataggio Grafico in formato PNG.....	34
4.6 GESTIONE MENÙ PAGINA PRINCIPALE.....	35
4.7 LOCALIZZAZIONE.....	37
4.8 GRAFICA.....	39
5 PIATTAFORMA, SVILUPPO E TESTING.....	43

1 SCOPO DEL DOCUMENTO

Il fine di questo documento è descrivere il processo di sviluppo dell'applicazione WorldBank. Analizzeremo le specifiche del progetto per arrivare alla definizione delle scelte progettuali e implementative, intraprese per soddisfare i requisiti richiesti. L'ultima sezione riguarda la piattaforma utilizzata per eseguire lo sviluppo e il testing dell'app.

L'App WorldBank si pone come obiettivo principale quello di:
“permettere all'utente, attraverso una interfaccia semplice, chiara e diretta, di ottenere i dati storici di un certo Paese, riguardanti uno specifico indicatore di un determinato argomento”.

2 SPECIFICHE PROGETTUALI

Il sito della WorldBank (<http://www.worldbank.org/>) mette a disposizione 1 servizio di tipo REST x l'acquisizione di dati storici riguardanti molte variabili (indicatori) provenienti da molteplici sorgenti (sources) e divise x argomenti (topics).

L'entry point del sito REST di WB è <https://data.worldbank.org/> dove è possibile leggere le istruzioni necessarie x scaricare le pagine (in vario formato tra cui json).

In tutto, ad oggi ci sono:

- 16695 indicatori;
- 21 argomenti;
- 43 sorgenti;
- 304 paesi (compresi anche gli aggregati)

- *L'App dovrà mostrare all'avvio una schermata che dia all'utente la possibilità di eseguire la scelta per:*
 - *Paese -> argomento -> indicatore*
 - *Argomento->indicatore->paese*
 - *Mostra immagine salvata precedentemente*
 - *Carica dati offline*
- *Finita la fase di scelta (che dovrà essere effettuata tramite alcune activity o fragment in successione) dovrà essere mostrato un grafico a linee (usare la libreria MPAndroidChart presente nell'Arsenal di Android: <https://android-arsenal.com/>)*
- *L'app dovrà inoltre su richiesta dell'utente:*
 - *Salvare localmente il grafico in formato png*
 - *Salvare localmente i dati scaricati in un database per essere successivamente ricaricati offline*

3 ANALISI DEI REQUISITI

Possiamo suddividere logicamente le specifiche del progetto in 2 blocchi:

- **RICERCA E SELEZIONE** di paesi, argomenti e indicatori, fino al raggiungimento dell'obiettivo: ottenere le informazioni sull'indicatore di un paese desiderato per visualizzarne il relativo grafico e offrire la possibilità di salvare tali informazioni; sia come File.png rappresentante il grafico, che come record del database contenente tutti i dati dell'indicatore.
- **CARICAMENTO DATI** precedentemente salvati alla fine della fase di ricerca. Offrire all'utente la possibilità di selezionare e visualizzare i dati di un indicatore salvato nel database, e/o visualizzare l'ultimo indicatore salvato come immagine.png (grafico).

Ad esse **aggiungiamo** inoltre i **seguenti requisiti** che l'app deve soddisfare per essere **user-friendly** e gradevole:

- ogni schermata dovrà mantenere lo stesso stile e proporzioni nelle diverse dimensioni degli schermi, e in aggiunta deve offrire l'orientamento sia in portrait che in landscape per permettere all'utente di avere una prospettiva diversa dei contenuti.
- offrire la possibilità di cancellare i singoli indicatori salvati nel database
- mostrare un appBar che aiuti l'utente a navigare attraverso le activity (freccietta indietro in aggiunta al pulsante Back nativo e action Home) e includa un menù con interessanti link per comprendere l'obiettivo e il lavoro fatto dall'organizzazione WorldBank e una semplice guida sull'utilizzo dell'App.

3.1 RICERCA DATI

PREMESSA

Dopo un attento studio del sito WorldBank e della sezione interna Worl Bank Data è noto che: il sito della WorldBank mette a disposizione 1 servizio di tipo **REST**(Representational State Transfer): architettura software per sistemi distribuiti basato su HTTP x l'acquisizione di dati, a cui si può accedere tramite un identificatore globale URI. Il funzionamento prevede una struttura degli URL ben definita atta a identificare univocamente una risorsa (fonte di informazioni) o un insieme di risorse,

Le componenti di una rete (client e server) comunicano attraverso 1 interfaccia standard quale HTTP e si scambiano *rappresentazioni* di queste risorse (il documento che trasmette le informazioni).

In particolare le risorse che a noi interessano messe a disposizione dal Worl Bank Data sono informazioni su: Paesi, Argomenti, Indicatori e Indicatore per paese.

Un **indicatore** è una serie temporale di valori: rappresenta dati come total population, gross national income, energy use e molti altri, relativi ad un determinato paese.

Gli indicatori sono divisi per **argomenti**: categorie di alto livello in cui tutti gli indicatori sono mappati; Agriculture & Rural Development, Education, and Trade sono esempi di argomenti.

La relazione tra **paesi** e indicatori è molti a molti.

L'API degli indicatori supporta 2 metodi equivalenti di base (restituiscono gli stessi dati) x creare query/richieste: 1 struttura basata su URL e 1 struttura basata su argomenti.

Scegliamo di utilizzare le API basate su URL perché a nostro parere più intuitive.

Tra le svariate API messe a disposizione da Worl Bank Data a noi serviranno, ed **utilizzeremo, le seguenti**:

- http://api.worldbank.org/v2/country/?format=json&per_page=500 : ottiene una lista di dati di paesi e specifica il set di risultati da restituire per pagina (per essere sicuri di ottenere una sola pagina, ovvero un solo file contenente tutti i dati di tutti i paesi)
- <http://api.worldbank.org/v2/topic>: restituisce una lista di tutti gli argomenti
- <http://api.worldbank.org/v2/topic/5/indicator?format=json> restituisce una lista di tutti gli indicatori sotto uno specifico argomento
- <http://api.worldbank.org/v2/country/br/indicator/NY.GDP.MKTP.CD?format=json>: recupera i dati degli indicatori relativi ai paesi (ad esempio, il precedente è una richiesta per i dati sul PIL del Brasile)

3.1.1 Caso d'uso richiesto

SPECIFICHE REQUISITO: l'utente ha intenzione di ricercare nel database online un indicatore di un paese, per visualizzarlo, piuttosto che cercarlo e caricarlo offline.

SCOPO REQUISITO: aiutare l'utente nel percorso di ricerca dell'indicatore e mostrare poi tale indicatore selezionato come grafico a linee.

Da qui infine, dare la possibilità di salvare tale indicatore.

La figura 1 mostra il **principio di funzionamento** a grandi blocchi del **requisito di ricerca**.

Nello specifico la figura mostra la ricerca per Country; tuttavia la ricerca per Topics è analoga. (la parte relativa al salvataggio dei dati verrà descritta in seguito e non illustrata).

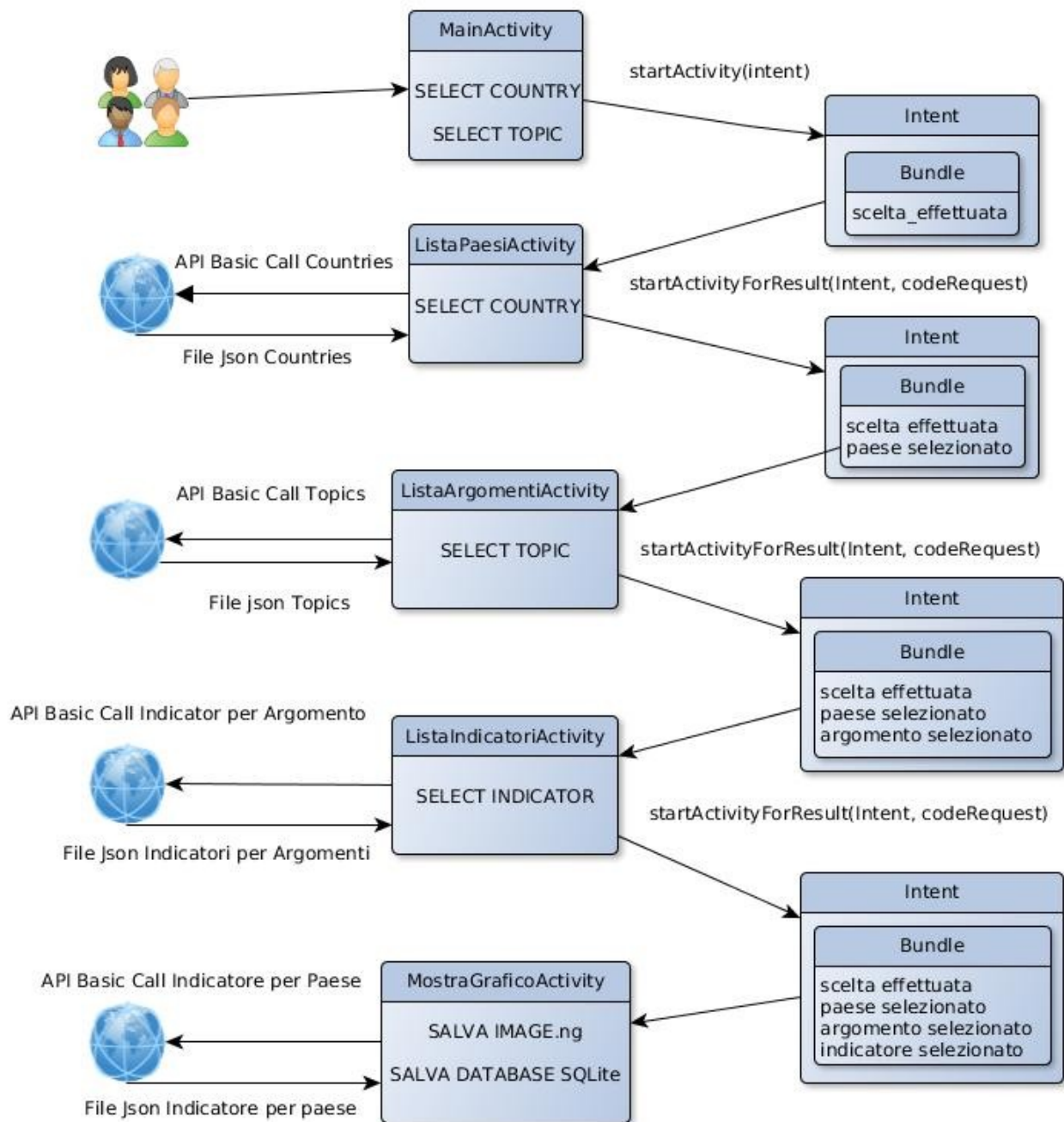


Fig. 1. FLUSSO ESECUZIONE LINEARE: ricerca e visualizzazione dell'indicatore come grafico a linee; opionalmente salvataggio dei dati contenuti dall'indicatore e del rispettivo grafico.

L'app, **indipendentemente** che l'utente abbia scelto di **effettuare la ricerca per Paese o per Argomento** dovrà, in base alle scelte effettuate (sempre dall'utente) durante il percorso di selezione, usufruire del servizio REST offerto per **scaricare e mostrare all'utente le informazioni utili per il raggiungimento dell'obiettivo**:

- l'app accederà direttamente (senza autenticazione) ai dati e li acquisirà **tramite chiamate alle API Basic** : l'API degli indicatori della banca mondiale fornisce l'accesso programmatico a questi dati e supporta entrambi i protocolli "http" e "https". Tra i vari formati di rappresentazione del documento, scegliamo di scaricare i **file** (i documenti forniti) in **formato JSON** perché compatto e facilmente comprensibile all'uomo.
- **ogni attività** (eccetto il main) **appena avviata dalla precedente** si **connette** al database della World Bank Data e **scarica il file (di testo) json appropriato**, ovvero il file contenente le informazioni che l'attività deve mostrare all'utente per aiutarlo nel percorso di scelta dell'indicatore.
Per esempio se l'utente sceglie di ricercare l'indicatore iniziando ricercando il paese da selezionare, l'attività successiva (avviata dal main) scaricherà il file json contenente la lista dei paesi presenti nel database online per mostrarla all'utente e permettergli di selezionarne uno a piacimento.
Identico discorso per le altre attività che collaborano al raggiungimento del requisito di ricerca.
- Ogni attività poi, tra tutte le informazioni contenute nel file scaricato, **estrae solo quelle che realmente che gli interessano**, ovvero solo le informazioni che ritiene necessarie mostrare all'utente per soddisfare il suo scopo.
Ad esempio il file Json contenente la lista dei paesi, per ogni paese contiene diverse informazioni, tra cui nome, identificatore univoco, livello di reddito, capitale ed altri; mentre all'attività serviranno solo alcune di queste informazioni per ogni paese.
- Seguendo il consiglio del Prof. Regoli l'app utilizza la libreria esterna **GSON** realizzata da GOOGLE: essa permette effettuare un mapping 1-a-1 tra gli elementi del file Json (object o array di object) e gli oggetti di un POO language come Java in modo sufficientemente rapido e conciso (eccetto casi particolari di file Json e/o invii da parte del server di file Json non analoghi tra loro).
- Quindi, una volta effettuato il **parsing** del file Json con GSON, l'attività avrà i suoi oggetti che rappresentano gli elementi del file Json d'interesse, ovvero rappresentano i dati e le informazioni del documento acquisito da World Bank Data.
Successivamente li **manipola a piacimento per presentarli all'utente** attraverso la propria UI, le proprie View, Widget, ascoltatori e adattatori.
In particolare, le attività di selezione del percorso avranno una **ListView** che mostrerà i dati in modo personalizzato grazie all'appropriato **ArrayAdapter<ElementoGenerico>**: il quale a runtime tramite inflazione nel suo metodo getView() (da noi sovrascritto) passerà appunto alla ListView le varie View contenenti i dati desiderati.
- **L'utente selezionerà poi l'elemento desiderato** per proseguire la ricerca.
L'attività registra la scelta dell'utente nel metodo

*onItemClick(AdapterView<?> parent, View view, **int** position, **long** id)*

e **lancia la successiva** e appropriata attività passagliandi (attraverso l'intent di avvio) la nuova informazione (selezione corrente dell'utente) più le precedenti informazioni sul percorso effettuato dall'utente, accumulate dalle attività precedenti alla corrente (selezioni precedenti dell'utente).

- L'**ultima attività**, riceverà quindi nel bundle (passatogli dall'attività che l'ha avviata) **tutte le informazioni necessarie** per costruire l'API che gli permetterà di scaricare l'indicatore del paese desiderato.
Scaricato tale file, anche lei utilizza GSON per ottenere i suoi oggetti.
Tali oggetti questa volta però vengono manipolati diversamente: per mostrare un grafico a linee costruito con la libreria MPAndroidChart come da specifiche e descritto di seguito nel prossimo capitolo.
- Infine l'ultima attività offrirà la possibilità di **salvare il grafico** appena costruito in formato PNG, e i tutti i dati contenuti nell'indicatore, nel database **SQLite locale**.
In entrambi i casi scriverà i dati **localmente su disco**, nella **cartella fornita dal S.O.**(posta in una determinata porzione del filesystem), **riservata e privata per l'app**.
Dato che utilizziamo solo l'internal storage non abbiamo bisogno dei alcun permesso nel manifest.

Se l'utente decide di salvare il grafico l'attività **codificherà il ChartLine in un Bitmap** e poi lo scriverà **aggiungendolo a quelli esistenti** in formato **PNG** sulla cartella a lei riservata e ottenuta tramite

```
/*apre 1 stream in scrittura verso 1 file nello storage interno, privato per l'app. S  
e il file non esiste lo crea*/  
outputStream = openFileOutput(nome_file_png, Context.MODE_PRIVATE);
```

Se invece l'utente decide di salvare tutti i dati contenuti nell'indicatore (maggiori informazioni rispetto a quelle mostrate nel grafico) in un database Sqlite, verranno utilizzate **a tale scopo delle classi per la gestione del database** (descritte di seguito nelle scelte progettuali ed implementative).
Inoltre, considerando la tipologia dei dati da salvare e le query attese dall'utente, abbiamo ritenuto di creare un **database MONOTABELLA**, dove ogni record del database rappresenta un univoco indicatore di un paese, più eventuali informazioni aggiunte dalla nostra app quali per esempio la data e **ora del salvataggio dei dati effettuato**.

E' intuitivo che questo percorso possa incontrare delle eccezioni che cambiano il suo normale flusso d'esecuzione.

Caso d'uso senza errori e/o eccezioni, quali per esempio l'assenza di collegamento ad internet, errori di connessione al server, indicatori selezionati nulli, e altri,
Tali flussi diversi d'esecuzione, che dovranno essere gestiti dall'app, sono illustrati di seguito.

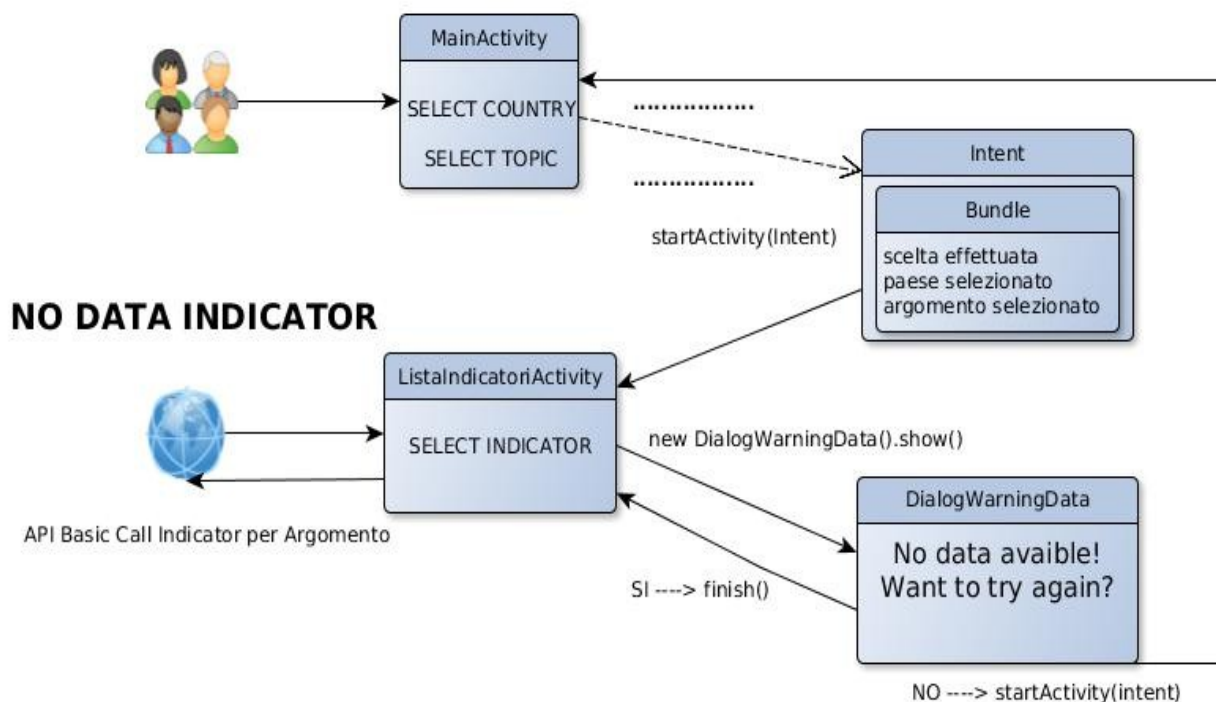
3.1.2 Casi d'uso eccezionali

Tra i casi d'uso eccezionali troviamo:

- **NESSUN INDICATORE PER IL PAESE SELEZIONATO**

è evidente che se **non sono disponibili i dati per l'indicatore** non è possibile costruire il grafico finale. La nostra App quindi, deve gestire questo evento informando l'utente attraverso una notifica (es. Dialog, nuova schermata, etc...). Dopodiché dovrebbe dare all'utente delle possibili alternative per proseguire; o ritenta una nuova selezione cambiando l'ultima variabile (indicatore o paese) oppure ricomincia dall'inizio con una nuova ricerca.

Fig. 2. FLUSSO ESECUZIONE ECCEZIONALE: assenza di dati nell'indicatore selezionato; meccanismo mostra dialog e invio dati tra fragment e attività chiamante.



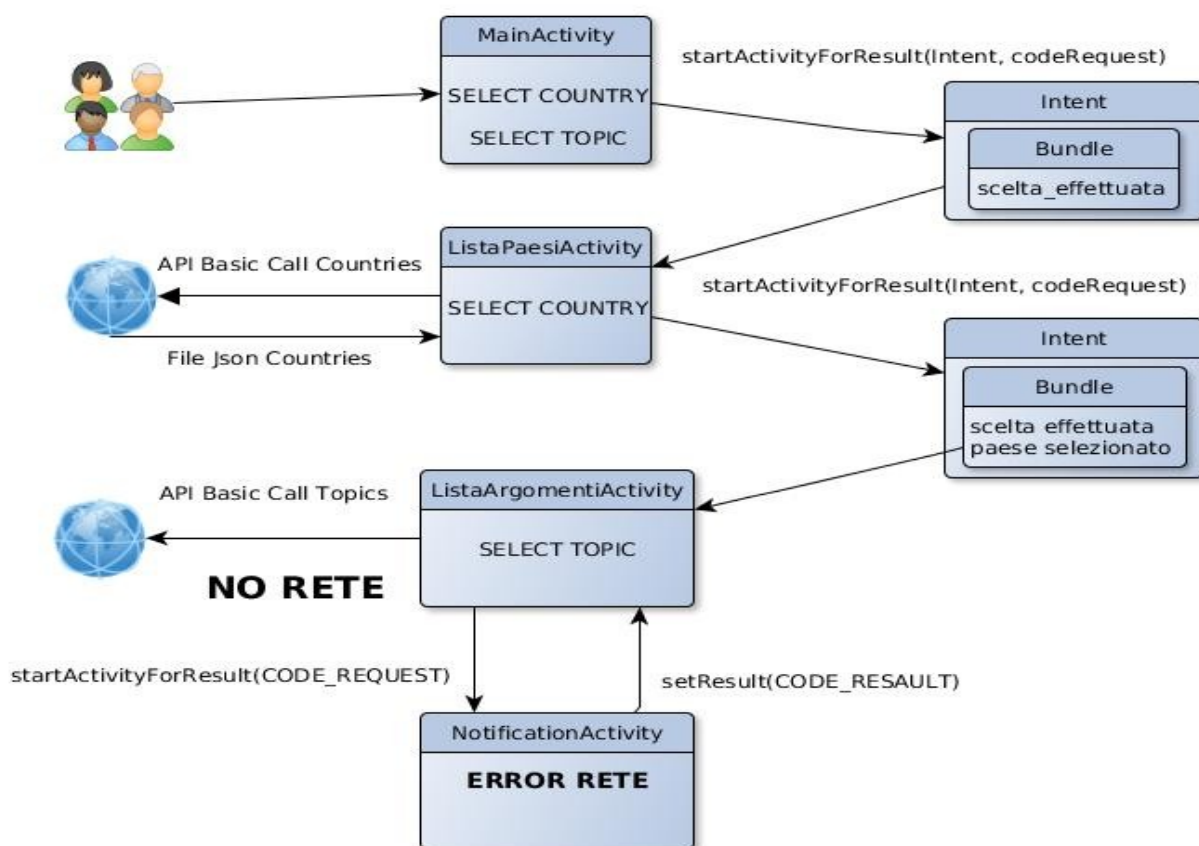
- **ERRORI CONNESSIONE INTERNET**

Un requisito fondamentale affinché l'utente possa arrivare ad ottenere il grafico è la **connessione a internet** : la nostra app ha la necessità di interrogare il sito della WorldBank per completare la ricerca e raggiungere lo scopo del requisito di ricerca.

A questo evento che possiamo classificare come “non previsto” ne possiamo accomunare un altro : il sito della WorldBank potrebbe aver **cambiato indirizzo** internet e quindi, potrebbe risultare irraggiungibile.

La nostra App quindi, deve poter gestire questi eventi informando l’utente attraverso una notifica (es. Dialog, nuova schermata, etc...). Anche qui bisogna valutare delle possibili alternative per l’utente: o torna al main e così può svolgere altre attività che non richiedono la connessione (carica dati offline) oppure visto che era intenzionato ad effettuare la ricerca online, gli si potrebbe dare la possibilità di riprovare nel caso la connessione torni subito, mostrandogli l’ultima schermata dove si trovava prima della mancanza di connessione.

Fig. 3. FLUSSO ESECUZIONE ECCEZIONALE: assenza di collegamento a internet; show notifica; meccanismo return con codice



3.2 CARICAMENTO DATI

SPECIFICHE REQUISITO: L’utente ha intenzione di ricercare offline, nella memoria interna al dispositivo, un indicatore di un paese precedentemente salvato e visualizzarlo, piuttosto che ricercarlo online.

SCOPO REQUISITO:

- aiutare l'utente nel selezionare un indicatore di un paese presente nel database interno al fine di visualizzarlo in forma tabellare, e/o eventualmente eliminarlo;
- mostrare l'ultimo file.png contenente l'immagine del grafico salvato nel filesystem interno

3.2.1 Caso d'uso richiesto

Carica e visualizza i dati degli indicatori salvati nello **storage interno**: dal Sqlite database per mostrare tutti i dati in forma tabellare, e dall'ultimo file.png per visualizzarne il relativo LineChart; Avremo quindi sia il database che i file salvati memorizzati nelle rispettive cartelle del filesystem fornite dal S.O. all'app, e private alla stessa.

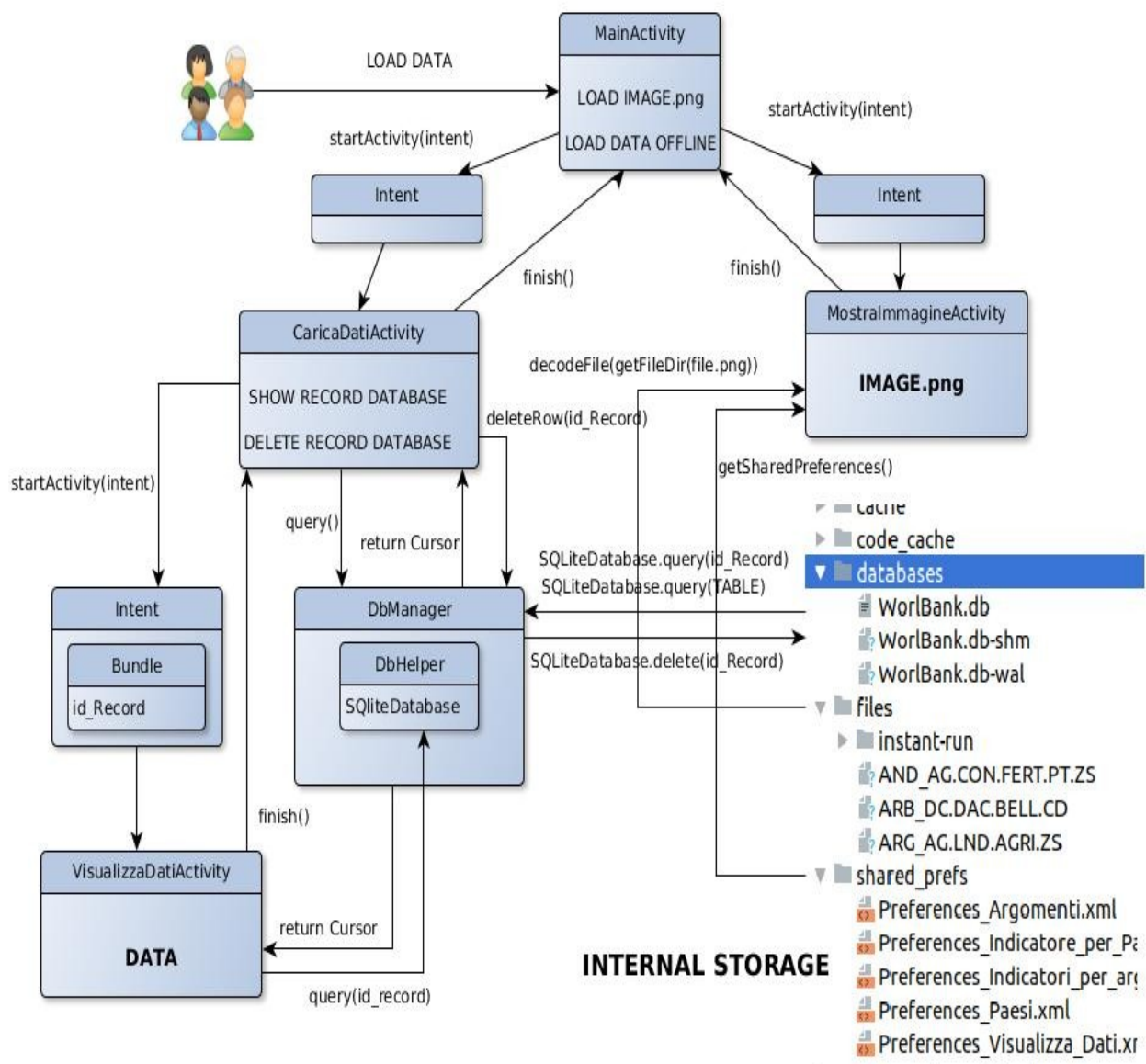


Fig. 4: FLUSSO ESECUZIONE NORMALE.

a) Utente sceglie di **visualizzare** l'ultima **immagine salvata**

Il Main lancerà un'attività MostraImmagine, la quale ha il compito di :

- **leggere** dalle **sharedPreferences** il nome dell'ultimo file salvato.
- **ottenere** dal sistema **la cartella riservata all'app** nello storage interno, e **aprire**, utilizzando le API Java di I/O, uno **stream in lettura** verso il file.png (contenente l'immagine salvata) con il nome ricavato prima.
- **Visualizzare** a schermo l'immagine attraverso un ImageView.
Per fare questo, dovrà prima **decodificare** il file.png in un oggetto Bitmap, il quale poi sarà passato all'ImageView per essere mostrarlo all'utente (maggiori dettagli sono descritti nel rispettivo paragrafo delle scelte progettuali).

b) Utente sceglie di **visualizzare i dati offline**

Il Main lancerà un'attività CaricaDati, la quale ha il compito di :

- **aprire il database** SQLite con l'aiuto delle classi DBManager e DBHelper.
Fondamentalmente la classe DBManager contiene un riferimento ad un DBHelper, attraverso il quale ottiene il database in scrittura o in lettura a seconda dei casi, e ha il compito di wrappare e specializzare le operazioni di default sull'oggetto SQLiteDatabase fornito dalla libreria android (maggiori dettagli nel rispettivo capitolo delle scelte progettuali).
- ottenere attraverso una **query** un Cursore su tutti i record memorizzati (l'intera unica tabella).
- estrarre dagli stessi record solo le informazioni ritenute necessarie per consentire all'utente di selezionare un singolo record del database (un indicatore di un paese salvato), per visualizzarne i dati sottostanti o eliminarlo.
Affiancherà quindi all'oggetto Cursor ottenuto un **MyCursorAdapter**, per fornire i dati alla propria ListView.
- Per **eliminarlo** utilizza sempre le classi per la gestione del database;
- Per visualizzare i dati sottostanti avvierà invece una nuova attività VisualizzaDati incapsulando nell'intent di lancio un Bundle contenente a sua volta **l'id del record** selezionato dall'utente.
L'attività appena avviata, in modo analogo alla classe genitore, effettuerà una **query** al database richiedendo questa volta però **solo il record desiderato**.
Manipolerà i dati dell'indicatore ottenuti in un Cursor e li mostrerà all'utente in **forma tabellare**; questa volta senza l'ausilio di un CursorAdapter, ma utilizzando l'inflazione per caricare i layout delle varie righe della tabella direttamente nell'activity.

In particolare segnaliamo che l'attività VisualizzaDati, per mostrare i dati, utilizza un **TableLayout** con Scroll, perché ci sembra più appropriato per questo tipo di dati.

Anche qui, come per il flusso di ricerca, posso verificarsi casi d'uso eccezionali quando nessun dato o nessun grafico è presente nello storage interno come mostrato di seguito.

3.2.1 Casi d'uso eccezionali

NESSUN DATABASE e/o NESSUN GRAFICO SALVATO

Può accadere che l'utente al termine della fase di ricerca non abbia mai salvato i dati (immagine e/o dati); quindi nella fase di caricamento non potrà vedere nulla. E' importante dunque avvisare l'utente di questo stato attraverso una notifica (es.Dialog) contenente un messaggio che indichi all'utente che alla fine della fase di ricerca ha la possibilità di salvare i dati (come immagine o come record di un database).

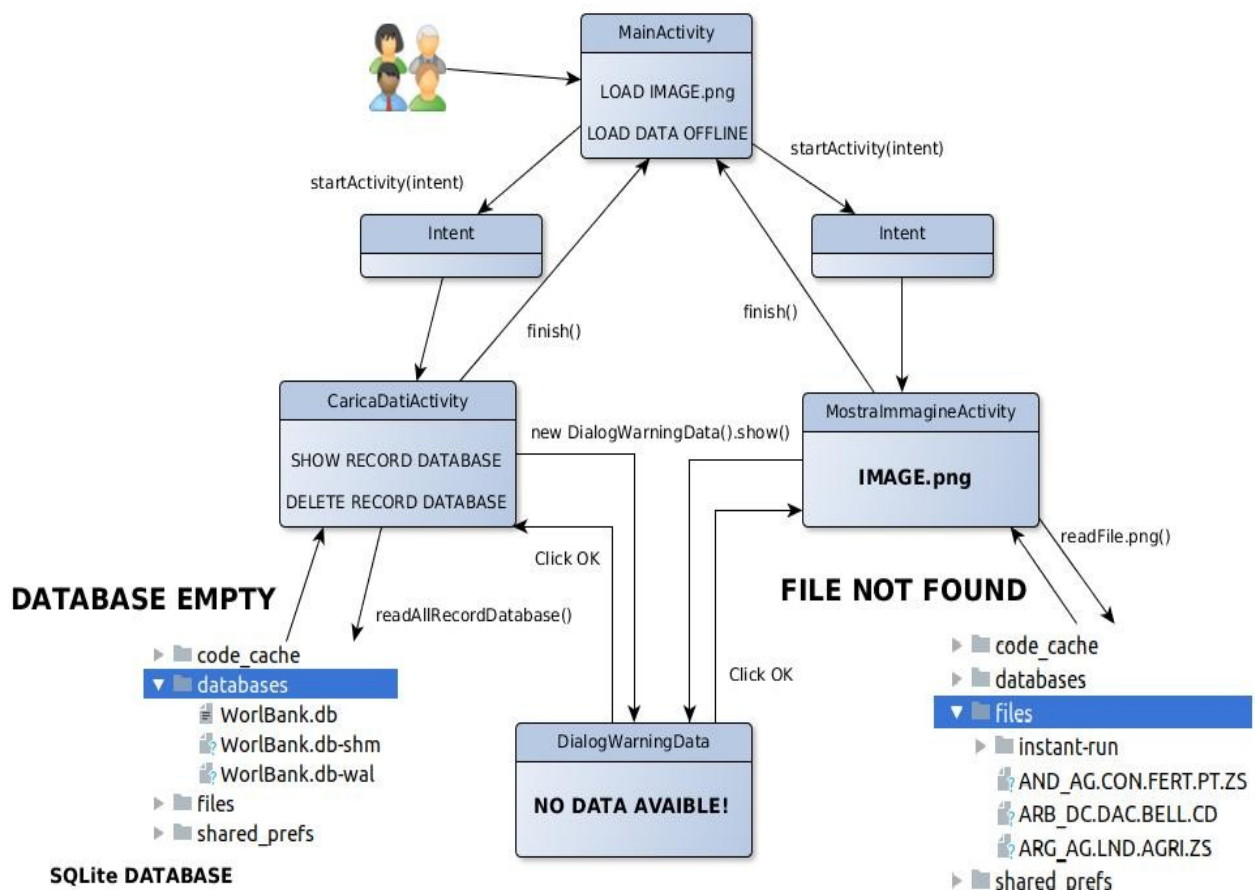


Fig. 5. FLUSSI ESECUZIONE ECCEZIONALI: nessun dato e/o nessun grafico salvato

3.3 CONSIDERAZIONI SUI REQUISITI

Nelle specifiche progettuali per “Salvare localmente” abbiamo inteso internal Storage piuttosto che External Storage.

Quindi tutti i dati salvati dall'app risiedono nella memoria interna del dispositivo e mantengono l'impostazione di default di essere privati per l'app (neppure l'utente può accedervi, a meno che non abbia accesso root.)

Inoltre la memoria interna è unica per tutte le app e i dati salvati al suo interno condividono tale spazio d'archiviazione con le altre risorse dell'app come codice, cartella *res*, *assets*, etc.

Per quanto detto quindi l'internal storage è 1 buon posto x i dati delle app privati a cui l'utente non ha bisogno di accedere direttamente, solitamente di piccole dimensioni, non volatili.

Mentre nel nostro caso, con l'**aumentare dei dati memorizzati nell'internal storage**, il database, e soprattutto le immagini potrebbero occupare **troppo spazio** da **impedire all'utente di installare altre applicazioni**.

Infine quando l'utente disinstalla l'app, i file dell'app salvati nella memoria interna vengono rimossi dal sistema.

Sulla base di queste osservazioni forse avremmo dovuto considerare l'ipotesi, almeno per le immagini, di salvarle nello storage esterno, di norma più grande.

Di default è pubblico e non sempre disponibile (si pensi per esempio alla scheda SD rimovibile).

Oppure come ultima (e forse migliore) ipotesi si poteva considerare la possibilità offerta da Android di salvare sempre nello storage esterno ma in modalità privata, cioè in 1 directory specifica dell'app che il sistema elimina quando l'utente disinstalla l'app.

4 SCELTE PROGETTUALI ED IMPLEMENTATIVE

4.1 MULTI-THREAD:

Ogni **operazione costosa** che un'activity deve fare, quali quelle di I/O:

- connessione al server WorlBank e download file Json
- accesso (lettura e scrittura) allo storage interno (SQLite Database e FileSystem)

vengono effettuate da un thread concorrente per non sovraccaricare il thread dell'interfaccia utente. Sappiamo infatti che se il thread della UI rimane bloccato x + di pochi secondi, il S.O. chiude l'app presentando la famigerata finestra di dialogo " non risponde " (**ANR**). Inoltre la libreria della UI Android *non* è thread-safe, quindi non si può manipolare la UI da un thread diverso da quello principale.

In particolare sono state utilizzate delle **AsyncTask**, che durante lo svolgimento del proprio lavoro collaborano con il thread UI per mostrare una ProgressBar attraverso l'implementazione del metodo onProgressUpdate() e l'invocazione di publishProgress(); al termine dello stesso, forniscono il risultato richiesto sempre al thread della UI nella onPostExecute().

I primi forniranno come risultato il file Json ottenuto dal server; mentre i secondi, i dati letti dallo storage interno (database SQLite utilizzato per i dati e/o la cartella di sistema fornita dal S.O. riservata all'app, per le immagini).

Unica eccezione per le operazione di I/O è l'**accesso** alle **SharedPreferences** su disco, che vengono effettuate dal thread della UI.

Questo perché il thread principale supporta bene le operazioni con le SharedPreferences, in quanto quest'ultime sono di norma di ridotte dimensione (poche centinaia di KiloByte) e ottimizzate dal sistema operativo.

Infine anche la **costruzione del grafico** utilizzando la libreria MPChartAndroid viene effettuata da un **AsyncTask** (interno alla Graficoactivity), in quanto durante i test l'emulatore ci indicava che in quel contesto il thread della UI lavorava al limite del sovraccarico.

4.2 ACQUISIZIONE DATI REMOTI

4.2.1 Connessione al server World Bank Data

In relazione a ciò che la nostra app deve fare, tra le diverse classi messe a disposizione dalla libreria Android per le operazioni di I/O in rete (ad esempio Volley), scegliamo di utilizzare le classi **URLConnection** e **URL**.

Fondamentalmente istanziamo un oggetto URL che rappresenta l'indirizzo della risorsa da acquisire, ovvero una particolare API Basic Call; e insieme alla collaborazione con **URLConnection** apriamo una connessione e un input stream in lettura al file desiderato.

Infine si utilizza un classico **BufferedReader** e ciclo di lettura per leggere l'intero file di testo.

L'attività è ora in possesso, in una variabile stringa privata del **proprio file** di testo in formato **Json**.

4.2.2 Utilizzo GSON

Durante l'analisi dei requisiti abbiamo deciso di utilizzare come formato dei documenti da scaricare il **formato JSON** perché compatto e facilmente comprensibile anche all'uomo.

E' noto che i **file** (regolari) forniti dal database online sono nel seguente formato (come anche illustrato nel prossimo paragrafo):

- contengono un elemento radice: una array Json (array ordinato di oggetti Json) il quale contiene 2 oggetti Json
- il 1° elemento (indice 0) dell'array radice è un **oggetto json intestazione** contenente stringhe e valori numerici che rappresentano i metadati della risorsa.
- Il 2° elemento (indice 1) dell'array radice è a sua volta un array Json contenente n oggetti Json. Quasi tutti sono elementi Json, ognuno dei quali rappresenta le informazioni di una singola entità : paese, argomento, indicatore (vedi paragrafo modello MVC).
Caso leggermente diverso per il file contenente l'indicatore di un paese: in questo caso ogni elemento dell'array Json interno rappresenta la coppia "valore indicatore – anno"

PREMESSA:

Gson trasforma in JSON il POJO e viceversa, basta istanziare l'oggetto Gson e chiamare i metodi fromJson e toJson: consente le trasformazioni in/da JSON anche su classi di cui non si dispone del sorgente, e per far questo utilizza la reflection.

La relazione tra campi JSON e variabili POO è 1-a-1, cioè vengono serializzate tutte le variabili dell'istanza; mentre se un "campo" JSON non ha una corrispondente member variable (con esattamente lo stesso nome), il dato non viene deserializzato.

Quindi una volta che l'**attività corrente** avrà **acquisito il file Json**, prosegue come descritto:

- attraverso GSON **mapperà l'array Json** interno del file scaricato in un oggetto **ArrayList<ElementoGenerico>**.
Dove ElementoGenerico rappresenta l'entità generica del nostro modello di dati.
Il nostro modello di dati prevede infatti una classe per ogni entità del database World Bank Data: classe Paese per rappresentare i paesi; classe Indicatori per rappresentare gli indicatori e così via.
Tali classi estendono la classe ElementoGenerico: come descritto nell'analisi dei requisiti, in realtà eccetto dove è necessario acquisire tutte le informazioni contenute nel file, il JSON da deserializzare risulta essere molto verboso mentre a noi interessa solo una parte.
Di conseguenza **per migliorare l'efficienza** scegliamo di estrapolare solo le informazioni sulle entità che all'attività interessano, e per farlo dichiariamo nelle nostre classi ElementoGenerico solo le informazioni che vogliamo tra tutte quelle presenti nella rispettiva entità Json.
- **completata la deserializzazione**, l'attività ListaPaesi avrà quindi la propria lista di elementi Paese, l'attività ListaArgomenti avrà la propria lista di elementi Argomento, e così' per le altre attività della fase di ricerca.
Esistono **diverse strategie** (almeno 2) per **ottenere tale ArrayList<T>** di oggetti: quella scelta da noi sfrutta la conoscenza della struttura del file che il server ci invia e l'utilizzo delle classi JsonParser e TokenType appartenenti sempre alla libreria GSON

JSON PARSER:

classe che opera a "livello più basso": deserializza il nostro JSON e lo trasforma in una struttura ricorsiva ad albero dove ogni nodo è rappresentato da un **JsonElement**.

L'albero è visitabile trasformando il nodo corrente, se ha figli, in un object json o in un array json: infatti a sua volta, JsonElement può estrarre uno dei tipi base di JSON: un JsonObject, un array JsonArray o un tipo primitivo (string, boolean, numeri interi e a virgola mobile).

In definitiva quindi con JsonParser **attraversiamo l'albero e preleviamo** l'elemento Json che interessa: **l'array Json** contenete la lista di entità relativa all'API Basic Call utilizzata; l'oggetto intestazione Json, eccetto rari casi dove è necessario, non viene deserializzato per efficienza.

TYPETOKEN<T>

Per deserializzare i Generics, come un `ArrayList<T>` non si può usare direttamente il metodo seguente, perché non funzionerebbe

```
public <T> T fromJson(String json, Class<T> classOfT)
```

Es. `gson.fromJson(jsonFile, ArrayList<T>.class);`

Questo dipende non da GSON, ma da come sono implementati i Generics in Java (a runtime, Java non ha informazione sui generics per cui il metodo `getClass` torna sempre `List` come risultato, a prescindere dal tipo di lista che è stata dichiarata).

Per evitare di utilizzare una classe Conteneir aggiuntiva (altra strategia disonibile), decidiamo di utilizzare una piccola classe di utilità `TypeToken` che permette di evitare il problema. Quindi il metodo utilizzato dell'oggetto GSON sarà:

```
public <T> T fromJson(JsonElement json, Class<T> classOfT)
```

Es.: `gson.fromJson(json_element, typeToken.getType());`

Di seguito estrapolati di codice che mostrano in sequenza le operazioni svolte dall'activity: (perché il file ricevuto contiene "troppo" risetto a quanto voluto)

```
public class ListaPaesiActivity extends ListaGenericaActivity {
    .....

    @Override
    public void onResume(){

        /*istanzia le classi che occorrono per realizzare correttamente la deserializzazione*/
        ArrayList<Paese> lista_paes = new ArrayList<Paese>();
        TypeToken<ArrayList<Paese>> listTypeToken = new TypeToken<ArrayList<Paese>>() {};
        .....
    }
    .....
}

public class ListaGenericaActivity extends AppCompatActivity implements
    AdapterView.OnItemClickListener {
    .....

    /*se non sono presenti errori il file json viene trasformato con GSON in una List<T>,
    e collega quest'ultima alla listView tramite l'adattatore che istanzia*/
    protected void caricaLayout(){

        /*con la libreria GSON ottengo la corrispondente lista/array di oggetti del file json*/
        MyGSON myGSON = new MyGSON(this);
        lista_oggetti = myGSON.getListFromJson(json_file, typeToken);
        .....
    }
}
```

```

    }
    .....
}

```

public class MyGSON {

.....

*/*riceve il file json e istanzia e ritorna la rispettiva lista Java*/*

public ArrayList getListFromJson(String file_json, TokenType typeToken){

*/*ottengo il nodo radice dell'albero*/*

JsonElement je = **new** **JsonParser()**.parse(file_json);

*/*sò che è un array e che contiene 2 oggetti Json*/*

JsonArray root = je.getAsJsonArray();

.....

*/*ottengo il secondo elemento sapendo che è un array Json di oggetti/entità Json*/*

JsonElement je2 = root.get(1);

JsonArray array_json = je2.getAsJsonArray();

*/*istanzio un Gson e sfrutto il metodo della TokenType per deserializzare l'array Json ricevuto*/*

Gson gson = **new** **Gson()**;

ArrayList list = gson.fromJson(je2, **TokenType.getType()**);

return list;

}

}

4.3 GESTIONE DELLE NOTIFICHE

4.3.1 Mancanza di connessione a Internet

Prima di spiegare la scelta effettuata per mostrare all'utente la notifica di questo evento vogliamo mostrare in che modo è stato eseguito il controllo della mancanza di connessione.

Sono state considerate tre opzioni: impostare un ascoltatore (BroadcastReceiver); andare a controllare la connessione direttamente nel momento in cui ne avevamo bisogno (es. Uso della classe ConnectivityManager di android); oppure sull'oggetto HttpURLConnection, attraverso il quale eseguiamo la connessione, impostiamo un timeout in ricezione (**client.setReadTimeout** (res.getInteger(R.integer.**TIMEOUT**))).

Abbiamo optato per la terza scelta perché più immediata.

In questo modo se i dati non diventano disponibili entro il tempo prefissato viene lanciata l'eccezione (**catch** (IOException e) {...}).

Proprio per come è fatto il S.O., dopo la prima volta che si verifica questo evento se riproviamo a connetterci l'eccezione verrà lanciata non appena invochiamo la openConnection()).

Tornando al discorso della scelta tra la dialog o la schermata abbiamo scelto la seconda opzione. La schermata è completamente personalizzata e presenta un messaggio del tipo: “*Unable to contact*”.

Una volta ricevuta la notifica bisogna decidere come proseguire: o l'utente torna alla pagina principale per fare altre operazioni che non richiedono la connessione a Internet (es. Carica immagine precedente) oppure torna nell'ultima schermata in cui si trovava prima della mancanza della connessione.

Alla fine dell'analisi si è deciso quest'ultima strada perché così facendo si dà la possibilità all'utente di ritentare una nuova selezione; se nel frattempo la connessione dovesse essere tornata l'utente avrà la possibilità di proseguire nella ricerca dato che si ritrova tutti i dati salvati nel bundle.

Per implementare questo meccanismo abbiamo dovuto gestire l'avvio dell'attività che mostra la notifica (Model/NotificationActivity) con startActivityForResult().

Infatti quando l'utente preme il relativo button per tornare indietro, la NotificationActivity prima di terminare setta un codice in modo tale che l'attività chiamante, una volta ricevuto il codice di risultato, può terminare (*finish()*) e permettere all'attività che si trova nello stack delle attività di tornare in primo piano, cioè proprio quello che volevamo.

Questo tipo di implementazione ci ha portato necessariamente a gestire un'altra condizione: l'utente invece di cliccare sul bottone di NotificationActivity potrebbe cliccare sul pulsante Back nativo del dispositivo per tornare indietro.

Quando questo avviene, il S.O. imposta un codice di risultato pari a *RESULT_CANCELED*; se l'attività chiamante non gestisce questo codice non potrà terminare e non potrà dare la possibilità all'altra attività che sta nello stack di tornare in primo piano.

Andando a gestire questo codice, invece, l'utente potrà indifferentemente cliccare sul button o sulla freccetta con il risultato ritrovandosi esattamente nell'ultima schermata visualizzata prima che venisse a mancare la connessione, e quindi riprendere da lì la ricerca.

Direttamente dal codice, un estratto per mostrare quanto spiegato fin'ora:

@Override

protected void onActivityResult(**int** requestCode, **int** resultCode, Intent data) {

.....

```
if(resultCode == res.getInteger(R.integer.RETURN_FROM_NOTIFICATION_ACTIVITY) ||  
    (resultCode == RESULT_CANCELED && requestCode ==  
    res.getInteger(R.integer.RETURN_FROM_NOTIFICATION_ACTIVITY))){  
    finish();}}
```

dove *RETURN_FROM_NOTIFICATION_ACTIVITY* è il codice settato dall'attività che mostra la notifica mentre *RESULT_CANCELED* è il codice settato dal S.O.

L'AND logico che troviamo tra resultCode e requestCode all'interno del blocco IF ci garantisce che l'attività chiamante terminerà solamente se ha lanciato l'attività di notifica per permettere all'altra che sta nello Stack di di tornare di nuovo in primo piano.

4.3.2 File JSON non valido

Con riferimento al casi d'uso eccezionale "NESSUN INDICATORE PER IL PAESE SELEZIONATO" vediamo che può accadere che non ci siano dati per l'indicatore selezionato. Questo evento chiaramente non può permettere all'utente di ottenere il grafico finale e quindi va avisato con una notifica.

Prima di spiegare come abbiamo gestito questo evento vogliamo far notare come è stato possibile rilevarlo: la chiave di volta sono stati gli innumerevoli test effettuati con diversi paesi e indicatori. Di fatto quando i dati sono disponibili il risultato della api `indicatore_per_paese` fornisce un file così strutturato:

```
[
  • {
    • "page": 1,
      • "pages": 1,
      • "per_page": 10000,
      • "total": 59,
      • "sourceid": "2",
      • "lastupdated": "2019-04-24"
    },
  • [
    • {
      • "indicator": {
        • "id": "AG.AGR.TRAC.NO",
        • "value": "Agricultural machinery, tractors"
      },
      • "country": {
        • "id": "AW",
        • "value": "Aruba"
      },
      • "countryiso3code": "ABW",
      • "date": "2018",
      • "value": null,
      • "unit": "",
      • "obs_status": "",
      • "decimal": 0
    },
    • {
      .....
      .....
    }
  ]
]
```

Notiamo che nella risposta (array Json) c'è sempre una intestazione e poi l'array con i dati che ci servono per costruire il grafico. Quando invece i dati non sono disponibili abbiamo verificato che la struttura del messaggio è la seguente:

```
[
  • {
    • "page": 0,
```

```

        • "pages": 0,
        • "per_page": 0,
        • "total": 0,
        • "sourceid": null,
        • "lastupdated": null
    },
    • null
]

```

C'è sempre un'intestazione ma stavolta al posto dell'array troviamo il **campo null**.

A noi chiaramente quello che ci interessa è sempre il secondo elemento del JSONArray e quindi se andiamo ad assegnare questo elemento ad un nuovo elemento json (JsonElement je2 = root.get(1)) verrà lanciata una eccezione.

Come prevenire tutto ciò? Anche qui tra le varie opzioni possibili abbiamo implementato un semplice controllo sfruttando il metodo isJSONArray() di GSON che ritorna true se l'elemento rappresenta effettivamente un array, false altrimenti.

Quindi eseguendo un semplice confronto si può prevenire questa condizione

```
(if(je2.isJSONArray()==false) return null).
```

Un **altro tipo di risposta** che abbiamo visto durante i test è la seguente:

```

[
  {
    "message": [
      {
        "id": "175",
        "key": "Invalid format",
        "value": "The indicator was not found. It may have been deleted or archived."
      }
    ]
  }
]

```

Il messaggio dice chiaramente che non è stato trovato nessun indicatore dando anche delle possibili motivazioni; inoltre il secondo elemento dell'array non esiste. Come gestire?

Osservando la struttura del messaggio ci accorgiamo che si tratta di un array di dimensione pari a 1; mentre se i dati sono disponibili la dimensione è sempre >=1.

Questo ci suggerisce come soluzione di eseguire un controllo sulla dimensione:

```
( if(root.size()==1) return null).
```

Per entrambe le condizioni che si possono verificare, quando l'attività che deve costruire il grafico si vede ritornare null (nel nostro caso trattasi di GraficoActivity) non potrà più continuare nel suo lavoro ma dovrà fare in modo di avvisare l'utente con una notifica.

In questo caso si è deciso di utilizzare le dialog (vedi Gestione delle dialog).

4.3.3 Gestione delle dialog

Per mostrare le notifiche all'utente durante i casi d'uso eccezionali si è deciso di fare uso di finestre di dialogo che appaiono in sovrapposizione alla schermata corrente (tranne la mancanza di connessione a internet per la quale si è deciso di fare uso di activity).

Da quando Android è stato rilasciato, la creazione di una finestra di dialogo con un layout personalizzato è sempre stata complicata, dal momento che la classe `Android Dialog` non era pensata per farlo.

Fortunatamente con il livello API 11, il team di Android ha rilasciato `DialogFragment` che successivamente è stato aggiunto alla libreria di supporto rendendolo disponibile per tutti i target.

Sostanzialmente un `DialogFragment` è un frammento che mostra una finestra di dialogo fluttuante su quella della sua attività e si comporta da contenitore per la finestra.

Poiché si tratta di un frammento, si integra nel ciclo di vita dell'attività e garantisce che ciò che accade nella finestra di dialogo rimanga sempre coerente con ciò che accade nella activity (ad esempio quando l'utente preme il pulsante Indietro o ruota lo schermo) mentre prima le dialog erano generalmente entità autonome.

Il minimo che deve essere implementato quando si crea un `DialogFragment` è il metodo `onCreateView` o il metodo `onCreateDialog`.

Usando `onCreateView` l'intera vista della finestra di dialogo verrà definita tramite XML personalizzato; usando invece `onCreateDialog` si possono costruire e configurare delle classi di dialogo standard come ad esempio le **AlertDialog** e quest'ultime sono proprio quelle utilizzate dalla nostra App.

Esse ci permettono di definire delle finestre di dialogo semplicemente personalizzando gli oggetti di dialogo già disponibili come ad esempio: title, button, message, icon, view etc.... Quindi per ogni tipo di notifica si è potuto personalizzare il contenuto della dialog a piacere anche se alla fine la struttura utilizzata è stata sempre la stessa: un titolo, un'icona, un messaggio, dei button per implementare le decisioni da parte dell'utente (es. si, no, ok...). Inoltre per le dialog che si aprono dal menu della pagina principale (vedi Menu App Bar) è stato aggiunto anche un **layout XML personalizzato**, altra funzione offerta da `AlertDialog`.

Un altro aspetto da considerare è che la scelta che viene compiuta dall'utente attraverso la dialog in alcuni casi deve essere comunicata all'attività che l'ha lanciata.

Ad esempio, osservando il grafico dei casi d'uso eccezionali in 3.2.1 si nota che quando non è disponibile nessun database o nessuna immagine l'utente non può fare altro che prenderne atto e poi cliccare su un button per chiudere la finestra (es. Click su ok).

Ma questa azione porta con sé il fatto che l'attività che l'ha lanciata non potendo mostrare nulla sarà costretta a terminare, riportando così in primo piano l'altra che si trovava nello stack.

Per permettere dunque il passaggio delle informazioni tra il fragment e l'activity si è implementato lo stesso meccanismo che usa un frammento per creare un listener personalizzato.

Vediamo i passi:

1. Si definisce un'interfaccia nella dialog con metodi che possono essere invocati per passare i dati del risultato all'activity:


```

public interface OnClickListener {
    void onFinishClickListener(String inputText);
}

```

2. Si imposta un listener sull'evento di click che trasmette l'informazione con il metodo dell'interfaccia:

```

@Override
public void onClick(DialogInterface dialog, int which) {
    .....
    DialogWarningData.OnClickListener listener =
        (DialogWarningData.OnClickListener) getActivity();
    ....
    listener.onFinishClickListener(res.getString(R.string.FINISH));
}

```

3. Si implementa l'interfaccia nell'activity per ricevere gli eventi della dialog:

```

@Override
public void onFinishClickListener(String inputText) {
    ....
    if(inputText.equals(res.getString(R.string.FINISH))){
        finish();
    }
}

```

Tutte le dialog vengono lanciate nel thread UI al termine del background ricevendo come parametro il risultato della computazione e visto che molte di esse sono uguali per tipologia (es. DialogWarningData vale sia quando mancano i dati che l'immagine) invece di utilizzarne due o anche più, si è fatto **uso del bundle** passando tutti gli "ingredienti" che servono a **differenziare** la stessa **dialog** a **seconda del contesto**:

Nella activity:

```

.....
@Override
protected void onPostExecute(.....){
    .....
    if(.....){
        DialogWarningData mydialog = new DialogWarningData();
        Bundle bundle = new Bundle();
        ....
        bundle.putStringArray(.....);
        bundle.putInt(.....);
        bundle.putBoolean(....., false);
        mydialog.setArguments(bundle);
        mydialog.show(getSupportFragmentManager(),
            "mydialog");
    }
    else

```

```
.....  
}
```

Nella dialog:

```
public class DialogWarningData extends AppCompatActivity {  
.....  
    public Dialog onCreateDialog(Bundle savedInstanceState) {  
        .....  
        AlertDialog.Builder builder = new AlertDialog.Builder(getContext());  
        .....  
        Bundle bundle = getArguments();  
        .....  
    }  
  
    .....  
    return builder.create();  
}
```

4.4 GESTIONE CICLO DI VITA DELLE ATTIVITÀ

Fino ad ora abbiamo visto come l'app e le sue activity si comportano nei diversi casi d'uso. Ora descriviamo come abbiamo gestito il **ciclo di vita della singola attività corrente**, comprendendo anche quelle situazioni non descritte sopra come ad esempio il caso in cui l'utente interrompe l'utilizzo della nostra app per dedicarsi ad altre app, piuttosto che il caso in cui l'utente cambia l'orientamento del dispositivo.

4.4.1 Callback del ciclo di vita dell'activity corrente

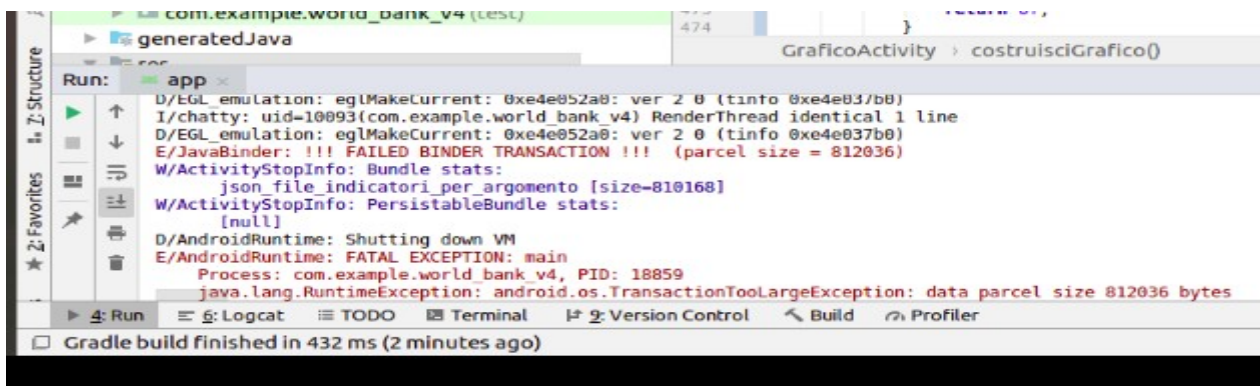
Scegliamo, sempre con l’ottica di migliorare le prestazioni, di far connettere e **scaricare il file Json** dall’attività corrente **solo la prima volta** che viene avviata da un’attività precedente.

In tutti gli **altri casi di ripresa** dell’attività, l’**attività recuperà** il file Json, e in generale tutte le informazioni che gli occorrono per essere ripresa dall’utente o dal sistema (le proprie variabili d’istanza non statiche), **dal Bundle se ricevuto** dal S.O. , **o in alternativa, dalle SharedPreferences.**

Questo perché riteniamo di norma più veloce un accesso eventualmente su disco piuttosto che in rete.

Unica eccezione : l'attività ListaIndicatori non prova a salvare ed eventualmente ricevere il Bundle di cui sopra, ma salva il proprio stato d'istanza solo sulle SharedPreferences.

Questo perché in fase di test ci siamo accorti che, sebbene la lista degli indicatori è un sottoinsieme della lista indicatori del database online (sottolista di indicatori mappata sull'argomento selezionato), alcune sottoliste di indicatori erano troppo grandi per essere contenute nel Bundle, generando a runtim la **TransactionLargeException**



In particolare le **sharedPreferences** salvate da ogni attività vengono **sovrascritte** ogni volta che l'attività è stata avviata (e ri-avviata) dalla precedente ed ha ottenuto nuove informazioni mentre era attiva in primo piano sullo schermo: quindi manterranno dimensioni quasi costanti durante tutta

l'esperienza utente e i dati mostrati all'utente sono **comunque aggiornati** all'ultimo avvio dell'attività.

Ogni attività utilizza un **proprio file** nelle SharedPreferences in cui salva e da cui recupera il proprio stato d'istanza.

Unica eccezione è l'attività MostraImmagine che per mostrare l'ultimo file salvato accede alle SharedPreferences create dall'activity GraficoActivity per prendere il nome del file.

Le SharedPreferences sono state create in **modalità PRIVATA** quindi non accessibili da altre app, ma da altre activity della stessa app si.

Prima di spiegare, più o meno in dettaglio, come sono state utilizzate le **callback** del ciclo di vita delle attività per gestire l'attività corrente (nei possibili scenari che possono avvenire, interni ed esterni all'app) mostriamo un grafico che sintetizza il quadro generale ed aiuta a fissare le idee.

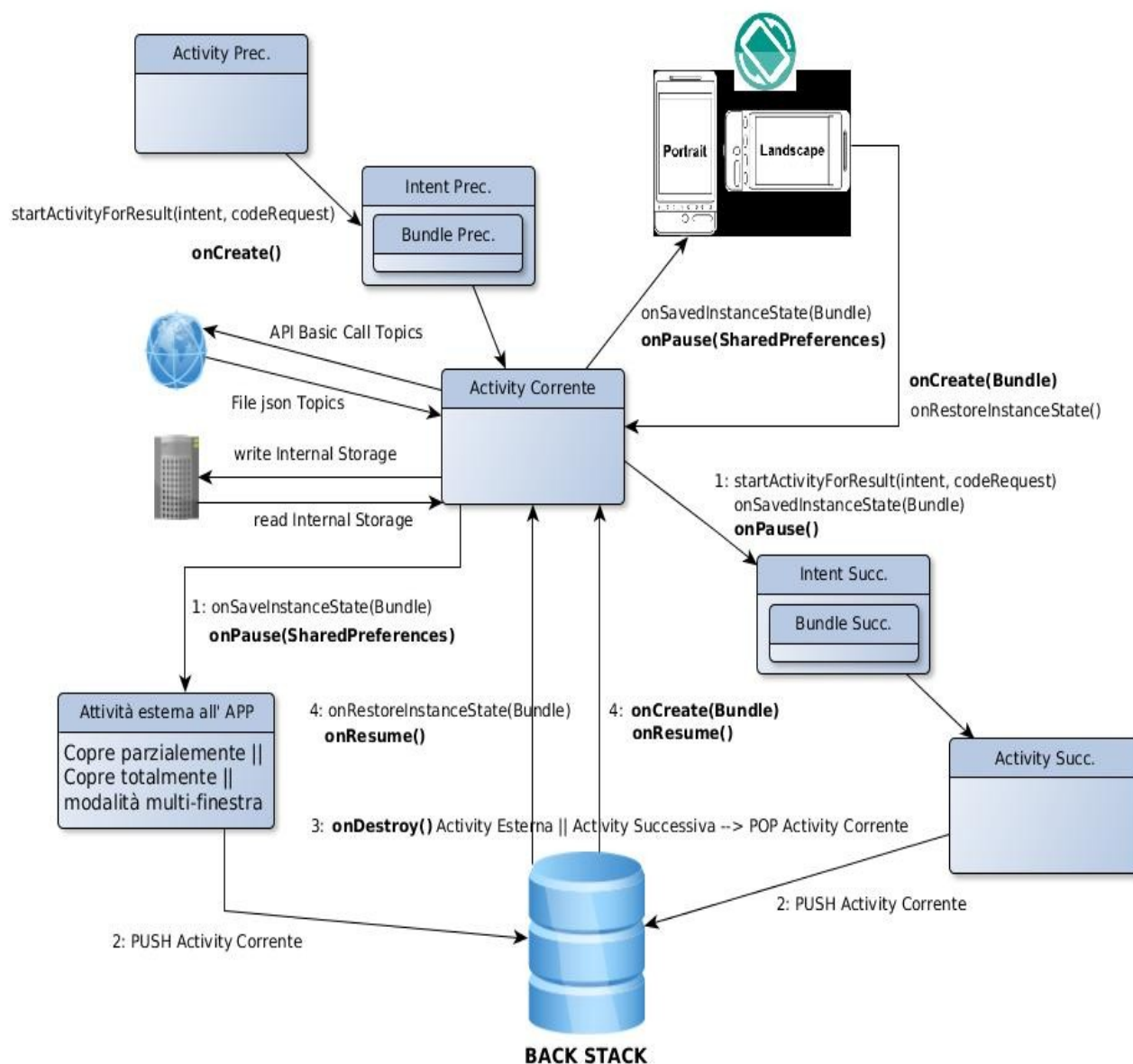


Fig.6: gestione callback ciclo di vita attività corrente

Ricordiamo che in generale l'attività può avere l'attenzione dell'utente, ma può anche perderla e riacquisirla, o cambiare orientamento del dispositivo; quindi l'attività sarà creata la prima volta solo dall'activity precedente, ma può essere ripresa (onResume()) o ricreata (onCreate() + onResume()) in diversi contesti.

Cercando di sintetizzare descriviamo i principi che governano il comportamento di ogni activity:

- **onCreate(Bundle):** l'attività eseguirà tale metodo **ogni volta che è stata lanciata** da un'attività **precedente o ricreata dal S.O.** causa **vincoli d'integrità**.
In quest'ultimo caso poi **può o meno ricevere il Bundle dal S.O.** : se l'attività è riuscita o meno a salvarlo precedentemente, quando era in 1° piano sullo schermo, nella **onSaveInstanceState(Bundle)**.
Sappiamo infatti che la onSaveInstanceState(Bundle) non è garantita essere chiamata dal S.O. Nel caso venga chiamata, viene fatto dopo la onPause().

In questo metodo l'attività svolge le operazioni che dovrebbero svolgersi solo una volta, come caricare le variabili statiche (quali per esempio il layout dell'attività).

Inoltre **verifica e registra**, controllando il Bundle ricevuto se è stata effettivamente lanciata da un'altra attività oppure è stata ricreata dal S.O. causa vincoli di sistema.

In definitiva se è stata lanciata da un'attività precedente sicuramente il Bundle ricevuto nella onCreate() sarà NULL, mentre il Bundle ricevuto nell'intent di lancio sarà sovrascritto.

- Dopo la onCreate() se è il caso che è stata lanciata da un'activity precedente allora l'attività passerà direttamente nello stato Ripresa (onResume()).
Se invece è il caso che l'attività è stata ripresa, allora a seconda se l'attività era riuscita a salvare il Bundle nella onSaveInstanceState() oppure no, il prossimo metodo che il S.O. chiamerà subito prima della onResume() è il metodo **onRestoreInstanceState(Bundle)**.
Quest'ultimo inoltre viene chiamato SOLO se Bundle != NULL : quindi sicuramente se l'attività passa attraverso questo metodo è il caso in cui è stata ripresa e non lanciata da una precedente, e registra tale affermazione.
- A questo punto in tutti i casi (creata, ricreata, ripresa), l'attività entra in **onResume()** dove vengono inizializzate, o reinizializzate se rilasciate (volontariamente o meno) nella onPause(), tutte le variabili e componenti che serviranno quando l'attività tornerà visibile e ad interagire con l'utente.
Sappiamo che infatti che dopo la onPause(), l'attività viene ripresa con tutte le variabili d'istanza reinizializzate a NULL (escluso appunto quelle statiche).
E' qui che assume importanza la verifica effettuata nei metodi precedenti alla onResume():
lo stato dell'istanza precedentemente salvato verrà infatti qui **ripristinato** :
 - dal Bundle se ricevuto, passato all'activity in onRestoreInstanceState()
 - dalle SharedPreferences sempre nel caso della ListaIndicatori, ma in generale se il S.O. non era riuscito a far salvare precedentemente all'activity il Bundle in onSaveInstanceState()
 - le inizializzerà per la 1° volta (per esempio scaricando il file Json dalla rete) se è stata lanciata dall'attività precedente.

- Senza scegliere troppo nei dettagli implementativi, ora che l'attività ha tutte le variabili caricate e quindi reinizializzate, **è qui**, nella `onResume()`, **che instanzierà o reinstancierà**, ed utilizzerà, classi e meccanismi quali `ArrayAdapter`, `CursorAdapter`, `ListView`, `Listeners`, `inflate()`, etc.. per mostrare l'interfaccia utente desiderata e reattiva all'utente. Dopo la `onResume()` l'**attività è in cima al BACK STACK** e interagisce con l'utente.
- A questo punto, per diversi motivi quali : l'utente interrompe l'attività premendo il pulsante Back nativo del dispositivo, tornano indietro attraverso l'`AppBar`, oppure ancora lancia un'altra attività, etc.. il S.O. sicuramente chiamerà e terminerà il metodo **`onPause()`**. Essendo l'unico metodo sicuro dove salvare i dati, è qui che vengono salvate le `SharedPreferences` contenenti lo stato dell'istanza in maniera persistente.
- Sempre nell'ottica dell'efficienza, lo stato dell'istanza dell'attività viene anche salvato nel metodo **`onSaveInstanceState(Bundle)`** attraverso un `Bundle`, cosicché nel qual caso si riesce a portare a termine la sua esecuzione, al momento della ripresa dell'attività, lo stato dell'istanza sarà recuperato dallo `Bundle` prima salvato e ora recapitato dal S.O.
- Non è stato implementato invece il metodo **`onStop()`** perché non è stato ritenuto necessario per supportare meglio la modalità multi-finestra.
- Infine eccetto per le attività che interagiscono con il database, la **`onDestroy()`** si occupa solo di chiamare il suo metodo della superclasse per permettere la distruzione dell'attività e la sua estrazione dal Back Stack.
Le classi che invece hanno riferimenti al Database e al Cursor , si preoccuperanno anche di rilasciare tali riferimenti.
Come vedremo nel paragrafo Gestione del database, poiché **`getWritableDatabase()`** e **`getReadableDatabase()`** sono costosi da chiamare quando il database è chiuso, è necessario lasciare aperta la connessione del database x tutto il tempo necessario x accedervi.
In genere, è ottimale **chiudere il database** nella chiamata **`onDestroy()`** dell'attività.

4.4.2 Meccanismo StartActivity

Osservando il grafico su caso d'uso richiesto di ricerca si evince che le activity coinvolte sono , nell'ordine le seguenti:

- MainActivity
- ListaPaesiActivity
- ListaArgomentiActivity
- ListaIndicatoriActivity
- GraficoActivity


Dunque spiegheremo quale logica è stata adottata nel momento in cui una delle Activity lancia la successiva sapendo che abbiamo due possibili scelte: **startActivity(intent)** o **startActivityForResult(intent,requestCodeID)**.

Nel primo caso l'attività "padre" non ha nessun bisogno di ottenere dei risultati da parte dell'attività "figlio" mentre nel secondo caso ci potrebbe essere questa possibilità e quindi i risultati del figlio (se presenti) possono essere raccolti in maniera asincrona dal padre.

Arriviamo quindi al nocciolo della questione: nella nostra fase di ricerca ci sono dei casi in cui una activity ha bisogno di conoscere i risultati dell'altra che ha lanciato?

In linea di massima la risposta è no perché l'obiettivo della fase di ricerca è quella di arrivare fino in fondo alla visualizzazione del grafico e quindi nessuna attività ha bisogno di sapere cosa ha fatto l'altra. Però ci può essere la seguente **eccezione**: ad un certo momento può venire a mancare la connessione a internet (requisito basilare per la fase di ricerca).

In questo caso la activity che ha rilevato questo evento (ListaPaesiActivity, ListaArgomentiActivity, o ListaIndicatoriActivity) deve necessariamente lanciare l'attività di notifica con la **StartActivityForResult** perché in questo caso, una volta ricevuto il codice di risultato settato da quest'ultima, termina in modo da riportare in primo piano l'attività precedente dallo stack activity (la gestione di questa notifica è spiegato nel paragrafo gestione delle notifiche).

Infine, un altro caso in cui si necessita della StartActivityForResult è quando l'utente, dopo aver ottenuto e visualizzato il grafico finale, clicca sull'**icona** dell'appBar() per tornare direttamente alla pagina iniziale senza passare per le schermate intermedie. In questo caso GraficoActivity imposta un codice di risultato (BACK_HOME) che viene ricevuto dall'activity chiamante in **OnActivityResult**. Quest'ultima capisce che deve terminare ma prima di farlo setta lo stesso codice. Si innesca un meccanismo di reazione a catena tale per cui le activity che hanno implementato startActivityForResult (ListaPaesiActivity, ListaArgomentiActivity, ListaIndicatoriActivity) terminano tutte riportando in primo piano MainActivity:

In GraficoActivity:

```
@Override
public boolean onOptionsItemSelected(MenuItem item){
    .....
    int id = item.getItemId();
    switch(id) {
        .....
        case R.id.ritorno_home:
            setResult(getResources().getInteger(R.integer.BACK_HOME));
        }
    finish();
    return false;
}
```

Nelle sottoclassi di ListaGenericaActivity:

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    .....
    if(resultCode == res.getInteger(R.integer.BACK_HOME)) {
        setResult(res.getInteger(R.integer.BACK_HOME));
        finish();
    }
}

```

Per quanto riguarda la fase di caricamento le activity coinvolte sono invece:

- MainActivity
- VisualizzaDatiActivity/MostraImmagineActivity

Ricordiamo che per queste operazioni l'utente non necessita di connessione e inoltre, se non dovessero essere presente nessun dato/grafico salvato, l'activity che ha rilevato questo evento (VisualizzaDatiActivity o MostraImmagineActivity) lancia la dialog per avvisare l'utente; il quale non può fare nient'altro che prenderne atto e chiudere la dialog.

Chiudendo la dialog, anche l'activity termina e ritorna in primo piano la MainActivity. Quindi per la fase di ricerca è stata solamente implementata la startActivity().

4.4.3 Gestione Database

Come da requisiti il database risiede nello storage interno (come da impostazione predefinita) e quindi è **privato all'app**.

Per gestire il database le seguenti vengono utilizzate 4 classi:

1. **DBHelper** extends SQLiteOpenHelper:

- Questa classe permette di **ottimizzare l'utilizzo del database** sottostante: quando viene utilizzata per ottenere riferimenti al database, il S.O. esegue le operazioni potenzialmente lunghe di creazione e aggiornamento dello stesso solo quando necessario e *non durante l'avvio dell'app*.
Per esempio per differire l'apertura del database fino al 1° utilizzo ed evitare il blocco dell'avvio dell'app con aggiornamenti di database di lunga durata.

- Contiene lo **schema** del database (dichiarazione formale) di come il database è organizzato sotto forma di **costanti** : 1 nome di database, il nome della tabella (unica) e i nomi delle colonne.
- si occupa di **aprire il database se esiste** oppure **creandolo altrimenti**.
In particolare facciamo l'override del metodo **onCreate(SQLiteDatabase db)** il quale viene invocato nel momento in cui non si trova nello spazio dell'app 1 database con nome indicato nel costruttore: verrà invocato 1 sola volta, quando il database non esiste ancora.

2. **SQLiteDatabase** (fornita dalla libreria, e rappresenta il database Sqlite):

- SQLiteDatabase contiene i **metodi che manipolano direttamente il database SQLite** come: creare, eliminare, aggiungere un record; eseguire **comandi SQL** ed altre attività di gestione di database comuni.
Ad esempio nell' **onCreate(SQLiteDatabase db)** di cui sopra viene utilizzato il metodo

```
public void execSQL(String sql)      /*che non è 1 SELECT o qualsiasi altra istruzione
SQL che                             estituisce dati*/
```

3. **DbManager** (tramite tra l'attività e il database)

- utilizzata dall'attività direttamente per gestire il database: contiene un riferimento all'oggetto DbHelper che lei istanzia, e tramite esso ottiene quando necessario il database sottostante in lettura e scrittura.
- Fornisce dei comodi metodi *wrapper*, dei sottostanti metodi SQLiteDatabase, per personalizzare l'accesso e la modifica del database.
Di seguito ne mostriamo due di essi utilizzati dall'attività CaricaDati come esempio:

Cancella un record del database quando l'utente clicca sull'icona cestino del rispettivo elemento:

```
public boolean delete(long id){
    SQLiteDatabase db = dbhelper.getWritableDatabase(); /*ottiene il riferimento al database in
                                                         lettura*/

    db.beginTransaction();
    try
    {
        if (db.delete(DbHelper.TABLE_NAME, DbHelper.COLUMN_ID + "=?",
        new String[]{Long.toString(id)})>0)
        return true;

    ...finally {
        db.setTransactionSuccessful();
        db.endTransaction();

        ....
    }
}
```

Metodo per ottenere un Cursore sull'intera tabella:

```
public Cursor query() {    /*ritorna tutte le righe della tabella DbHelper.TABLE_NAME*/
    Cursor crs=null;
    try
    {
        SQLiteDatabase db = dbHelper.getReadableDatabase();    /*ottiene il riferimento al
                                                                database in lettura*/
        crs=db.query(DbHelper.TABLE_NAME, null, null, null, null, null, null, null);
    }
    .....
    return crs;
}
```

RecordTabella:

è stata da noi introdotta questa classe che implemta un record, non strettamnte necessaria, per facilitare l'interazione con la tabella del database, in particolare nel nostro caso per aggiungere i record nel database.

Si può notare inoltre dai snippet di codice sopra mostrati:

- l'utilizzo delle **TRANSAZIONI** (politica del completo successo o fallimento totale) nel caso di scritture verso il database per aiutare a mantenerei dati coerenti e prevenire perdite di dati dovuti a chiusura anomala dell'app;
- e che nel casi di query() verso il database, l'attività riceverà un appropriato **Cursor** attraverso il quale fornirà l'insieme di risultati ottenuti ad un AdapterView (come per esempio ListView) tramite un CursorAdapter, o direttamente ad un Layout (nel nostro caso il TableLayout) sempre tramite inflazione, per mostrarli poi a piacimento all'utente.

4.4.4 Model-View-Controller

Nel costruire l'app si è cercato di seguire il modello di progettazione di software Model-View-Controller (MVC), o similare, il cui obiettivo principale è quello di separare l'interfaccia utente, il

modello dei dati sottostanti, e la logica di input; per migliorarne modularità, riutilizzo, comprensione e manutenzione.

Fondamentalmente abbiamo:

- **MODELLO**
Insieme di classi che rappresentano le informazioni/dati da rappresentare: sia quelle fornite ed acquisite dal World Bank Data come **Paese, Indicatore, Valore Grafico, etc..**, e sia quelle generate dalla nostra app come ad esempio RecordTabella
- **VISTA**
Insieme di classi che fondamentalmente si occupano di fornire un'interfaccia grafica reattiva attraverso la quale interagiscono con l'utente, come ad esempio tutte le **Activity e le Dialog**. Nella nostra app, e come spesso accade, i confini tra i ruoli dei vari componenti del modello non sono sempre netti: infatti per avere un codice più compatto abbiamo deciso che ogni attività implementa anche un appropriato Listeners in ascolto sui rispettivi widget (Button piuttosto che elementi delle ListView) appartenenti al layout dell'attività stessa. Compito che forse concettualmente sarebbe più corretto assegnare ad un Controller.
- **CONTROLLER**
Insieme di classi che in base agli input degli utenti e agli eventi consegnatogli dal S.O. manipolano i dati del modello sottostante e li forniscono alle Viste del modello.

4.5 GESTIONE GRAFICO

4.5.1 Utilizzo Libreria esterna MPChartAndroid

Come da specifiche per **disegnare sullo schermo un grafico a linee** rappresentante ad un indicatore si è utilizzata la libreria esterna MPChartAndroid.

La classe GraficoActivity si comporta esattamente come le altre attività della fase di ricerca, solo che **una volta in possesso dei suoi dati** come oggetti POO li manipola diversamente: in particolare invece di fornirli ad un BaseAdapter per mostrarli in una ListView, li **dà in pasto alle API della libreria MPChartAndroid**.

I dettagli del suo utilizzo sono forniti con il codice sorgente.

Ci preme solo sottolineare che abbiamo investito tempo nel cercare di **offrire un grafico personalizzato**, piuttosto che lasciare lo stile di default: per esempio abbiamo utilizzato dei

formattatori personalizzati sia per gli assi x e y, che per i valori disegnati all'interno del grafico; e via dicendo.

4.5.2 Salvataggio Grafico in formato PNG

Per mancanza di tempo e forse per nostra incapacità non siamo riusciti ad inserire altre informazioni che a noi interessava mostrare direttamente nella View che rappresenta il LineChart: in particolare era nostro desiderio inserire oltre alla Leggenda anche il Titolo e il Sottotitolo.

Abbiamo **ovviato a questo problema** inserendo il LineChart in un LinearLayout che funge da contenitore e che include al suo interno due TextView rispettivamente per il titolo e il sottotitolo.

Quindi per il salvataggio del grafico in un file.png non è stato possibile utilizzare la comoda API della libreria MPChartAndroid


```
public Bitmap getChartBitmap()           /*ritorna un Bitmapa rappresentante il LineChart*/
```

ma in realtà verrà codificato in un Bitmap e poi salvato come File.png, non solo il LineChart, ma tutto il LineaLayout che lo contiene, con annesso titolo e sottotitolo, come mostrato sotto:

```
private class SalvaGraficoTask extends AsyncTask< Chart, Integer, String > {  
    .....  
  
    @Override  
    protected String doInBackground(Chart ... params) {  
        ....  
        LinearLayout view = findViewById(R.id.linearLayoutGrafico);  
        Bitmap bitmap = Bitmap.createBitmap(view.getWidth(), view.getHeight(), Bitmap.Config.ARGB_8888);  
        Canvas canvas = new Canvas(bitmap)           /*disegna la vista nel canvas che a sua volta avvolge il  
                                                         bitmpa*/  
  
        view.draw(canvas);  
        FileOutputStream outputStream;                /*stream per scrivere il grafico bitmap sul disco*/  
        .....  
        try{  
            /*apre 1 stream in scrittura verso 1 file nello storage interno, privato e se il file non esiste lo crea*/  
            outputStream =  
                openFileOutput(nome_file_png, Context.MODE_PRIVATE);  
            .....  
            if(bitmap.compress(Bitmap.CompressFormat.PNG, 80 , outputStream)){  
                .....  
            }  
        }  
    }  
}
```

4.6 GESTIONE MENÙ PAGINA PRINCIPALE

Per soddisfare il requisito di fornire un AppBar user-friendly si è deciso di implementare nella stessa AppBar della pagina principale un OptionMenu.

Il menu () si trova in alto a destra e permette di accedere alle seguenti informazioni:

1. Sito della worldBank
2. Informazioni sull'App
3. Contatti

Vediamo l'**implementazione**:

Per impostazione predefinita, ogni attività supporta un menu di opzioni e Android fornisce un formato XML standard per definire le voci di menu.

E' possibile aggiungere elementi a questo menu e gestire i clic sugli elementi stessi.

Il modo più semplice (e da noi adottato) per aggiungere voci di menu è "gonfiare" la risorsa xml del menu definita nella cartella *res\menù* (vedi di seguito paragrafo Localizzazione), nell'attività tramite la classe *MenuInflater*.

Dopodichè quando l'utente seleziona un elemento dal menu delle opzioni, il sistema chiama il metodo *onOptionsItemSelected()* che riceve come parametro il MenuItem selezionato.

Caricamento e gestione della **risorsa menù xml** nell'attività:

```
.....
@Override
public boolean onCreateOptionsMenu(Menu menu){
    getMenuInflater().inflate(R.menu.menu_di_scelta, menu); /*"gonfia" la risorsa menù.xml*/
    return true;
}

.....
@Override
public boolean onOptionsItemSelected(MenuItem item){
    .....
    int id = item.getItemId();
    switch(id) {
        case R.id.Menu_1:
            .....
            break;
        case R.id.Menu_2:
            .....
            break;
        case R.id.Menu_3:
            .....
            break;
    }
}
```

```

        return false;
    }

```

Per quanto riguarda le informazioni del menu di cui i punti 2 e 3, sono state implementate delle **Dialog** (v.Gestione delle Dialog).

Per il punto 1 invece si hanno due possibilità:

- (1) implementare un intent esplicito e quindi si decide a priori quale attività lanciare passandogli i dati con il metodo *putExtra()* della classe Intent e inoltre bisogna costruire il layout XML dell'activity.
- (2) implementare un intent implicito e quindi lasciare al sistema operativo la scelta dell'applicazione da lanciare. In questo caso visto che si tratta di applicazioni di sistema non bisogna costruire nessun layout XML.

Per sfruttare una delle funzionalità più importanti di Android si è scelta la soluzione (2) come mostrato nel frammento di codice seguente:

```

@Override
public boolean onOptionsItemSelected(MenuItem item){
    .....
    int id = item.getItemId();
    switch(id) {
        case R.id.Menu_1:
            intent = new Intent(Intent.ACTION_VIEW);
            intent.setData(Uri.parse(res.getString(R.string.WORLDBANK_SITE)));
            /* If there are more than one browser installed, a window will appear to allow the user to
            choose what he prefers.*/
            startActivity(Intent.createChooser(intent,
                res.getString(R.string.CHOOSE_BROWSER)));
            break;
        .....
    }
    return false;
}

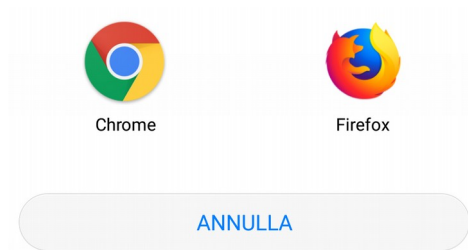
```

Come si può vedere dal codice, prima si decide l'azione da compiere: nel nostro caso la costante predefinita **ACTION_VIEW**, a cui deve saper rispondere l'eventuale applicazione presente nel dispositivo (per **visualizzare** la **pagina web** di **worldBank**).

Poi si setta la tipologia dei dati che dovrà gestire l'applicazione nel momento in cui risponderà alla chiamata: nel nostro caso l'url del sito di worldBank.

Dopo aver creato l'**intento** e aver impostato le informazioni aggiuntive, il MainActivity chiama *startActivity()*.

Se il sistema identifica **più di un'attività in grado di gestire l'intento**, visualizzerà una finestra di dialogo (a volte indicata come "**finestra di disambiguazione**") per consentire all'utente di selezionare l'app da utilizzare, come mostrato nella seguente figura:



Se invece è presente solo una attività che può gestire l'intento, il sistema lo avvierà immediatamente.

Al contrario se non è presente nessuna applicazione che può gestire l'intento verrà visualizzata una finestra con un messaggio impostato dal sistema per informare l'utente che l'azione non può essere portata a termine.

4.7 LOCALIZZAZIONE

Abbiamo utilizzato fortemente la localizzazione delle risorse perché l'esternalizzazione delle stesse ci ha permesso una manutenzione e aggiornamento indipendente dal codice.

Fondamentalmnte **1 modifica ad 1 risorsa si ripercuote su n modifiche** ad oggetti che richiamano tale risorse.

Tanto per fare un esempio, **l'utilizzo della localizzazione** di Android ci ha **aiutati** a creare uno stile estetico personalizzato e omogeneo alla nostra app: **definendolo un'unica volta nella cartella \res e impostando il riferimento ad esso in tutte le View che lo richiedono** per conformizzarsi allo stile dell'app.

Nei svariati tentativi di dare all'app una veste di nostro gradimento, modificare il codice in un solo punto piuttosto che in n punti, ha facilitato il raggiungimento del nostro obiettivo

In particolare abbiamo utilizzato le seguenti sottosottocartelle della cartella "res":

Values:

Come accennato, l'utilizzo di questa cartella ci ha permesso per esempio di formattare diversi *stili* per poi propagarli a secondo della funzione nei diversi layout, (es. i button hanno nei diversi layout tutti lo stesso stile (<style name="styleButton">).

Stesso discorso ovviamente vale per il file *strings* e integers sempre da noi utilizzati.

Sottolineiamo che, solo per questioni di tempo, non abbiamo implementato il **meccanismo dei modificatori**, il quale avrebbe permesso alla piattaforma android di caricare all'avvio della nostra

app i contenuti della cartella **values-xx** (dove **xx** è il suffisso della lingua) appropriati, in base alla lingua con cui è impostato il dispositivo.

Per esempio :

res/values/strings.xml: default (ad esempio inglese)

res/values-it/strings.xml: lingua italiana

res/values-fr/strings.xml: lingua francese

Quindi non sfruttiamo questo meccanismo che ci permette di avere un app multilingue (e di sfruttare quindi a pieno tutte le potenzialità della localizzazione e della cartella values), ma “permettiamo all’app di essere internazionale” utilizzando solo contenuti in lingua inglese nella cartella di default.

Infine, abbiamo sfruttato anche il file *colors* per inserire dei colori formattati direttamente da noi ed utilizzati nel codice.

Drawable:

In questa cartella abbiamo inserito tutte le immagini in senso lato utilizzate nell’app: le immagini, i background e i border utilizzati in appositi layout e nei button.

<*Button*

```
...
    android:background="@drawable/background_grafico"
..
.. />
```

A differenza della cartella *values*, in questo caso, grazie all’utilizzo dello strumento fornito da Andorid Studio denominato “**Asset Studio**”, per le immagini sull’**actionBar** abbiamo sfruttato il meccanismo dei qualificatori (caratteristica anche di questa cartella con le relative sottocartelle *drawable ldpi, mdpi, hdpi, etc*) per gestire le diverse densità di risoluzione dei diversi display che il sistema riconoscerà, e quindi sfruttare a pieno, in questo caso, tutta la potenzialità della localizzazione.

Mipmap:

Ovviamente come qualsiasi app che si rispetti, abbiamo creato e inserito in questa cartella un’**icona di lancio personalizzata** dell’app.

Anche qui, come sopra, per la creazione e modifica di queste immagini in varie risoluzioni abbiamo utilizzato il tool “**Asset studio**”.

Menù:

Qui abbiamo inserito i due file.xml che rappresentano i due distinti layouts dei menù (caricati tramite inflazione) nell’appBar del MainActivity e nell’appBar di GraficoActivity.

Layout:

Abbiamo inserito in tale sottocartella tutti i file.xml rappresentanti i layout (con i propri layout, view e widget interni) di tutte le interfacce grafiche dell'attività, sia in **portrait** che in **landscape** dove necessario.

Tutti i layout delle interfacce grafiche da noi sviluppate sono **implementati come file XML, seguendo il consiglio della guida degli sviluppatori di Android.**

4.8 GRAFICA

Basandoci sull'analisi dei requisiti abbiamo cercato di sviluppare delle UI capaci di adattarsi alle possibili diverse dimensioni degli schermi dei dispositivi, mantenendo lo stesso stile e le stesse proporzioni.

In particolare per soddisfare la **parte del requisito** che richiedeva che ogni schermata deve mantenere lo stesso **stile e proporzioni** (ad esempio allineare i Button di selezione sempre in fondo al layout, dare a tutti i widget una veste grafica uniforme) **sia nelle diverse dimensioni degli schermi che nei due possibili orientamenti che l'app deve offrire** abbiamo:

- evitato di usare dimensioni fisse per le view, dimensionandole invece solo con attributi dinamici quali **match parent, match constraint e wrap content** per adattarsi in altezza e larghezza rispettivamente al genitore, ai vincoli o al contenuto rispettivamente. Ove necessario sempre nell'ottica di mantenerle le proporzioni si è utilizzato anche il parametro **“peso”** per permettere un ulteriore adattamento dinamico.
- Tuttavia **solo questa scelta** implementativa **non è stata sufficiente** a soddisfare il requisito nella sua interezza. Infatti anche utilizzando solo attributi dinamici, alcuni widgets hanno una loro dimensione minima fissata, come per esempio le TextView dove l'altezza minima fissata è data dalla dimensione con cui viene scritto il contenuto. Per cui **per schermi relativamente molto piccoli**, questo può comportare che il widget possa comunque essere disegnato in parte o completamente fuori dal proprio contenitore, tanto da non rendere intuibile la sua presenza da parte dell'utente. Quindi, dove non presenti di default (MainActivity e TableLayout) sono state inserite delle **ScrollView** per permettere all'utente di venire a conoscenza che il layout non “termina” con lo schermo, ma può essere “scollato” (verso il basso nel nostro caso).

Nei layout delle Activity che mostrano Liste non è stato invece necessario utilizzare una ScrollView una in quanto già contenuta nella ListView stessa).

- Come descritto in Localizzazione, abbiamo sfruttato la stessa nel processo di assegnamento di una veste (puramente estetica) coerente a tutte le View, compresa l'appBar e i suoi menù. Processo che ha visto l'assegnazione di margini esterni ed interni standard su tutti i widget, stesso background nei widget logicamente analoghi, stesse dimensioni delle immagini nelle appBar delle diverse attività, etc.
In definitiva, abbiamo cercato anche per la grafica di curare ogni aspetto, infatti queste scelte sono stato frutto di diversi test su diversi emulatori e device.

Per soddisfare la **seconda parte del requisito** che richiedeva una **prospettiva diversa** dei contenuti in landscape (di norma l'utente gira lo schermo per vedere più grande, quanto meno le immagini) abbiamo dovuto implementare per il MainActivity un layout distinto per entrambi gli orientamenti (*layout/land*).

Inquanto tale soluzione è stato l'unico modo che empiricamente ci ha permesso di ottenere quello che volevamo: le **immagini vengono scalate a seconda dell'orientamento** del telefono (comportamento analogo al visualizzatore d'immagini di sistema), mantenendo le corrette proporzioni.

Utilizzando invece un solo layout l'immagine del main non scalava come desiderato anche settando i possibili attributi dedicati.

Inoltre in fase di sviluppo abbiamo testato l'interfaccia della GraficoActivity con diversi tipi di layout, e alla fine abbiamo optato per l'inserimento delle azioni di salvataggio del grafico e dei dati dell'indicatore:

- Save Date SQLite
- Save Image.png

nel menù localizzato sull'ActionBar, per permettere una visualizzazione del grafico più ampia.

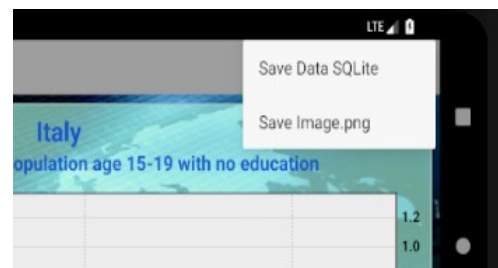
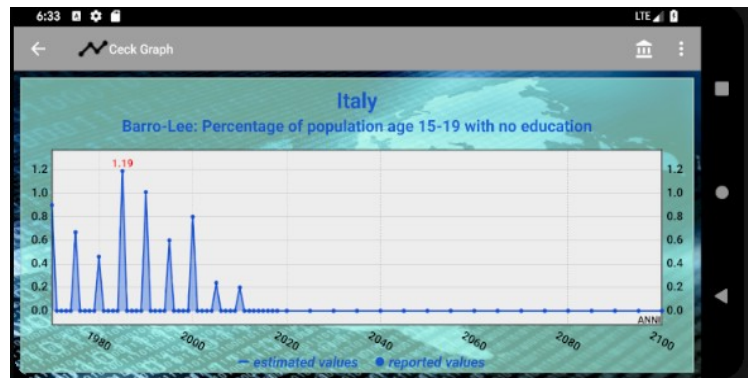
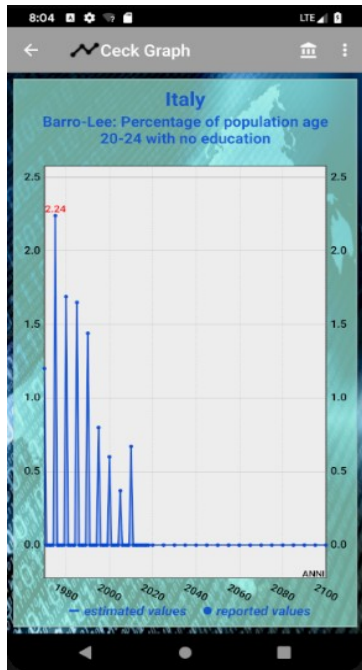
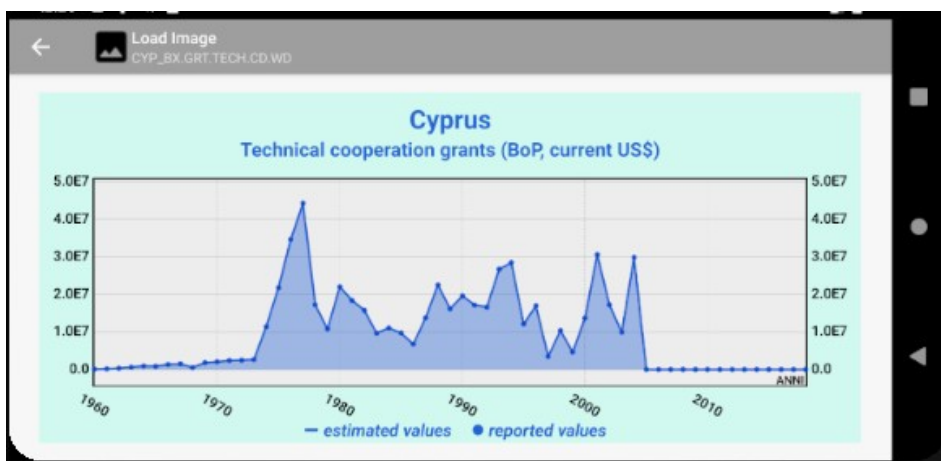


Fig. 8: Visualizzazione GraficoActivity in portrait e landscape



Infine approfittiamo di questo paragrafo per ricordare che l'immagine del grafico salvata in PNG è l'immagine presa nel momento in cui l'utente l'ha salvata: un'istantanea del **LineChart**.



Proprio per concludere diamo un accenno alla **costruzione dei Layout**:

- fondamentalemnte dove ritenuto necessario, come ad esempio nel layout del MainActivity è stato utilizzato un *ConstraintLayout*.
In quanto il sito degli sviluppatori di Android consiglia di preferirlo in caso di layout particolarmente annidati perché più efficiente e più flessibile fornendo gerarchie di viste di tipo piatte.
- Per quanto riguarda invece le viste delle righe caricate dinamicamente dalle varie ListView abbiamo utilizzato fondamentalmente dei *LinearLayout* perché data la semplicità delle viste da caricare ci sembrava il più appropriato.

5 PIATTAFORMA, SVILUPPO E TESTING

Si è scelto di configurare **minSdkVersion = 23** (API LEVEL) per permettere alla nostra app di girare correttamente su tutti i dispositivi con versione S.O. \geq Marshmallow 6.0.

Questo perché prendendo tale livello di API come limite inferiore, si ha un buon compromesso tra % di dispositivi con installato un S.O. versione \geq 6.0 (circa il 70% attualmente, che tenderà ad aumentare) e le nuove funzionalità e caratteristiche delle più recenti piattaforme Android.

Negli numerosi test effettuati si è verificato che effettivamente la nostra app funziona correttamente su piattaforme Android con API level \geq 23.

Inoltre come descritto nell'analisi dei requisiti, ogni schermata deve mantenere lo stesso stile e proporzioni nella maggioranza delle **diverse dimensioni degli schermi, sia in portrait che landscape.**

Grazie alle scelte progettuali descritte nel paragrafo Grafica del precedente capitolo, mostriamo di seguito che l'app soddisfa il requisito su diversi emulatori e device.

Fig 8. MainActivity in portrait e landscape su emulatore Galaxy Nexus 4,65 pollici per mostrare lo stesso stile e stesse proporzioni in entrambi gli orientamenti.

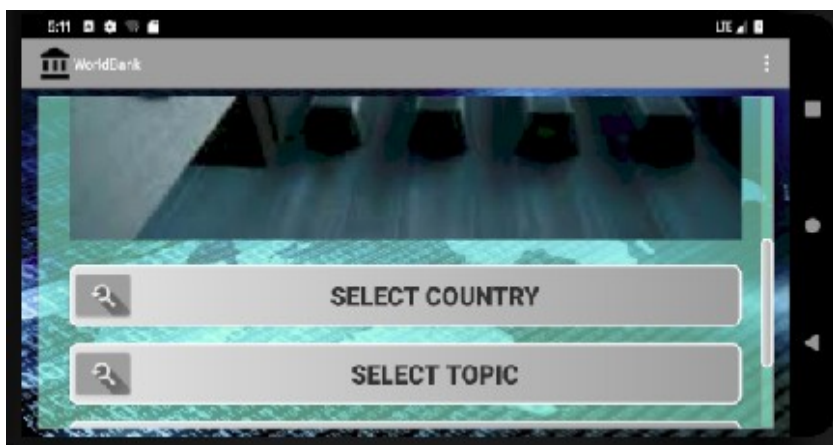
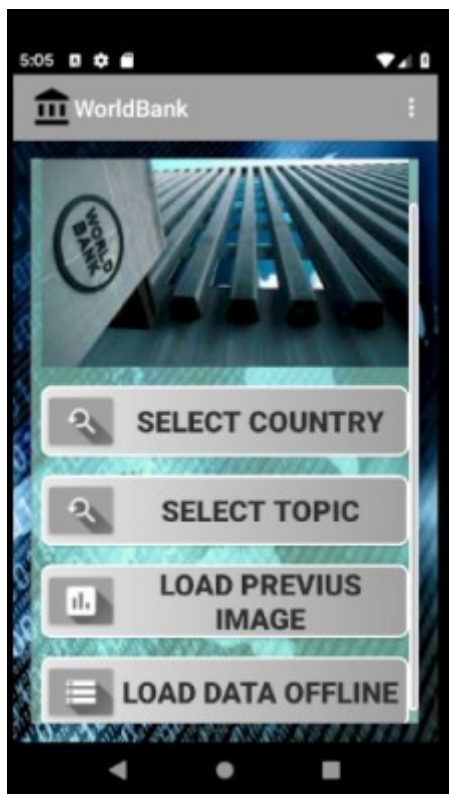


Fig. b) MainActivity in portrait e landscape su emulatore Pixel 2 XL 5,99 (la 2° immagine a destra) pollici per mostrare lo stesso stile e stesse proporzioni in schermi di diverse dimensioni. In particolare si può notare che l'emulatore in Fig. a) Galaxy Nexus 4,65 pollici presenta la ScrollBar

Fig.b

Fig.a

