

Progetto di Programmazione di Sistemi Embedded

Carraro Eddie, Corrò Alessandro, Maritan Nicola

19 giugno 2023

Indice

1 Introduzione	4
1.1 Widget in Android	4
1.1.1 Widget e Home screen widget	4
1.1.2 Home screen widget in Android	4
1.1.3 Widget negli ultimi anni	5
1.2 Storia dei Widget in Android	5
1.2.1 Gli albori	5
1.2.2 Accesso di terzi all'API dei widget	5
1.2.3 Widget oggi	6
2 Panoramica sui widget	7
2.1 Tipologie di widget	7
2.1.1 Widget informativi	7
2.1.2 Widget di collezioni	7
2.1.3 Widget di controllo	8
2.1.4 Widget ibridi	8
2.2 Limitazioni dei widget	9
2.2.1 Gestì	9
2.2.2 Elementi	9
2.2.3 Consumo della batteria	10
2.2.4 Frequenza di aggiornamento	10
2.3 Scelte di progettazione	11
2.3.1 Contenuto	11
2.3.2 Navigazione	11
2.3.3 Ridimensionamento	11
2.4 Novità in Android 12	12
2.4.1 Cambiamento dinamico dei colori con Material You	12
2.4.2 Aggiornamento dei widget di Google	14
2.4.3 Freeform	15
2.4.4 Gli Smart Widget di Samsung	16
3 Struttura dei widget in Android	17
3.1 Componenti	17
3.1.1 Classe AppWidgetProviderInfo	18
3.2 Classe AppWidgetProvider	19
3.2.1 Dichiarazione nel manifest	19
3.2.2 Implementazione	20
3.2.3 Classe PendingIntent	21
3.2.4 Gestione degli eventi in onUpdate()	21
3.2.5 Ricevere Intent broadcast	22

3.3	Widget Layout	22
3.3.1	Classe <code>RemoteViews</code>	22
3.3.2	Angoli arrotondati	23
3.4	Mostrare collezioni	23
3.4.1	Implementazione	24
3.4.2	Classe <code>AppWidgetProvider</code> per i widget di collezioni	25
3.4.3	Dati persistenti	26
3.4.4	Interfaccia <code>RemoteViewsFactory</code>	26
3.4.5	Aggiungere azioni di un singolo item	27
3.4.6	Step di un <code>RemoteViewsFactory</code>	29
3.4.7	<code>RemoteCollectionItems</code>	29
3.5	Widget avanzati	29
3.5.1	Ottimizzazioni nell'aggiornamento	29
3.5.2	Aggiornamento del widget in risposta ad un evento broadcast	31
3.6	Miglioramenti applicabili ad un widget	31
3.6.1	Widget picker	32
3.6.2	Aggiungi una descrizione al widget	33
3.6.3	Transizioni più fluide	33
3.6.4	Modifiche a runtime di <code>RemoteViews</code>	34
3.7	Layout flessibili	34
3.7.1	Layout responsivi	34
3.7.2	Layout esatti	37
3.8	Configurazione del widget da parte dell'utente	38
3.8.1	Dichiarazione della activity di configurazione	38
3.8.2	Implementazione della activity	38
3.8.3	Riconfigurazione di un widget	39
3.8.4	Pin di un widget	40
3.9	Widget host	40
3.9.1	Binding dei widget	41
3.9.2	Responsabilità dell'host	41
4	Cypher	42
4.1	Struttura dell'app	43
4.1.1	Primo accesso	43
4.1.2	Schermata principale	43
4.1.3	Aggiunta di credenziali	44
4.1.4	Statistiche	45
4.1.5	Impostazioni	46
4.1.6	Implementazione delle liste	46
4.1.7	Model-View-ViewModel	48
4.1.8	Crittografia	49
4.1.9	Hashing	49
4.1.10	Internazionalizzazione	49
4.2	Widget	49
4.2.1	Anteprime	50
4.2.2	Credentials widget	50
4.2.3	Stats widget	56
4.2.4	Add widget	59
4.3	Conclusioni	60

Capitolo 1

Introduzione

Nel 1984, Apple introdusse nell'OS MacOS per Apple Macintosh i *Desktop Accessories* (DA) [27]. L'obiettivo era condensare in una porzione di schermo una applicazione *single-purpose*, come una calcolatrice, un orologio o una nota [1]. A causa delle limitazioni tecnologiche del tempo, le DA dovevano essere leggere e perciò potevano offrire poche funzionalità. Nel 2009, più di 20 anni dopo, Android introdusse la possibilità di sviluppare *widget* per le proprie applicazioni. I widget occupano una porzione della home screen dell'utente e possono offrire un *primo assaggio* alle app complete, ad esempio mostrare gli eventi futuri del calendario o visualizzare i dettagli di una canzone in riproduzione in background. Ciò avvenne per la prima volta con l'SDK 1.5 e l'AppWidget framework [47]. Ancora oggi [10] i widget sono fondamentali per realizzare funzionalità accessibili direttamente dall'home screen. Deve essere data la possibilità di muoverli e ridimensionarli a seconda delle necessità dell'utente.

1.1 Widget in Android

Di seguito si tratta la presenza dei widget in Android.

1.1.1 Widget e Home screen widget

In Android, il termine "widget" è, in alcuni contesti, ambiguo. Infatti, esso può far riferimento a:

- Un elemento UI del package `android.widget` che eredita dalla classe `View`. Esempi di widget sono i tipici `Button`, `EditText`, `ImageView`, etc. Invece, non vengono considerati widget in tal senso degli elementi UI che sono dei "contenitori" [72], come `ViewGroup` o `ListView`.
- Un elemento piazzato nella home screen dell'utente, con lo scopo di fornire funzionalità di supporto ad una applicazione. In tale contesto, si parla più precisamente di *home screen widget*. Nel report si tratterà lo sviluppo di home screen widget in Android. D'ora in poi, qualora non venga specificato, si farà riferimento a questi elementi parlando di widget.

1.1.2 Home screen widget in Android

Android supporta il posizionamento di widget fin dal SDK 1.5 (2009) [47]. Per aggiungere un widget alla schermata principale del dispositivo, gli utenti possono semplicemente eseguire una serie di azioni specifiche, come ad esempio premere a lungo su uno spazio vuoto della schermata, selezionare l'opzione "Aggiungi widget" e quindi scegliere il widget desiderato

dall'elenco disponibile. L'importanza dei widget si evince dalla mole di tutorial e guide presenti nel web [4], soprattutto per gli utenti meno esperti [19].

1.1.3 Widget negli ultimi anni

I widget si sono affermati con decisione nel mondo mobile [42, 10]. Infatti, in Android 12 (2021) sono state introdotte diverse funzionalità per permettere una maggiore configurazione e personalizzazione dei widget [30, 55].



Figura 1.1: Widget Android nella home screen. Tratto da [6].

1.2 Storia dei Widget in Android

1.2.1 Gli albori

La versione iniziale di Android non includeva la funzionalità dei widget, ma Google si rese presto conto dell'importanza di fornire agli utenti un modo più diretto e personalizzato per interagire con le app.

Nel 2009, con l'introduzione di Android 1.5 Cupcake, Google introdusse i primi widget ufficiali nel sistema operativo. Questi widget erano limitati alle applicazioni predefinite fornite da Google, come l'orologio, il calendario e i contatti. Tuttavia, fornivano agli utenti la possibilità di visualizzare informazioni chiave senza dover aprire le app corrispondenti.

1.2.2 Accesso di terzi all'API dei widget

Con il passare del tempo, Google ha continuato a migliorare il supporto dei widget in Android. Con l'avvento di Android 2.1 Éclair nel 2010, gli sviluppatori di terze parti ottennero l'accesso alle API dei widget, consentendo loro di creare e distribuire le proprie creazioni tramite il Google Play Store.

Questa apertura agli sviluppatori ha portato a una proliferazione di widget creativi e utili per una vasta gamma di scopi. Gli sviluppatori hanno sfruttato le funzionalità dei widget per offrire una vasta gamma di informazioni e funzionalità come previsioni del tempo, feed di notizie, controlli musicali, strumenti di produttività e molto altro ancora.

Con l'introduzione di Android 4.0 Ice Cream Sandwich nel 2011 e versioni successive, Google ha continuato a migliorare il supporto dei widget. Sono state introdotte nuove dimensioni e layout per consentire agli sviluppatori di creare widget più flessibili e personalizzabili.

1.2.3 Widget oggi

Oggi, i widget sono diventati una parte essenziale dell'esperienza utente di Android. Gli utenti possono scegliere tra una vasta gamma di widget disponibili sul Google Play Store e personalizzare le loro schermate principali in base alle loro preferenze. Google stessa ha continuato a offrire widget predefiniti per le sue applicazioni, come Gmail, Google Calendar e Google Drive, fornendo agli utenti un accesso rapido alle informazioni e alle funzionalità più utilizzate.

In conclusione, i widget Android sono diventati una caratteristica distintiva del sistema operativo, consentendo agli utenti di personalizzare le loro esperienze utente e di accedere rapidamente a informazioni e funzionalità importanti senza dover aprire le applicazioni corrispondenti. Grazie alla continua evoluzione del sistema operativo e alla creatività degli sviluppatori, i widget Android continuano a essere uno strumento potente e versatile.

Capitolo 2

Panoramica sui widget

In [10] sono descritte le caratteristiche base dei widget in Android.

2.1 Tipologie di widget

Di seguito vengono trattati i principali tipi di widget. Si noti che la validità di tale discussione non è strettamente vincolata alla piattaforma Android, ma è bensì estendibile anche ad iOS ed altri Sistemi Operativi.

2.1.1 Widget informativi

I widget informativi mostrano tipicamente elementi di informazione cruciali e tengono traccia di come tali informazioni cambiano nel tempo. Esempi di widget informativi includono widget del meteo [51], widget dell'orologio [29] o widget per il tracciamento dei punteggi sportivi. Toccando un widget informativo, di solito si avvia l'applicazione associata e si apre una vista dettagliata delle informazioni del widget. Essi forniscono una rapida visualizzazione delle informazioni pertinenti senza la necessità di aprire l'applicazione completa. I widget informativi sono progettati per fornire un'esperienza utente efficace ed efficiente, consentendo agli utenti di ottenere informazioni importanti in modo rapido e intuitivo. Deve essere possibile modificare la dimensione [56], la posizione e lo stile di un widget informativo per soddisfare le esigenze estetiche o funzionali dell'utente.

2.1.2 Widget di collezioni

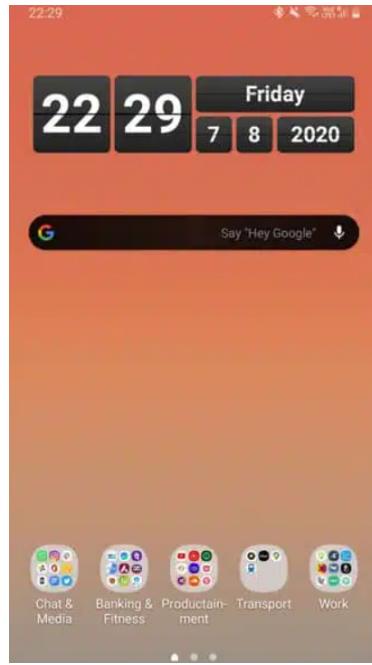
I widget di collezioni si occupano di mostrare a schermo molteplici elementi dello stesso tipo, per esempio collezioni di immagini della galleria [33], una collezione di email [34] oppure una collezione di appuntamenti e eventi [36]. Tali widget spesso sono scorribili verticalmente per visualizzare tutti gli elementi della collezione. I widget di Google Calendar sono riportati in Figura 2.3

Tipicamente, i widget di collezioni permettono di:

- Esplorare la collezione;
- Aprire un elemento della collezione per vederne i dettagli associati;
- Interagire con gli elementi, ad esempio per contrassegnarli.



(a) Widget meteo



(b) Widget orologio

Figura 2.1: Esempi di widget informativi

2.1.3 Widget di controllo

Un widget di controllo si occupa di mostrare a schermo delle funzioni di operazioni utilizzate frequentemente, cosicché l'utente possa usufruirne direttamente dalla home screen, senza aprire l'applicazione. Un esempio di widget di controllo è un widget di controllo per elementi di una smart home. Interagire con un widget di controllo potrebbe aprire una vista associata nell'applicazione.

2.1.4 Widget ibridi

Molti widget possono essere classificati come appartenenti a più tipologie. Per esempio, un player di musica [13] è principalmente un widget di controllo, ma mostra a schermo informazioni sul brano e sull'artista come un widget informativo.

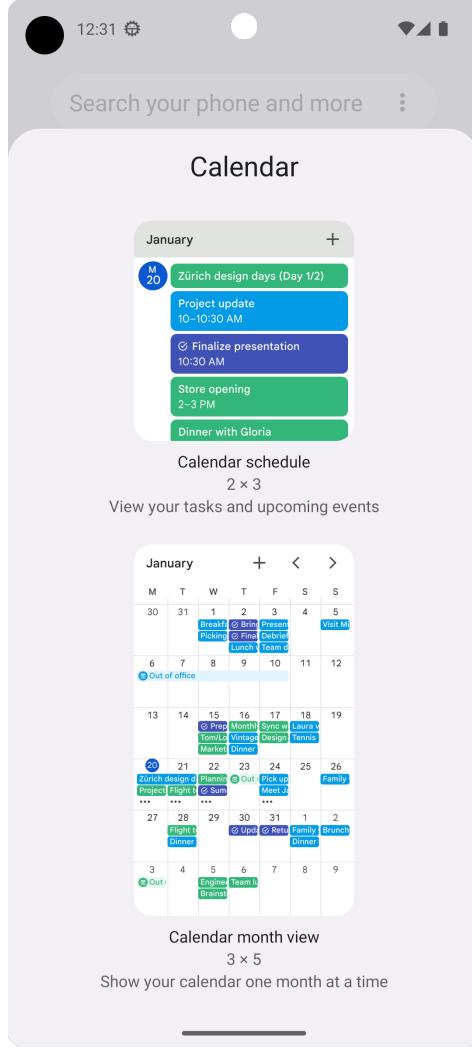


Figura 2.2: Widgets di Google Calendar. Screenshot da emulatore Android.

2.2 Limitazioni dei widget

2.2.1 Gestì

Poichè i widget risiedono nella schermata iniziale, sono obbligati a coesistere con la navigazione di quest'ultima. Ciò limita pesantemente l'utilizzo di gesti, a differenza di quanto accade in una normale applicazione. Gli unici gesti disponibili sono *touch* e *vertical swipe*. Con queste modalità di input disponibili, è possibile cliccare su dei pulsanti o scorrere delle liste.

2.2.2 Elementi

Date le limitazioni sui gesti disponibili per i widget, alcuni elementi costitutivi dell'interfaccia utente che si basano su gesti limitati non sono disponibili per i widget. Per un elenco completo degli elementi costitutivi supportati e ulteriori informazioni sulle restrizioni di layout in Android, consultare [25] e [54].

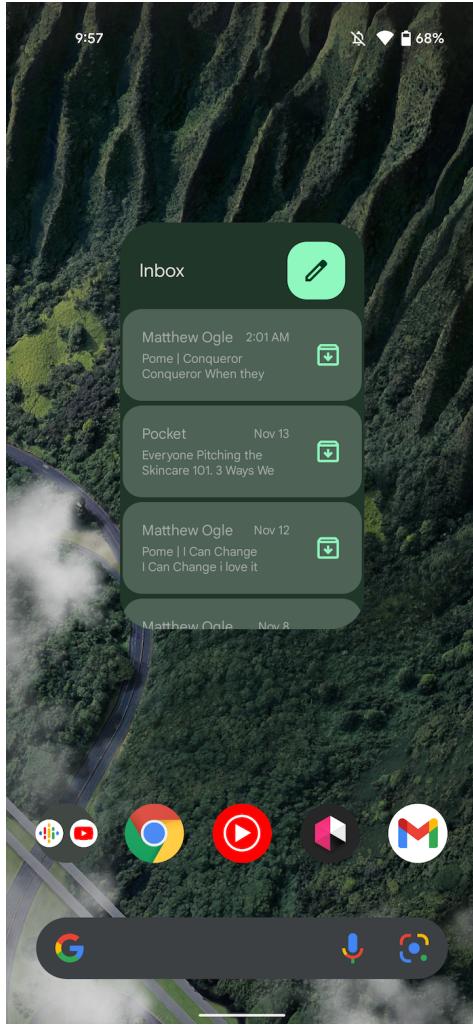


Figura 2.3: Widget *Inbox* di Gmail. Tratto da [35].

2.2.3 Consumo della batteria

A causa del tempo di durata della batteria degli smartphone molto ridotto rispetto ai telefoni cellulari di qualche decina di anni fa, gli utenti hanno sempre più cercato di ridurre i consumi e di evitare l'utilizzo di tools energivori per il proprio smartphone. In ambito di widget l'impatto sulla batteria dipende dal caso particolare. L'aspetto principale da considerare è il periodo di aggiornamento del widget [22]. Aggiornamenti sporadici permettono un consumo limitato della batteria, mentre widget che necessitano di aggiornamenti frequenti consumeranno di più (nonostante i costi siano comunque ridotti).

2.2.4 Frequenza di aggiornamento

Si è visto come una elevata frequenza di aggiornamento del widget porti ad un maggiore utilizzo della batteria. D'altro canto, se essa è troppo bassa, le informazioni presenti nel widget (soprattutto per determinate tipologie di questi ultimi) rischiano di non essere più valide. Di conseguenza, nasce l'esigenza di selezionare una frequenza di aggiornamento che bilanci il consumo della batteria e il rinnovamento della vista del widget.



Figura 2.4: Widget di collezione di immagini della galleria in iOS.

2.3 Scelte di progettazione

2.3.1 Contenuto

I widget costituiscono un espeditivo per attirare l’utente ad utilizzare la relativa app pubblicizzandone le novità. Così come un articolo civetta, i widget concentrano l’attenzione dell’utente sulle informazioni dell’app e ne forniscono una connessione diretta. A tal proposito, è importante che l’app mostri più informazioni di quanto non lo faccia il relativo widget.

2.3.2 Navigazione

In aggiunta alle mere informazioni sul contenuto, il widget deve essere in grado di collegarsi direttamente alle funzionalità dell’app più utilizzate [10]. Ciò permette all’utente di completare un’azione saliente in modo rapido, oltre che a estendere il raggio d’azione dell’app stessa. In particolare, le caratteristiche principali di navigazione da fornire all’utente sono:

- La possibilità di avere accesso alle funzioni generative, ovvero quelle funzioni che permettono all’utente di creare nuovo contenuto per l’app;
- L’apertura dell’app ad inizio pagina, in modo da offrire più flessibilità nella navigazione.

2.3.3 Ridimensionamento

Tenere premuto un widget ridimensionabile permette di ridimensionarlo. L’utente può utilizzare i lati e gli angoli del widget per impostarne la dimensione. È possibile creare un widget completamente ridimensionabile oppure limitarlo a cambiamenti lungo un solo asse [55]. Permettere all’utente di ridimensionare un widget rende l’esperienza migliore, dal momento che è lui stesso a scegliere quanti e quali contenuti visualizzare. In Figura 2.8b, 2.8a e 2.8c viene raffigurato un widget ridimensionabile in tre modi differenti, a seconda di quali servizi l’utente vuole usufruire.



Figura 2.5: Widget di controllo delle luci della propria casa. Tratto da [10].

2.4 Novità in Android 12

Android ha dedicato parte dell'aggiornamento Android 12 (API 31) al lato grafico del sistema operativo (gli aggiornamenti di natura tecnica sono descritti nel Capitolo 3), introducendo nuovi strumenti di personalizzazione. Per quanto riguarda i widget, le novità sono molteplici [6]:

- Vi è stato un arrotondamento degli angoli, che costringono gli sviluppatori ad intervenire rimuovendo qualsiasi sorta di informazioni, per l'appunto, negli angoli, poiché queste possono non essere mostrate venendo di fatto "tagliate"
- I widget ora possono cambiare tema in base a quello di sistema (tema chiaro o scuro) e introducono la variazione di palette di colori grazie alla Material You e alla palette scelta dall'utente per il sistema.
- Vi sono state modifiche sul sistema di ridimensionamento dei widget.
- Per quanto riguarda novità più pratiche, sono stati aggiunti dei nuovi componenti ai widget, ovvero la **CheckBox**, lo **Switch** e il **RadioButton**. Il widget risulta comunque essere **statico**, per cui ogni volta che si interagisce con uno degli elementi l'app dovrà salvare lo stato e rimanere in ascolto per cambi di stato futuri.

Per quanto riguarda il penultimo punto citato, si passa da un ridimensionamento "a pixel" ad uno "a blocchi". Di conseguenza, a partire dall'API 31 è necessario utilizzare il blocco sulla schermata home come unità di misura per determinare le dimensioni di un widget (è comunque necessario utilizzare anche la vecchia nomenclatura per avere retrocompatibilità).

2.4.1 Cambiamento dinamico dei colori con Material You

La novità più interessante per gli utenti che amano la personalizzazione è il cambiamento dinamico dei colori del widget in base allo sfondo e tema selezionati; tutto ciò è possibile



Figura 2.6: Esempio di widget ibrido in iOS.

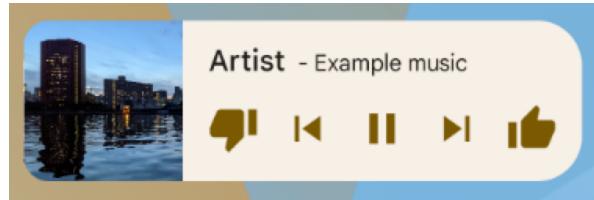


Figura 2.7: Widget ibrido player di musica. Tratto da [10].

grazie alla *Material You* [14], il più grande aggiornamento di design della storia di Android. Esso permette ai widget di rinascere e di essere, oltre che un elemento dalla profonda utilità, anche un modo per abbellire la UI dell’utente. Come si può notare in Figura 2.9, il widget di Google Maps utilizza i colori di sistema scelti dall’utente e cambia di conseguenza, risultando più coerente con il resto della UI.

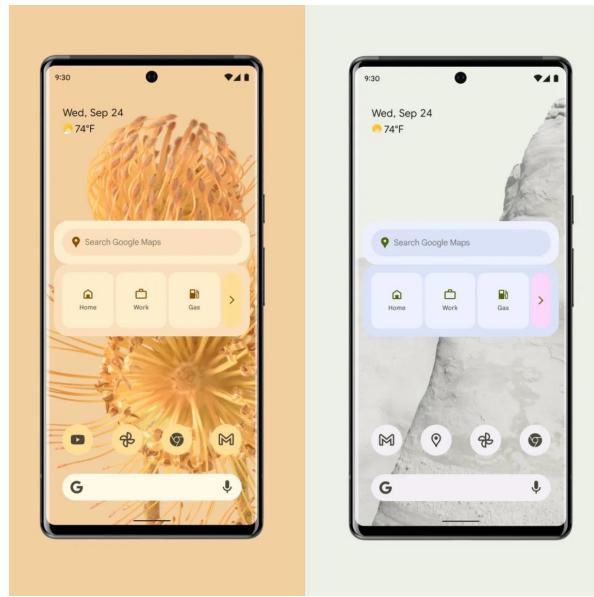


Figura 2.9: Esempio di widget che utilizzano Material You. Tratto da [14].



(a) Esempio di widget di dimensione minima



(b) Esempio di widget di media dimensione



(c) Esempio di widget grande

Per utilizzare i colori dinamici, è necessario utilizzare il tema di default di sistema inserendo
`@android:style/Theme.DeviceDefault.DayNight` nel root layout in questione, oppure usando il tema della Material 3: `Theme.Material3.DynamicColors.DayNight`[\[32\]](#).

2.4.2 Aggiornamento dei widget di Google

In concomitanza con l'introduzione di Material You, Google ha aggiornato i suoi widget più utilizzati, ovvero quelli delle app Drive [\[37\]](#), Mail [\[34\]](#), Maps [\[38\]](#) e Google Photo [\[39\]](#). Questi supportano i colori di Material You, rendendoli molto più estetici e moderni delle versioni precedenti.

Un altro dei widget aggiornati da Google è quello di YouTube Music [\[76\]](#). Se la versione di dimensione 1x1 risulta essere semplicemente una comoda scorciatoia per interrompere la riproduzione della canzone corrente, un'istanza più grande permette all'utente di "sbirciare"

le canzoni subito successive (mostrando la copertina dell'album o del singolo) senza aprire l'app. Una immagine del widget viene riportata in figura 2.10.



Figura 2.10: Widget di YouTube music. Grandezze diverse del widget mostrano più o meno informazioni. Tratto da [73].

2.4.3 Freeform

Google ha introdotto un nuovo concetto di forma per i widget: il *freeform*. Degli esempi vengono riportati in Figura 2.11. Se nelle versioni passate i widget erano di forma rettangolare, con Android 12 vi è la possibilità di realizzare forme stravaganti e fuori dall'ordinario. Queste possono rappresentare certamente un ottimo mezzo di personalizzazione della UI, ma anche un modo alternativo per strutturare widget utili.



Figura 2.11: Nuova versione del widget di Google Photo, con varie forme applicate. Tratto da [73].

In Figura 2.12a, 2.12b e 2.12c si riporta un esempio del widget di Google Drive. In questo si vede un'istanza del widget di dimensione 2x2 a forma di quadrifoglio, con lo scopo di far risaltare le varie azioni possibili, tra cui la ricerca o l'aggiunta di un file nel Drive.

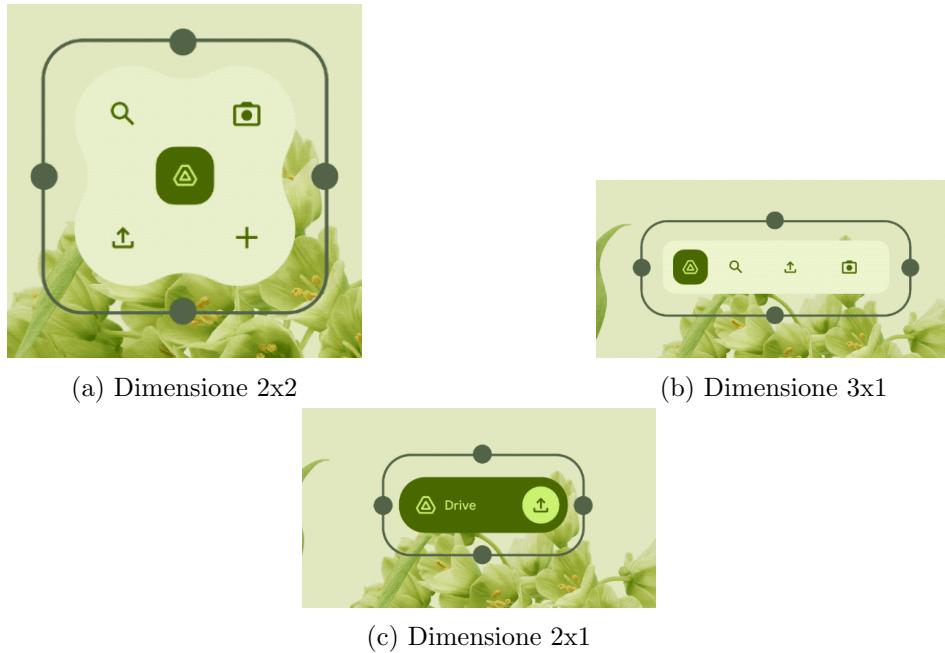


Figura 2.12: Nuova versione del widget di Google Drive, con vari esempi di dimensioni [73]

2.4.4 Gli Smart Widget di Samsung

Molte aziende produttrici di smartphone Android non usano la versione stock [71], bensì creano delle skin proprietarie per differenziare maggiormente i loro dispositivi da quelli di altri brand. Ad esempio, Samsung ha sviluppato la *One UI*. Nella versione 4.1 di quest'ultima sono stati introdotti gli *Smart Widget*, ovvero dei widget che possono essere "impilati" come un mazzo di carte. È possibile accedere ad un particolare widget con uno swipe verso destra o sinistra. L'idea è poi stata ripresa da Apple, permettendo così di risparmiare spazio "sovrapponendo" più widget in un'unica regione.

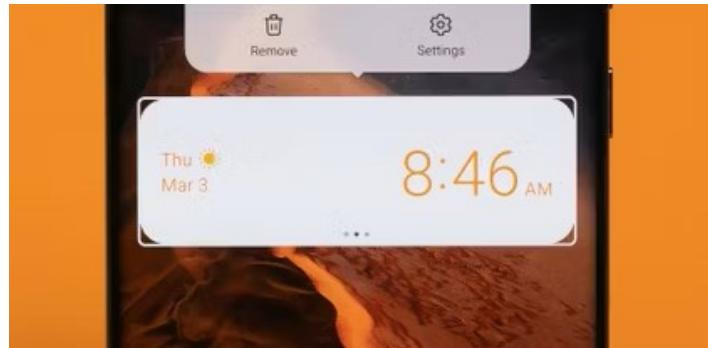


Figura 2.13: Esempio di Smart Widgets. Tratto da [43].

Come si può notare nell'Immagine 2.13 (ed in particolar modo nella parte inferiore della regione in cui è presente il widget con l'orario), vi sono 3 "puntini", che indicano il numero di widget condensati in quella regione di schermo.

Capitolo 3

Struttura dei widget in Android

Lo sviluppatore Android ha a disposizione una serie di API dedicate alla implementazione dei widget. In questo capitolo si tratta la struttura dei widget in Android e le sue componenti chiave. Vengono descritte le classi fondamentali e come queste interagiscono.

3.1 Componenti

Per creare un widget sono necessarie le seguenti componenti:

- Un oggetto della classe `AppWidgetProviderInfo` [57], il quale descrive i metadati del widget, come la frequenza di aggiornamento e il layout. Viene istanziato automaticamente da Android, e i suoi attributi vengono prelevati da un file XML.
- Un oggetto di una sottoclasse di `AppWidgetProvider` [9], il quale definisce i metodi base che permettono all'utente di interfacciarsi con il widget. Viene dichiarata nel *manifest*. È compito del programmatore definire la sottoclasse e il comportamento dei suoi metodi principali (e.g. `onUpdate`).
- Layout del widget.

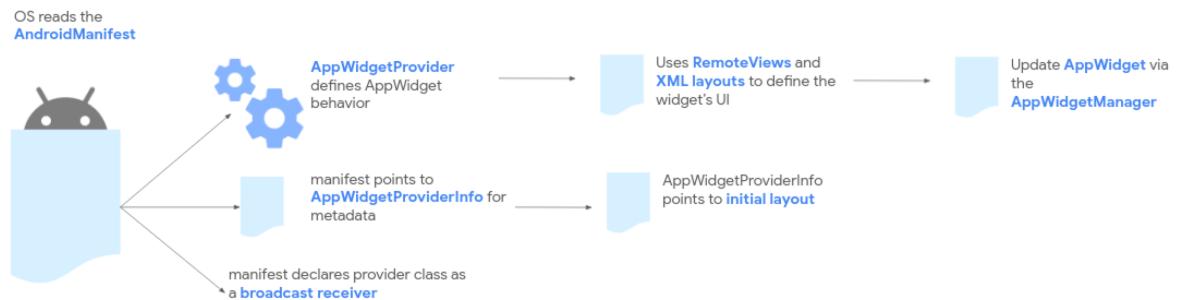


Figura 3.1: Schema del processing di un app widget. Tratto da [22].

3.1.1 Classe AppWidgetProviderInfo

L'oggetto di tipo `AppWidgetProviderInfo` definisce le caratteristiche principali del widget. È necessario definirlo in un file XML utilizzando l'elemento `<appwidget-provider>` e salvarlo nella directory `res/xml` del progetto. Un esempio di XML contenente gli attributi dell'`AppWidgetProviderInfo`:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="40dp"
    android:minHeight="40dp"
    android:targetCellWidth="1"
    android:targetCellHeight="1"
    android:maxResizeWidth="250dp"
    android:maxResizeHeight="120dp"
    android:updatePeriodMillis="86400000"
    android:description="@string/example_appwidget_description"
    android:previewLayout="@layout/example_appwidget_preview"
    android:initialLayout="@layout/example_loading_appwidget"
    android:configure="com.example.android.ExampleAppWidgetConfigurationActivity"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen"
    android:widgetFeatures="reconfigurable|configuration_optional">
</appwidget-provider>
```

Dove:

- `targetCellWidth` e `targetCellHeight` (Android 12) specificano la dimensione di default del widget in termini di *celle* della home screen. Tali attributi vengono ignorati in Android 11 nelle versioni inferiori. Inoltre, potrebbero essere ignorati se la home screen non supporta un layout basato su griglia.
- `minWidth` e `minHeight` specificano la dimensione di default del widget in dp. Se il valore minimo di larghezza o altezza non coincide con la dimensione delle celle, il valore arrotondato a quello della cella. È opportuno specificare sia gli attributi `targetCellWidth/targetCellHeight` che `minWidth/minHeight`, in modo che la tua app possa ricorrere all'uso di `minWidth` e `minHeight` nel caso in cui il dispositivo dell'utente non supporti `targetCellWidth` e `targetCellHeight`. Se supportati, gli attributi `targetCellWidth` e `targetCellHeight` hanno la precedenza sugli attributi `minWidth` e `minHeight`.
- `minResizeWidth` e `minResizeHeight` (Android 4) specificano la dimensione minima del widget. Sotto tali dimensioni, il widget diventerebbe inutilizzabile. Si sottolinea che tali attributi possono permettere all'utente di ridimensionare il widget sotto la sua dimensione di default. Per esempio, la `minResizeWidth` viene ignorata se è più grande della `minWidth`, oppure se la ridimensione orizzontale non è supportata.
- `maxResizeWidth` e `maxResizeHeight` (Android 12) specificano la dimensione massima del widget. Se i valori non sono multipli della dimensione della griglia di celle, vengono arrotondati alla più vicina dimensione di cella. In modo analogo a `minResizeWidth` e `minResizeHeight`, tali attributi vengono ignorati se più piccoli di `minWidth` e `minHeight` rispettivamente.

- **resizeMode** (Android 3.1) specifica le regole sotto le quali il widget può essere ridimensionato. Esso può essere ridimensionato solo verticalmente **vertical**, solo orizzontalmente **horizontal**, sia verticalmente che orizzontalmente **horizontal|vertical** o non essere ridimensionabile affatto **none**.
- **updatePeriodMillis** definisce il periodo con cui il framework richiede un aggiornamento dall'**AppWidgetProvider** chiamando il metodo di callback **onUpdate()**. In particolare, la invocazione non è garantita di avvenire esattamente in questo periodo. In tale contesto, sono valide le discussioni affrontate in 2.2.4.
- **initialLayout** fa riferimento alla risorsa layout che definisce il layout del widget.
- **configure** definisce l'activity da lanciare quando l'utente aggiunge il widget nella home screen, permettendolo di configurarne le proprietà. Da Android 12 è possibile definire una configurazione di default, in modo tale da saltare la procedura di configurazione.
- **description** (Android 12) specifica la descrizione mostrata dal selezionatore di widget da mostrare per il widget.
- **previewLayout** (Android 12) specifica una risorsa layout da visualizzare nel selezionatore di widget. Idealmente, esso deve essere della stessa dimensione di default del layout del widget specificato in **initialLayout** con valori realistici di default.
- **previewImage** (solo Android 11 e inferiore) specifica l'immagine di preview da visualizzare nel selezionatore di widget. Se non è fornita, l'utente visualizzerà l'icona della applicazione. Tale campo corrisponde a **android:previewImage** nel **receiver** nel **AndroidManifest.xml**. È opportuno specificare sia gli attributi **previewImage** che **previewLayout**, in modo che l'app possa ricorrere all'uso di **previewImage** nel caso in cui il dispositivo dell'utente non supporti **previewLayout**.
- **widgetCategory** dichiara se il widget può essere visualizzato nella home screen (**home_screen**), nella lock screen (**keyguard**) o entrambi. Da Android 5 in poi, solo l'attributo **home_screen** è supportato.
- **widgetFeatures** dichiara le feature, sottoforma di flag, supportate dal widget. Le due flag più utilizzate sono **configuration_optional**, per la configurazione opzionale, e **reconfigurable**, per permettere di riconfigurare il widget dopo averlo piazzato.

3.2 Classe AppWidgetProvider

La classe **AppWidgetProvider** gestisce i broadcast del widget e ne aggiorna il contenuto in risposta agli eventi del ciclo di vita.

3.2.1 Dichiarazione nel manifest

Per prima cosa è necessario dichiarare la classe **AppWidgetProvider** in *AndroidManifest.xml*. Ad esempio:

```
<receiver
    android:name="ExampleAppWidgetProvider"
    android:exported="false">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
```

```

</intent-filter>
<meta-data
    android:name="android.appwidget.provider"
    android:resource="@xml/example_appwidget_info" />
</receiver>

```

Il tag `<receiver>` è necessario poiché il provider è un `BroadcastReceiver` [17, 18]. Esso deve implementare il metodo `onReceive()` per rispondere ai broadcast di sistema (aggiornamento, posizionamento) o definiti dal programmatore. Il tag necessita dell'elemento `android:name`, il quale specifica l'`AppWidgetProvider` impiegato dal widget. Il componente non dovrebbe essere esportato. Fanno eccezione i rari casi in cui un processo differente vuole inviare broadcast al provider.

L'elemento `<intent-filter>` deve includere un elemento `<action>` con l'attributo `android:name`, in grado di specificare che l'`AppWidgetProvider` accetta i broadcast `ACTION_APPWIDGET_UPDATE`. Questo è l'unico broadcast che il programmatore deve definire esplicitamente. L'`AppWidgetManager` invia automaticamente gli altri broadcast all'`AppWidgetProvider` se necessario.

L'elemento `<meta-data>` specifica la risorsa `AppWidgetProviderInfo` e richiede i seguenti attributi:

- `android:name`: Specifica il nome dei metadati. Usare `android.appwidget.provider` per identificare l'`AppWidgetProviderInfo` come descrittore dei dati.
- `android:resource`: Specifica la posizione del `AppWidgetProviderInfo`.

3.2.2 Implementazione

La classe `AppWidgetProvider` estende `BroadcastReceiver` [17, 18]. Il suo compito è quello di ricevere le trasmissioni di evento rilevanti per il widget, come l'eliminazione o aggiornamento del widget stesso. Quando ciò accade, vengono invocati alcuni dei seguenti metodi:

- `onUpdate()`. Viene invocato per aggiornare il widget ad intervalli regolari definiti dall'attributo `updatePeriodMillis`. Tale metodo viene chiamato quanto l'utente aggiunge il widget nella home screen, di conseguenza esso deve provvederne un setup. Però, se la configuration activity viene definita senza la flag `configuration_optional`, `onUpdate()` non viene invocato quando l'utente aggiunge il widget, ma solo per gli aggiornamenti successivi.
- `onAppWidgetOptionsChanged()`. Viene invocato quando il widget viene disposto per la prima volta o quando viene ridimensionato. Da Android 12 è possibile ottenere i range delle dimensioni invocando `getAppWidgetOptions()`, il quale ritorna un `Bundle` che include:
 - `OPTION_APPWIDGET_MIN_WIDTH`: Contiene il limite inferiore in larghezza, in dp, di una istanza di un widget.
 - `OPTION_APPWIDGET_MIN_HEIGHT`: Contiene il limite inferiore in altezza, in dp, di una istanza di un widget.
 - `OPTION_APPWIDGET_MAX_WIDTH`: Contiene il limite superiore in larghezza, in dp, di una istanza di un widget.
 - `OPTION_APPWIDGET_MAX_HEIGHT`: Contiene il limite superiore in altezza, in dp, di una istanza di un widget. `OPTION_APPWIDGET_SIZES` (Android 12): Contiene la lista delle possibili dimensioni (`List<SizeF>`), in dp, di una istanza di un widget.

- `onDeleted(Context, int[])`, invocato ogni volta che il widget viene eliminato dal *widget host*.
- `onEnabled(Context)`, invocato quando viene creato per la prima volta il widget. Se l'utente aggiunge due o più widget nella home screen, questo metodo è invocato solo la prima volta. Tale metodo è adatto per aprire un nuovo database o per effettuare operazioni di setup.
- `onDisabled(Context)`, invocato quando l'ultima istanza del widget viene eliminata dal *widget host*. In tale metodo è necessario ripulire ogni operazione effettuata in `onEnabled(Context)`.
- `onReceive(Context, Intent)`, invocato per ogni trasmissione e prima di ogni metodo di callback.

3.2.3 Classe PendingIntent

Per far eseguire una azione da un elemento di un widget è necessario istanziare un `PendingIntent` [53]. Esso è una descrizione di un `Intent` e di una azione da performare con esso. Le istanze di questa classe vengono istanziate con:

- `getActivity(Context, int, Intent, int)`
- `getActivities(Context, int, Intent, int)`
- `getBroadcast(Context, int, Intent, int)`
- `getService(Context, int, Intent, int)`

L'oggetto ritornato può essere utilizzato per invocare activity, creare servizi o invocare broadcast verso altri `BroadcastReceiver` e l'`AppWidgetProvider` stesso. L'oggetto ritornato può essere dato ad un'altra applicazione cosicché possa performare l'operazione specificata. Per tale motivo è importante porre attenzione a come viene costruito il `PendingIntent`. Quasi sempre, l'`Intent` di base passato come argomento deve essere esplicito [45] per essere sicuri che la componente scelta sia l'unico destinatario.

3.2.4 Gestione degli eventi in `onUpdate()`

In `onUpdate()` è possibile definire una azione eseguita alla pressione di un pulsante tramite un `PendingIntent`. Per fare riferimento ad un elemento del widget si utilizza la classe `RemoteViews` [59], discussa in 3.3. Di seguito si reporta un esempio in cui un `Button` invoca `ExampleActivity` quando premuto, utilizzando un `PendingIntent`. L'azione viene legata alla pressione dell'oggetto con il metodo `setOnClickListener(Int, PendingIntent)` [59]. Di seguito viene riportato un esempio.

```
// Crea l'Intent per lanciare la ExampleActivity
val pendingIntent: PendingIntent = PendingIntent.getActivity(
    /* context = */ context,
    /* requestCode = */ 0,
    /* intent = */ Intent(context, ExampleActivity::class.java),
    /* flags = */ PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
)

// Ottiene il layout per il widget e assegna un on-click listener
```

```

// al pulsante
val views: RemoteViews = RemoteViews(
    context.packageName,
    R.layout.appwidget_provider_layout
).apply {
    setOnClickPendingIntent(R.id.button, pendingIntent)
}

```

3.2.5 Ricevere Intent broadcast

Le azioni di un Intent rilevanti per un widget sono:

- ACTION_APPWIDGET_UPDATE
- ACTION_APPWIDGET_DELETE
- ACTION_APPWIDGET_ENABLED
- ACTION_APPWIDGET_DISABLED
- ACTION_APPWIDGET_DISABLED
- ACTION_APPWIDGET_OPTIONS_CHANGED

La classe `AppWidgetProvider` implementa `onReceive` in modo tale da gestire `Intent` con queste azioni. È possibile supportare ulteriori azioni introducendo ulteriori `BroadcastReceiver` o facendo l'override della callback nel provider. In quest'ultimo caso è necessario garantire l'invocazione di `super.onReceive()` per preservare la risposta del provider agli `Intent` lanciati dal sistema.

3.3 Widget Layout

3.3.1 Classe `RemoteViews`

Il layout del widget è definito da un file XML nella directory `res/layout/`. I widget, a differenza di Activity e Fragment, sono basati sulle `RemoteViews` [59]. Essa è una classe che descrive una gerarchia di viste che possono essere visualizzate in un altro processo, come quello del widget. Tale classe è limitata rispetto a `View` [69]. Infatti, `RemoteViews` può, al momento della stesura, supportare i seguenti layout:

- `AdapterViewFlipper`
- `FrameLayout`
- `GridLayout`
- `LinearLayout`
- `ListView`
- `RelativeLayout`
- `StackView`
- `ViewFlipper`

e i seguenti UI widget:

- `AnalogClock`
- `Button`
- `Chronometer`
- `ImageButton`
- `ImageView`
- `ProgressBar`
- `TextClock`
- `TextView`
- `CheckBox` (Android 12)
- `RadioButton` (Android 12)
- `RadioGroup` (Android 12)
- `Switch` (Android 12)

UI widget che ereditano da tali classi non sono supportati. Nonostante la possibilità di inserire elementi come `CheckBox` o `Switch`, il widget rimane senza stato. È compito del programmatore far conservare lo stato del widget e fargli ascoltare gli eventi. Si sottolinea l'assenza dell'UI widget `EditText` nella lista. Infatti, anche il widget Google Search non permette l'inserimento dalla home screen, ma piuttosto rimanda ad una activity invocata dopo un click. Tale comportamento viene mostrato in Figura 3.2 e 3.3

3.3.2 Angoli arrotondati

Android 12 ha introdotto dei parametri di sistema per impostare il raggio degli angoli arrotondati del widget:

- `system_app_widget_background_radius` specifica il raggio dello sfondo del widget, il quale non sarà mai più largo di 28dp.
- `system_app_widget_inner_radius` specifica il raggio di ogni vista all'interno del widget. Questo è 8dp in meno di `system_app_widget_background_radius`.

Un esempio di quanto appena descritto viene mostrato in Figura 3.5.

Se il widget non utilizza `@android:id/background` il launcher cerca automaticamente di identificare il background e tagliare il widget utilizzando un rettangolo con angoli arrotondati fino a 16dp.

3.4 Mostrare collezioni

Un widget di collezioni utilizza il `RemoteViewsService` [60], un servizio [65, 66] per visualizzare collezioni che sono supportate da dati in `ContentProvider` [20] o database. Un widget può presentare i dati in uno delle seguenti *collection views*:

- `ListView` mostra i dati in una lista scorribile verticalmente (esempio in Figura 3.7).

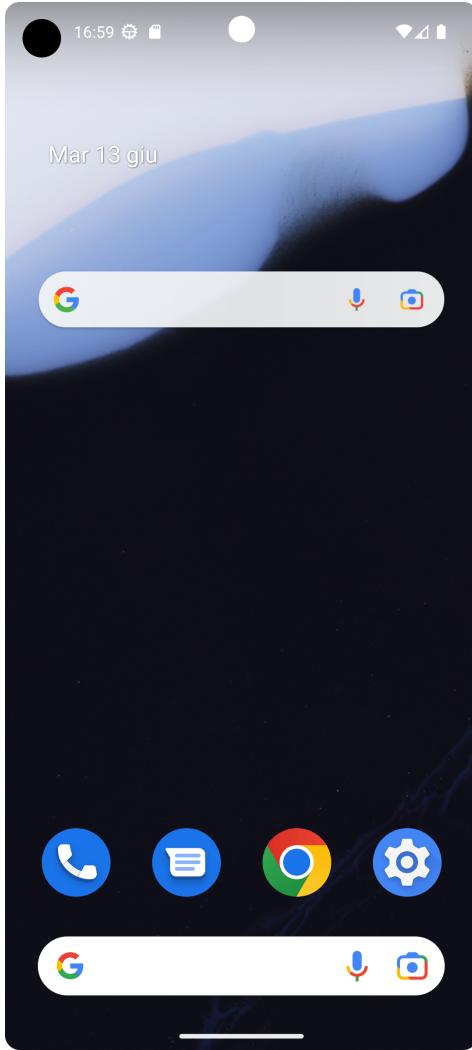


Figura 3.2: Home screen con il widget di Google Search.

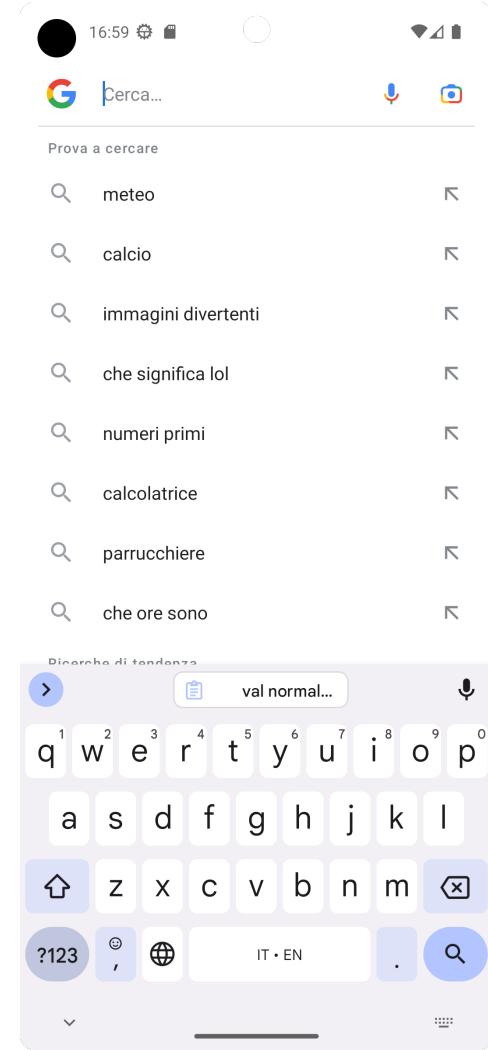


Figura 3.3: Una activity viene invocata dopo aver cliccato sul widget.

- **GridView** mostra i dati in una griglia bidimensionale scorribile verticalmente.
- **StackView** mostra i dati in una pila di carte, dove l'utente può estrarre la carta in cima alla pila per visualizzare quella sotto con uno swipe verso l'alto. Al contrario, con uno swipe verso il basso si rimette in cima la carta estratta (esempio in Figura 3.6).
- **AdapterViewFlipper** è un **ViewAnimator** in grado di creare una animazione fra due o più viste. Solo un elemento della collezione è mostrato alla volta.

All'interno di ogni **RemoteViewsService** deve essere definita una sottoclasse di **RemoteViewsFactory** [61]. Essa è un piccolo *wrapper* attorno alla interfaccia **Adapter**, ed è responsabile di creare e ritornare un elemento della collezione come oggetto di tipo **RemoteViews**. **RemoteViewsFactory** è una interfaccia per un adapter fra una vista di collezioni (e.g. **ListView**, **StackView**) e i dati di quella vista.

3.4.1 Implementazione

Nell'**AndroidManifest.xml** deve essere possibile legare i widget di collezioni con il **RemoteViewsService**. Per far ciò è necessario dichiarare il servizio nel manifest con il permesso

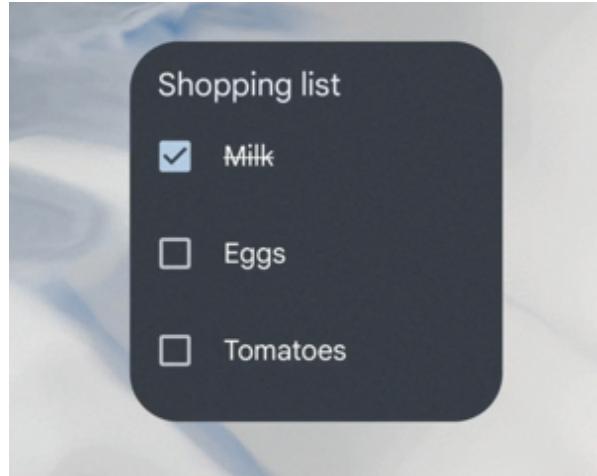


Figura 3.4: Widget che fa uso di delle CheckBox. Tratto da [22].

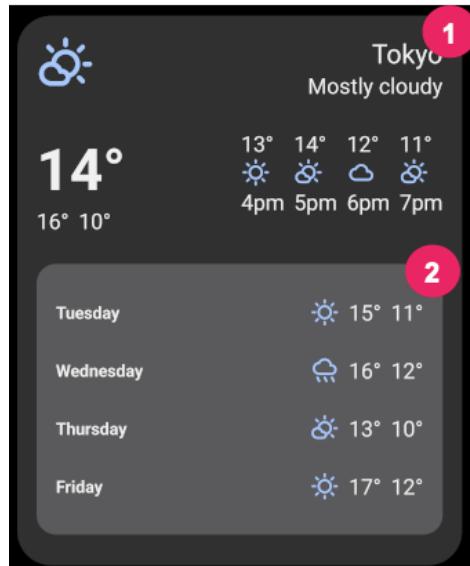


Figura 3.5: Esempio di widget con angoli arrotondati: quelli del widget (1) e quelli di una vista all'interno del widget (2).

BIND_REMOTEVIEWS. Ciò previene che altre applicazioni possano accedere liberamente ai dati nei widget. Un esempio di dichiarazione nel manifest è la seguente:

```
<service android:name="MyWidgetService"
        android:permission="android.permission.BIND_REMOTEVIEWS" />
```

dove MyWidgetService è la sottoclasse di RemoteViewsService.

3.4.2 Classe AppWidgetProvider per i widget di collezioni

Nei widget di collezioni il metodo `onUpdate()` dell'AppWidgetProvider è responsabile di invocare il metodo `setRemoteAdapter()` [59]. Questo metodo si occupa di riferire al widget il servizio che gli provvederà i dati. Il `RemoteViewsService` ritorna l'implementazione del `RemoteViewsFactory` e il widget può visualizzare i dati. A tal fine, è necessario passare un `Intent` con componente esplicita il `RemoteViewsService` e il widget ID fra gli extra. Un esempio è il seguente:



Figura 3.6: Uno StackView. Tratto da [68]

```
// Imposta l'intent che avvia l'AppWidgetProvider, che fornisce
// le viste per questa collezione.
val intent = Intent(context, StackWidgetService::class.java).apply {
    // Add the widget ID to the intent extras.
    putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId)
    data = Uri.parse(toUri(Intent.URI_INTENT_SCHEME))
}
// Istanza l'oggetto RemoteViews per il layout del widget.
val views = RemoteViews(context.packageName, R.layout.widget_layout).apply
{
    // Configura l'oggetto RemoteViews per utilizzare un adapter RemoteViews.
    // Questo adapter si collega a un RemoteViewsService tramite l'intento
    // specificato.
    // In questo modo popoli i dati.
    setRemoteAdapter(R.id.stack_view, intent)

    // La vista vuota viene visualizzata quando la collezione non ha elementi.
    // Deve essere nello stesso layout utilizzato per istanziare l'oggetto
    // RemoteViews.
    setEmptyView(R.id.stack_view, R.id.empty_view)
}
```

3.4.3 Dati persistenti

Il widget deve rimanere funzionante anche quando l'app non è attiva. Salvare i dati da visualizzare in classi il cui ciclo di vita è legato a quella dell'applicazione porta a crash e a disfunzionalità del widget. In modo analogo, non è possibile dati persistenti all'interno del `RemoteViewsService`, a meno che esso non sia statico. Tali limitazioni spingono lo sviluppatore a far uso di strumenti più affidabili e sofisticati, come un database SQL. È possibile, inoltre, utilizzare un `ContentProvider` [20] che pesca i dati da una sorgente persistente.

3.4.4 Interfaccia `RemoteViewsFactory`

I due metodi chiave nell'implementazione dell'interfaccia sono:

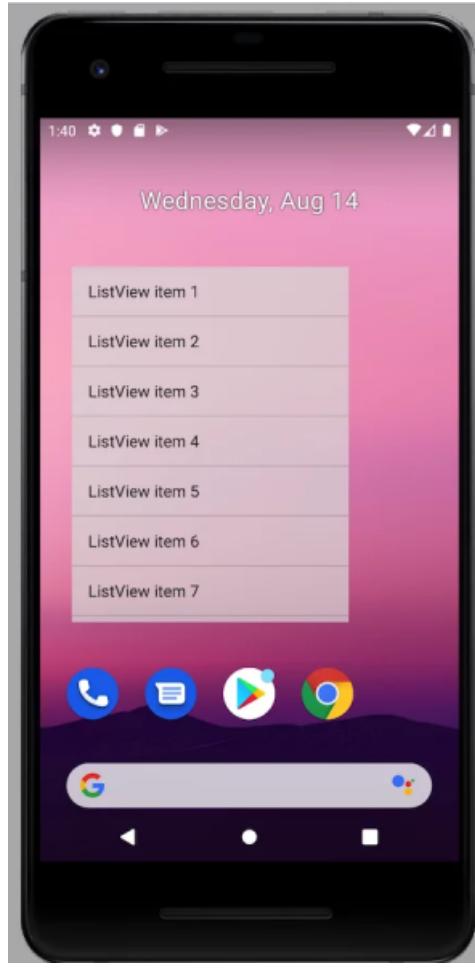


Figura 3.7: Uno `ListView`. Tratto da [21].

- `onCreate()`: Esso viene invocato quando la factory viene creata per la prima volta. In questo metodo si impostano le connessioni con la sorgente di dati (`ContentProvider` o database). Finché il widget è attivo, il sistema accede a tali oggetti utilizzando la loro posizione nella lista e ne mostra il contenuto.
- `getViewAt()`: Ritorna una `RemoteViews` dell'elemento della lista costruito. Finché il widget è attivo, il sistema accede a tali oggetti utilizzando la loro posizione nella lista e ne mostra il contenuto.

3.4.5 Aggiungere azioni di un singolo item

Come descritto in 3.2.4, per eseguire azioni alla pressione di un oggetto si utilizza `setOnClickListener()`. Però, tale approccio non è permesso per gli elementi di una collezione. Per farlo, è necessario utilizzare `setOnClickFillInIntent(Int, Intent)`. Ciò permette di impostare un *pending intent template* per la vista di collezione e poi impostare un *fill-in intent* su ogni elemento della collezione tramite il `RemoteViewsFactory`.

MyRemoteViewsFactory:

```
override fun getViewAt(position: Int): RemoteViews {  
    // Costruisce una remote views basato sull'XML del widget item
```

```

    // e imposta il testo in base alla posizione.
    return RemoteViews(context.packageName, R.layout.widget_item).apply {
        setTextViewText(R.id.widget_item, widgetItems[position].text)

        // Imposta un fill-in intent da "riempire" nel template del
        // pending intent, il quale è impostato nell'
        // AppWidgetProvider.
        val fillInIntent = Intent().apply {
            Bundle().also { extras ->
                extras.putInt(EXTRA_ITEM, position)
                putExtras(extras)
            }
        }
        // Rende possibile distinguere l'azione on-click
        // su un dato item.
        setOnClickFillInIntent(R.id.widget_item, fillInIntent)
        ...
    }
}

```

MyAppWidgetProvider:

```

override onUpdate(
    context: Context,
    appWidgetManager: AppWidgetManager,
    appWidgetIds: IntArray
)
{
    ...
    // Questa sezione permette agli elementi di avere comportamenti individualizzati.
    // Lo fa creando un modello di pending intent.
    // Gli elementi singoli di una collezione non possono creare i propri pending
    intent.
    // Al contrario, l'intera collezione crea un modello di pending intent,
    // e gli elementi singoli impostano fillInIntent per creare comportamenti
    unici
    // per ciascun elemento.
    val toastPendingIntent: PendingIntent = Intent(
        context,
        StackWidgetProvider::class.java
    ).run {
        // Imposta l'azione per l'intent.
        // Quando l'utente tocca una vista specifica, ha l'effetto di
        // inviare l'azione TOAST_ACTION.
        action = TOAST_ACTION
        putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId)
        data = Uri.parse(toUri(Intent.URI_INTENT_SCHEME))

        PendingIntent.getBroadcast(context, 0, this, PendingIntent.FLAG_UPDATE_CURRENT)
    }
}

```

```

        rv.setPendingIntentTemplate(R.id.stack_view, toastPendingIntent)
        ...
    }

    override fun onReceive(context: Context, intent: Intent) {
        val mgr: AppWidgetManager = AppWidgetManager.getInstance(context)
        if (intent.action == TOAST_ACTION) {
            val appWidgetId: Int = intent.getIntExtra(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID
            )
            // EXTRA_ITEM rappresenta un valore personalizzato fornito dall'Intent
            // passato al metodo setOnClickFillInIntent() per indicare la posizione
            // dell'elemento cliccato.
            val viewIndex: Int = intent.getIntExtra(EXTRA_ITEM, 0)
            Toast.makeText(context, "Touched view $viewIndex", Toast.LENGTH_SHORT).show()
        }
        super.onReceive(context, intent)
    }
}

```

3.4.6 Step di un RemoteViewsFactory

In Figura 3.8 viene mostrato l'ordine e il modo in cui i metodi di un `RemoteViewsFactory` vengono invocati per mostrare a schermo le viste.

Sfruttando tale interazione è possibile mantenere i dati sulle viste aggiornati. A tal fine, è necessario invocare il metodo di `notifyAppWidgetViewDataChanged()` dell'`AppWidgetManager`. Questo invoca la callback `onDataSetChanged` del `RemoteViewsFactory`, permettendo l'acquisizione di nuovi dati.

3.4.7 RemoteCollectionItems

Da Android 12 in poi è possibile passare una collezione direttamente quando si popola una collection view. Ciò è reso possibile da `setRemoteAdapter(int, RemoteViews.RemoteCollectionItems)`. Impostando l'adapter in tal modo, non è necessario implementare il `RemoteViewsFactory` o invocare `notifyAppWidgetViewDataChanged`. Inoltre, tale approccio rimuove la latenza dovuta alla comparsa di nuovi item della collezione durante lo scorrimento. Però, tale meccanismo porta benefici quando il numero di oggetti nella collezione è piccolo.

3.5 Widget avanzati

Esistono pratiche, descritte nella documentazione [24], per rendere l'esperienza dell'utente con il widget migliore.

3.5.1 Ottimizzazioni nell'aggiornamento

Come discusso in 2.2.4, l'aggiornamento di un widget è costoso in termini di batteria e potenza di calcolo. Per tale motivo esistono diversi tipologie di update:

- *Full update* tramite `AppWidgetManager.updateAppWidget(int, RemoteViews)`. In tal modo il widget viene aggiornato completamente. La chiamata sostituisce le Remo-

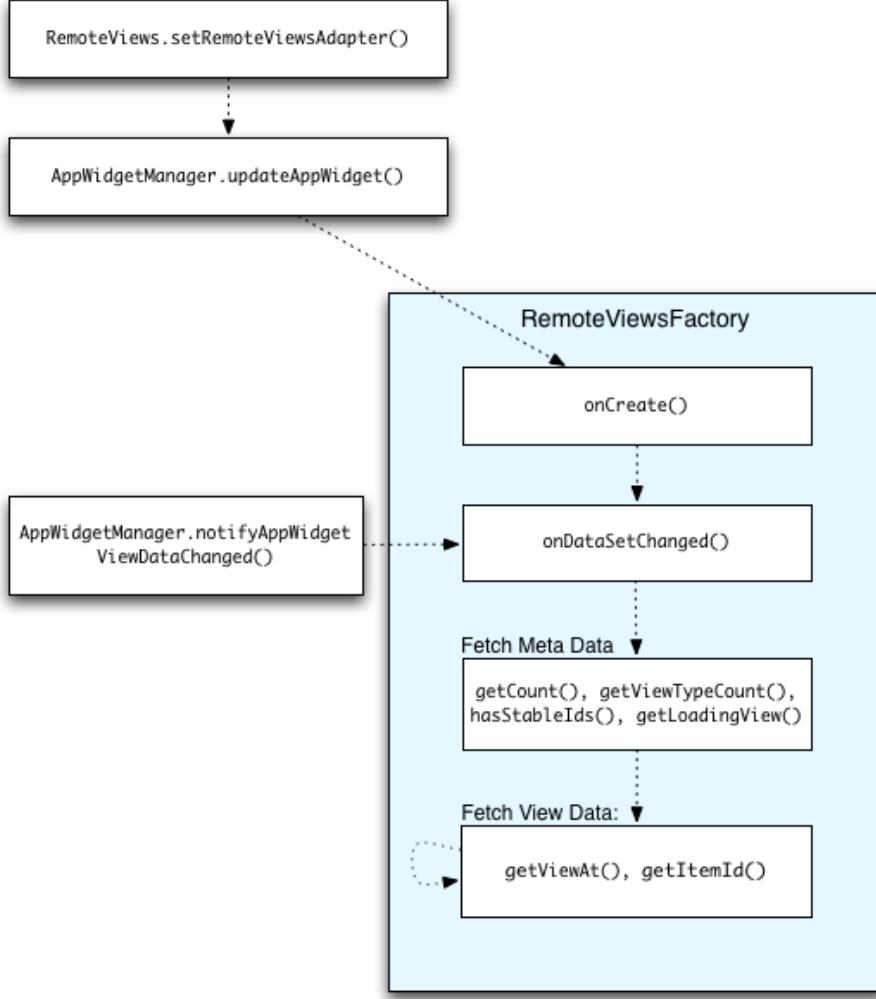


Figura 3.8: Step di un `RemoteViewsFactory` durante un aggiornamento. Tratto da [68].

`teViews` precedenti con delle nuove `RemoteViews`. Per ovvie ragioni, questo tipo di aggiornamento è il più costoso.

- *Partial update* tramite `AppWidgetManager.partiallyUpdateAppWidget(int, RemoteViews)` per aggiornare solo una parte del widget. In particolare, tale metodo fa un merge delle `RemoteViews` provviste come argomento e delle `RemoteViews` precedentemente impostate. Questo metodo viene ignorato se il widget non riceve almeno un full update attraverso `updateAppWidget(int[], RemoteViews)`.
- *Collection data refresh* tramite `AppWidgetManager.notifyAppWidgetViewDataChanged` per invalidare la collezione di dati nel widget. Tale metodo attiva l'invocazione di `RemoteViewsFactory.onDataSetChanged`. Nel frattempo che la collezione viene aggiornata, i dati vecchi vengono mostrati nel widget. In questo metodo è possibile eseguire operazioni costose in modo sincrono.

Inoltre è possibile aggiornare il widget in base all'interazione con l'utente da una activity dell'applicazione (invocando una full update a seguito, per esempio, di un click di un pulsante) o da una interazione remota, come una notifica o un app widget. In quest'ultimo caso è

possibile costruire un `PendingIntent` e aggiornare il widget dalla componente invocata, come una `Activity`, un `BroadcastReceiver` o un `Service`.

3.5.2 Aggiornamento del widget in risposta ad un evento broadcast

È possibile programmare un *job* attraverso un `JobScheduler` [48], specificando il broadcast da emettere con il metodo `JobInfo.Builder.addTriggerContentUri`.

È possibile registrare un `BroadcastReceiver`, ma questo si rivela dispendioso di risorse [24]. Di conseguenza, tale approccio va utilizzato con attenzione e facendo ascoltare il receiver solo ad alcuni broadcast. Nel caso si decida di aggiornare il widget in questo modo, è necessario tenere a mente che il sistema concede al receiver 10 secondi prima di sollevare un errore *Application Not Responding* (ANR) [7]. Se l'aggiornamento sfiora questo intervallo di tempo, è opportuno considerare le seguenti alternative:

- Programmare un task usando la classe `WorkManager` [74].
- Permettere al receiver di venir eseguito per un massimo di 30 secondi con `goAsync` [18].

3.6 Miglioramenti applicabili ad un widget

Vi sono dei miglioramenti (illustrati in [32]) che rendono il widget più piacevole all'utilizzo e, di fatto, lo completano a 360°. In primo luogo è possibile, da Android 12, utilizzare i colori dinamici (vedi la Sezione 2.4.1). Per le versioni precedenti alla 12, si può creare un tema di default e inserirlo nella cartella `values-v31`.



Figura 3.9: Widget *light theme* con Dynamic Colors. Tratto da [32].



Figura 3.10: Widget *dark theme* con Dynamic Colors. Tratto da [32].

In secondo luogo, è implementabile l'attivazione del *supporto vocale*. Tale meccanismo, con il supporto di un assistente vocale, permette di richiamare l'applicazione, la quale utilizzerà i widget per mostrare le informazioni richieste. Il supporto vocale per il proprio widget è realizzabile grazie ai Built-In Intents (*BII*s) [8]. Inoltre è possibile *pinnare* i widget nella schermata dell'assistente vocale, permettendo l'utente di accedervi più facilmente.

3.6.1 Widget picker

Da Android 12 in poi, è possibile aggiungere preview dinamiche e descrizioni per il widget che li necessita. Nonostante tale operazione sia facoltativa da parte dello sviluppatore, in generale, è buona prassi farlo per ogni widget. In particolare, da Android 12 la preview visibile nel widget picker è **scalabile**; se precedentemente la preview era una risorsa *drawable* statica (che talvolta non rappresentava accuratamente come il widget sarebbe stato al suo utilizzo nella home screen), ora invece è possibile disporre di una dinamica, mediante l'utilizzo dell'attributo `previewLayout` in `appwidget-provider`, al cui interno inseriremo il layout che vogliamo mostrare come preview (è consigliato utilizzare lo stesso layout utilizzato per il widget). Per mostrare una preview accurata, è possibile definire dei valori fittizi per le view desiderate da utilizzare nella preview stessa (altrimenti il risultato potrebbe essere impreciso):

- Per le `TextView`, si utilizza `android:text="@string/my_widget_item_fake_1"`
- Invece, per le `ImageView`, si usa `android:src="@drawable/my_widget_icon"`
- Per quanto riguarda invece elementi più complessi quali `ListView`, `GridView`, ecc., è consigliabile utilizzare un layout separato che contiene il widget effettivo e una collezione con valori fittizi. In questo caso, in `appwidget-provider` è necessario specificare come `previewLayout` il nuovo layout creato ad hoc, mentre come `initialLayout` usiamo il layout del widget [2].

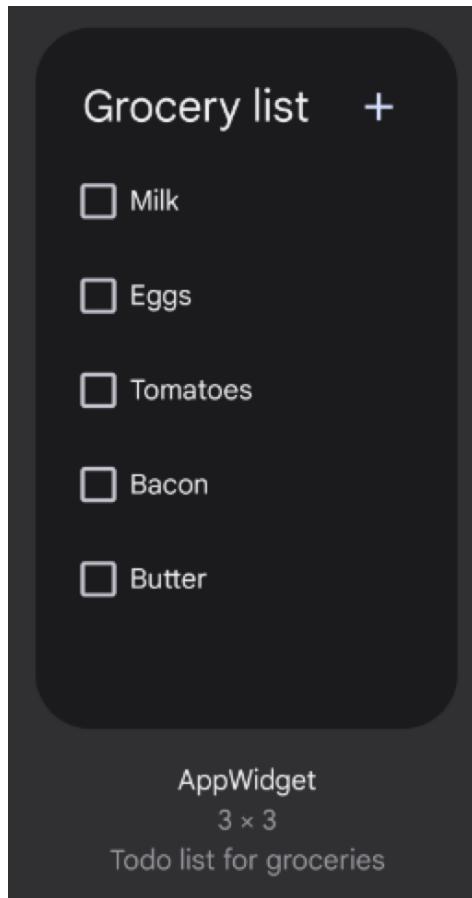


Figura 3.11: Esempio di una preview di un widget. Tratto da [32].

Retrocompatibilità per le preview scalabili

Per versioni precedenti ad Android 12, si deve specificare l'attributo `previewImage` (il quale ha una precedenza minore rispetto a `previewLayout` nelle nuove versioni). Lo svantaggio di questo approccio consiste nel dover riaggiornare la `previewImage` in caso di modifica del widget.

3.6.2 Aggiungi una descrizione al widget

Da Android 12, è possibile aggiungere una descrizione del widget nel widget picker. Per fare ciò, si utilizza l'attributo `description` in `appwidget-provider`. È consigliato essere concisi nella descrizione, in quanto descrizioni troppo lunghe potrebbero causare problemi a seconda del dispositivo utilizzato. Per le versioni Android 11 e inferiori esiste la possibilità di usare `descriptionRes` per inserire una descrizione, ma quest'ultima verrà ignorata dal widget picker.

3.6.3 Transizioni più fluide

Sempre grazie ad Android 12, è possibile usare transizioni più *smooth* all'apertura dell'app dal widget. Per attivarle è necessario usare, come elemento di background del layout, `android:id/background` oppure `android.R.id.background`. Google consiglia di evitare l'utilizzo di *broadcast trampolines* [15], in quanto le animazioni non li supportano e la user

experience sarebbe pessima. Come prima, è possibile utilizzare `@android:id/background` per versioni precedenti alla 12, ma verrà ignorato.

3.6.4 Modifiche a runtime di RemoteViews

Sempre a partire da Android 12, è possibile utilizzare vari metodi di `RemoteViews` che permettono la modifica a runtime degli attributi di `RemoteViews` stessa. Ad esempio:

```
// Set the colors of a progress bar at runtime.  
remoteView.setColorStateList(  
    R.id.progress,  
    "setProgressTintList",  
    createProgressColorStateList()  
)  
  
// Specify exact sizes for margins.  
remoteView.setViewLayoutMargin(  
    R.id.text,  
    RemoteViews.MARGIN_END,  
    8f,  
    TypedValue.COMPLEX_UNIT_DP  
)
```

3.7 Layout flessibili

Tra le novità di Android 12 vi sono quelle relative al dimensionamento dei layout dei widget [25]. È stata introdotta la possibilità di:

- Specificare ulteriori vincoli nel dimensionamento, già discussi nel dettaglio in 3.1.1;
- Realizzare *layout responsivi* o *layout esatti*

Nelle precedenti versioni di Android era possibile ottenere i possibili range delle dimensioni dagli extra `OPTION_APPWIDGET_MIN_WIDTH`, `OPTION_APPWIDGET_MIN_HEIGHT`, `OPTION_APPWIDGET_MAX_WIDTH` e `OPTION_APPWIDGET_MAX_HEIGHT`. Si doveva poi stimare la dimensione del widget. Però, tale strategia non era implementabile in tutti i casi.

3.7.1 Layout responsivi

Quando il layout deve cambiare a seconda della dimensione del widget, è possibile creare un piccolo insieme di layout, ognuno valido in un range di dimensioni. Ciò permette di rendere il widget scalabile e avere una minore impronta sul consumo di batteria. Infatti, in questo caso il sistema non deve richiamare l'applicazione ogni volta che deve mostrare il widget con una dimensione diversa.

Per implementare un layout responsivo è necessario creare un *view mapping*, una mappa `Map<SizeF, RemoteViews>`, che associa ad un range di dimensioni il layout del widget da impostare e aggiornare. Di seguito si propone l'esempio presente nella documentazione. Si consideri il seguente mapping:

```
override fun onUpdate(...)  
{  
    val smallView = ...
```

```

    val tallView = ...
    val wideView = ...

    val viewMapping: Map<SizeF, RemoteViews> = mapOf(
        SizeF(150f, 100f) to smallView,
        SizeF(150f, 200f) to tallView,
        SizeF(215f, 100f) to wideView
    )
    val remoteViews = RemoteViews(viewMapping)

    appWidgetManager.updateAppWidget(id, remoteViews)
}

```

Si assumano i seguenti valori nel `AppWidgetProviderInfo`:

```

<appwidget-provider
    android:minResizeWidth="160dp"
    android:minResizeHeight="110dp"
    android:maxResizeWidth="250dp"
    android:maxResizeHeight="200dp">
</appwidget-provider>

```

Le informazioni nel view mapping e nel `AppWidgetProviderInfo` dicono che:

- `smallView` supporta da 160dp (`minResizeWidth`) × 110dp (`minResizeHeight`) a 160dp × 199 dp (cioè il valore di soglia - 1dp).
- `tallView` supporta da 160dp × 200dp a 214dp (cioè il valore di soglia - 1dp) × 200dp.
- `wideView` supporta da 215dp × 110dp (`minResizeHeight`) a 250dp (`maxResizeWidth`) × 200dp (`maxResizeHeight`).

In particolare, il widget deve supportare i range di dimensioni fra `minResizeWidth` × `minResizeWidth` a `maxResizeWidth` × `maxResizeWidth`. All'interno del range, è possibile definire i valori di soglia delle diverse viste. Degli esempi di responsive layout sono riportati in Figura 3.12, 3.13, 3.14, 3.15 e 3.16, tutte tratte dalla documentazione [25].



Figura 3.12: Widget con layout responsivo 3x2. Tratto da [25].



Figura 3.13: Widget con layout responsivo 4x2. Tratto da [25].

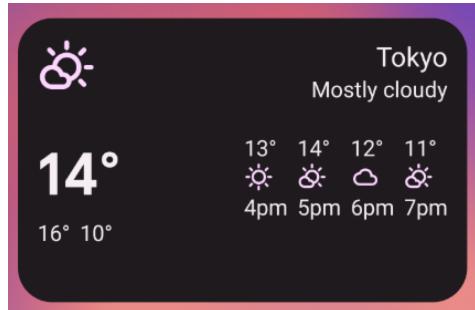


Figura 3.14: Widget con layout responsivo 5x2. Tratto da [25].



Figura 3.15: Widget con layout responsivo 5x3. Tratto da [25].



Figura 3.16: Widget con layout responsivo 5x4. Tratto da [25].

3.7.2 Layout esatti

Quando il numero di layout da visualizzare è ben superiore a 3 o 4, è opportuno realizzare diversi layout ognuno dedicato ad una dimensione del widget. A tale scopo si possono seguire i seguenti passi:

1. Eseguire un overload di `AppWidgetProvider.onAppWidgetOptionsChanged()`, callback invocata quando le dimensioni del widget cambiano.
2. Invocare `AppWidgetManager.getAppWidgetOptions()`, il quale ritorna un `Bundle` contenente le dimensioni.
3. Accedere a `AppWidgetManager.OPTION_APPWIDGET_SIZES` dal `Bundle`.

Di seguito si riporta un esempio di quanto appena descritto:

```
override fun onAppWidgetOptionsChanged(
    context: Context,
    appWidgetManager: AppWidgetManager,
    id: Int,
    newOptions: Bundle?
) {
    super.onAppWidgetOptionsChanged(context, appWidgetManager, id, newOptions)
    // Get the new sizes.
    val sizes = newOptions?.getParcelableArrayList<SizeF>(
        AppWidgetManager.OPTION_APPWIDGET_SIZES
    )
    // Controlla che la lista di dimensioni sia stato inserito dal launcher
    if (sizes.isNullOrEmpty()) {
        return
    }
    // Associazione delle dimensioni alle RemoteViews scelte dal programmatore
```

```

    val remoteViews = RemoteViews(sizes.associateWith(::createRemoteViews))
    appWidgetManager.updateAppWidget(id, remoteViews)
}

// Crea la RemoteViews per una data dimensione
private fun createRemoteViews(size: SizeF): RemoteViews { }

```

3.8 Configurazione del widget da parte dell'utente

Android permette all'utente di configurare il proprio widget al momento dell'inserimento nella schermata home o dopo averlo posizionato. A tale scopo, il programmatore deve implementare una activity di configurazione.

3.8.1 Dichiarazione della activity di configurazione

L'activity di configurazione è una activity a tutti gli effetti, dunque essa va dichiarata assieme alle altre componenti nel manifest. Essa viene lanciata con l'azione ACTION_APPWIDGET_CONFIGURE, di conseguenza essa deve poter accettare questi tipi di intent attraverso un intent filter [46]. Un esempio di dichiarazione di activity di configurazione è la seguente:

```

<activity android:name=".ExampleAppWidgetConfigurationActivity">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_CONFIGURE"/>
    </intent-filter>
</activity>

```

Per invocare azionare il lancio dell'activity di configurazione si deve aggiungere al AppWidgetProviderInfo.xml:

```

<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    android:configure="com.example.android.ExampleAppWidgetConfigurationActivity"
    ... >
</appwidget-provider>

```

3.8.2 Implementazione della activity

L'activity di configurazione deve ritornare un risultato. Esso deve sempre contenere il widget ID passato dall'intent che ha invocato l'activity, passato come extra come EXTRA_APPWIDGET_ID. Inoltre, è importante considerare che il sistema non invia un broadcast ACTION_APPWIDGET_UPDATE quando la activity di configurazione viene lanciata: di conseguenza, il metodo onUpdate() non viene invocato alla creazione del widget. Infatti, è responsabilità della activity di configurazione richiedere un update dall'AppWidgetManager quando il widget viene creato per la prima volta. Una linea guida per la realizzazione di una activity di configurazione è la seguente:

1. Reperire il widget ID dall'intent che ha lanciato l'activity:

```

val appWidgetId = intent?.extras?.getInt(
    AppWidgetManager.EXTRA_APPWIDGET_ID,
    AppWidgetManager.INVALID_APPWIDGET_ID
) ?: AppWidgetManager.INVALID_APPWIDGET_ID

```

2. Impostare il risultato dell'activity in `RESULT_CANCELED`: in questo modo, se l'utente torna indietro dalla activity di configurazione, l'OS reperisce l'esito e non aggiunge il widget sullo schermo.

```
val resultValue = Intent().putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
appWidgetId)
 setResult(Activity.RESULT_CANCELED, resultValue)
```

3. Effettiva configurazione del widget a seconda delle preferenze dell'utente.
4. Una volta completa la configurazione, si ottiene una istanza di `AppWidgetManager`:

```
val appWidgetManager = AppWidgetManager.getInstance(context)
```

5. Aggiornare il widget con una `RemoteViews` invocando:

```
val views = RemoteViews(context.packageName, R.layout.example_appwidget)
appWidgetManager.updateAppWidget(appWidgetId, views)
```

6. Creare un intent di ritorno, impostare l'esito positivo e terminare l'activity di configurazione.

```
val resultValue = Intent().putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
appWidgetId)
setResult(Activity.RESULT_OK, resultValue)
finish()
```

3.8.3 Riconfigurazione di un widget

È possibile permettere all'utente di riconfigurare un widget già piazzato nello schermo. Tale feature fu introdotta in Android 9, ma venne supportata a pieno da Android 12 in poi. Per far configurare un widget all'utente è necessario specificare la flag `reconfigurable` in `widgetFeatures` dell'`AppWidgetProviderInfo.xml`.

```
<appwidget-provider
    android:configure="com.myapp.ExampleAppWidgetConfigurationActivity"
    android:widgetFeatures="reconfigurable">
</appwidget-provider>
```

L'utente può riconfigurare il widget tenendo premuto su di esso e selezionando il pulsante di riconfigurazione in basso a destra. La Figura 3.17 mostra un widget che permette la sua riconfigurazione.

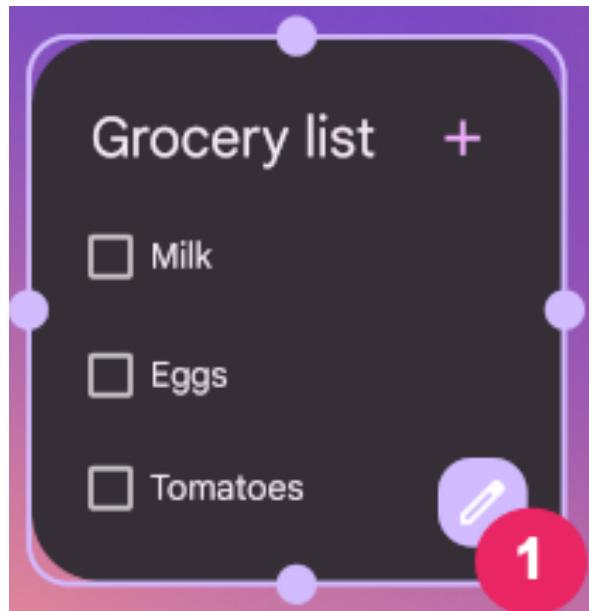


Figura 3.17: Widget con pulsante di riconfigurazione in basso a destra. Tratto da [31].

3.8.4 Pin di un widget

Da Android 8 in poi, è possibile creare dei *pinned widget*. Questi danno accesso a specifici task e possono essere aggiunti all'home screen direttamente dall'app. Per far ciò, è possibile utilizzare il metodo `requestPinAppWidget()`. Un esempio è riportato di seguito:

```
val appWidgetManager = AppWidgetManager.getInstance(context)
val myProvider = ComponentName(context, ExampleAppWidgetProvider::class.java)

if (appWidgetManager.isRequestPinAppWidgetSupported())
{
    // Crea un PendingIntent solo se l'app ha bisogno di essere
    // notificata quando l'utente decide di pinnare il widget.
    // Se l'operazione di pin fallisce, l'app non viene
    // notificata.
    val successCallback = PendingIntent.getBroadcast(
        /* context = */ context,
        /* requestCode = */ 0,
        /* intent = */ Intent(...),
        /* flags = */ PendingIntent.FLAG_UPDATE_CURRENT)

    appWidgetManager.requestPinAppWidget(myProvider, null, successCallback)
}
```

3.9 Widget host

Android è altamente modulare e permette di modificare e sostituire componenti chiave dell'OS, in modo analogo alle diverse distribuzioni GNU/Linux e a differenza dei proprietari iOS, macOS e Windows. Nel caso in cui si decida di sviluppare una componente sostitutiva alla schermata home, è possibile integrare i widget creati attraverso un `AppWidgetHost` [11]. Nell'implementazione di un `AppWidgetHost` giocano un ruolo chiave le seguenti componenti:

- **AppWidgetHost**. Esso provvede all’interazione del widget service per le app (e.g. la home screen) che vogliono includere un widget nella loro UI. Esso deve avere un ID unico fra tutti gli host, tipicamente hard-coded.
- App widget ID. Come trattato precedentemente, ogni widget possiede un ID univoco. Esso viene assegnato al momento del *binding*, ottenuto dall’host tramite `allocateAppWidgetId()`.
- **AppWidgetHostView** [12], una superficie in cui il widget viene racchiuso quando deve essere *inflated* dall’host.
- Bundle delle opzioni. L’**AppWidgetHost** utilizza tale bundle per comunicare informazioni all’**AppWidgetProvider** come il widget deve essere visualizzato.

3.9.1 Binding dei widget

Quando un utente aggiunge un widget ad un host avviene una procedura chiamata *binding*. Esso consiste nell’associazione di un particolare widget ID ad uno specifico host e **AppWidgetProvider**. Per far ciò, è necessario dichiarare nel manifest il seguente permesso:

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

Inoltre, l’utente deve esplicitamente garantire il permesso a runtime. Per mostrare il dialog all’utente di richiesta di permesso si può utilizzare il seguente codice:

```
val intent = Intent(AppWidgetManager.ACTION_APPWIDGET_BIND).apply {
    putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId)
    putExtra(AppWidgetManager.EXTRA_APPWIDGET_PROVIDER, info.componentName)
    // This is the options bundle described in the preceding section.
    putExtra(AppWidgetManager.EXTRA_APPWIDGET_OPTIONS, options)
}
startActivityForResult(intent, REQUEST_BIND_APPWIDGET)
```

3.9.2 Responsabilità dell’host

In tutte le versioni di Android, l’**AppWidgetHost** ha le seguenti responsabilità:

- È necessario allocare il widget ID come specificato sopra. Inoltre, ci si deve assicurare che alla rimozione del widget si invochi `deleteAppWidgetId()` per deallocare il widget ID.
- Al piazzamento del widget, l’host deve controllare se deve essere invocata l’activity di configurazione o meno. Inoltre, deve gestire i casi in cui la suddetta activity non ritorni o finisca con un errore.
- Si deve assicurare che il widget sia piazzato con una dimensione in numero di dp specificata nell’**AppWidgetProviderInfo**. Deve essere anche gestito il caso in cui l’applicazione contenga una griglia, facendo riferimento al numero di celle che soddisfa `minWidth` e `minHeight`.

Capitolo 4

Cypher

Nell'era digitale in cui viviamo, la sicurezza delle password è diventata un elemento cruciale per proteggere la nostra identità e i nostri dati personali. Con la crescita del numero di account online che utilizziamo ogni giorno, diventa sempre più difficile tenere traccia di tutte le nostre password in modo sicuro e conveniente.

Per venire incontro alle esigenze degli utenti, oltre a dimostrare l'utilizzo dei componenti chiave dei widget, abbiamo sviluppato un'applicazione che ne fa uso: *Cypher*.

Con *Cypher*, gli utenti possono creare un database locale per memorizzare le credenziali di accesso ai propri servizi. Ogni credenziale è composta da un username, il servizio o sito web associato e la password corrispondente. In questo modo, gli utenti possono organizzare e accedere facilmente alle loro credenziali senza doverle ricordare o annotare manualmente. Per rendere la conservazione delle password sicure, esse sono crittografate all'interno del database. Per visualizzarle, è necessario inserire una *master password*. In modo simile ai servizi web, tale password non è salvata in chiaro nell'applicazione, ma viene hashata, così da rendere quasi impossibile il furto della chiave di crittografia.

Nello sviluppare tale applicazione abbiamo fatto particolarmente attenzione all'aspetto estetico dell'interfaccia e all'utilizzo delle linee guida chiave relative ai widget. In particolare, oltre al normale utilizzo dell'app, sono disponibili tre diversi home widget ridimensionabili in grado di offrire soluzioni differenti all'utente finale.



Figura 4.1: Logo di *Cypher*

4.1 Struttura dell'app

4.1.1 Primo accesso

Al primo accesso dell'app, viene mostrata una schermata (Figura 4.2) in cui è possibile configurare per la prima volta la propria master password, chiave di crittografia di ogni password nel database. È possibile cambiarla in futuro, tramite la schermata di Impostazioni 4.1.5. Inoltre, *Cypher* rimanda alla pagina di Google dove vengono consigliate le migliori pratiche per creare una password forte [23].

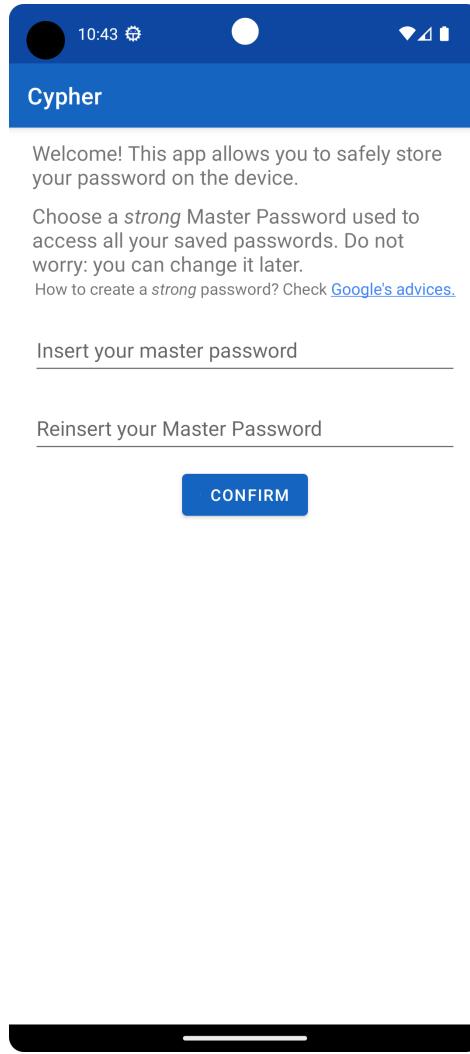


Figura 4.2: Schermata di primo accesso all'app.

4.1.2 Schermata principale

La schermata principale fornisce 3 elementi principali:

- Una lista in cui vengono mostrare le credenziali salvate (e la presenza di vari pulsanti che svolgono varie funzioni);
- Un pulsante per entrare nella sezione delle statistiche (4.1.4) dell'app, posto nella barra di navigazione in basso;

- Un pulsante che fa entrare l'utente nelle impostazioni (4.1.5) posto in alto a destra.

È possibile tornare nella schermata principale cliccando sul pulsante *Credentials* sulla barra di navigazione inferiore. Per quanto riguarda la lista di credenziali, vi sono 2 parti principali: la parte sinistra fa vedere nome utente, nome del servizio e password (bloccata o sbloccata, in quest'ultimo caso vengono mostrati degli asterischi), mentre la parte a destra si compone di 4 pulsanti. Essi hanno le seguenti funzioni (a partire dal primo in alto a sinistra):

- Il primo consiste in un tasto **modifica**, che permette di modificare le credenziale;
- Il secondo è invece un tasto **copia**, utilizzato per copiare la password nella clipboard e poi incollarla dove necessario.;
- Il terzo è il tasto **blocca** o **sblocca**. Questo pulsante permette di bloccare o sbloccare la password. Una volta sbloccata la password, il pulsante cambia comportamento, permettendo di ribloccarla. Viceversa, se la password è bloccata, cliccare sul pulsante permette di sbloccarla;
- L'ultimo è invece il tasto **elimina**, che serve a eliminare la credenziale dal database.

I quattro pulsanti vengono riposizionati diversamente nel caso in cui il dispositivo sia disposto orizzontalmente (Figura 4.3). Ovviamente, per compiere una qualsiasi di queste operazioni è necessario inserire la master password, per garantire all'utente una sicurezza maggiore riguardo ad eventuali utilizzatori terzi che vogliono "sbirciare" le credenziali dell'utente primo. Unica eccezione a tale regola è la possibilità di poter oscurare una credenziale senza inserire la master password.

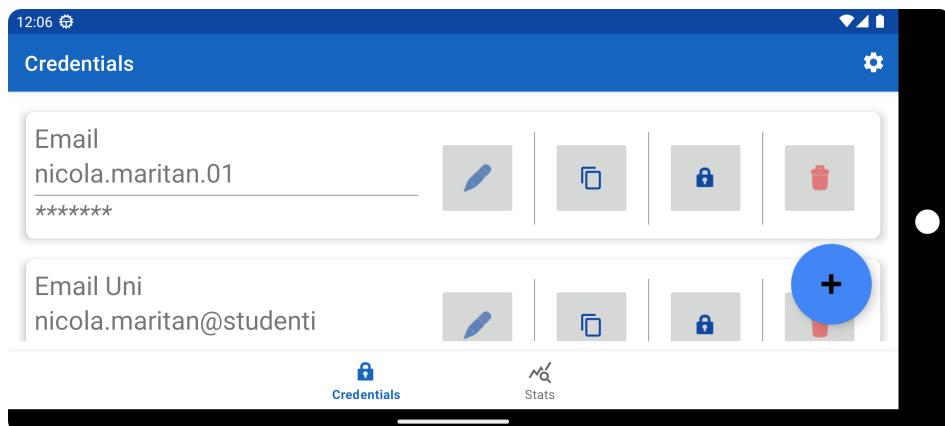


Figura 4.3: Schermata "Credentials" in versione landscape.

4.1.3 Aggiunta di credenziali

Per aggiungere un nuovo set di credentiali è molto semplice:

- Per prima cosa, posizionarsi nella schermata delle *Credentials* e cliccare sul tasto "+", un **Floating Action Button** [3] (FAB) (Figura 4.6);
- Il FAB (Figura 4.7) aprirà un **Dialog** che permette di aggiungere i dati desiderati;
- È necessario inserire servizio, nome utente, password e anche la master password, per permettere soltanto all'utente proprietario di quest'ultima di agire sull'inserimento (Figura 4.8);

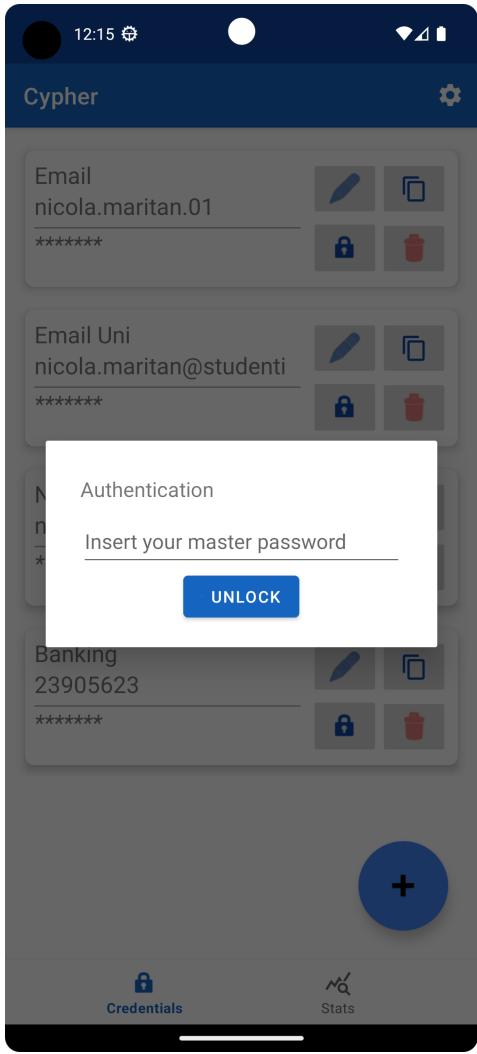


Figura 4.4: L'utente clicca sul lucchetto della credenziale che vuole sbloccare e inserisce la Master Password.

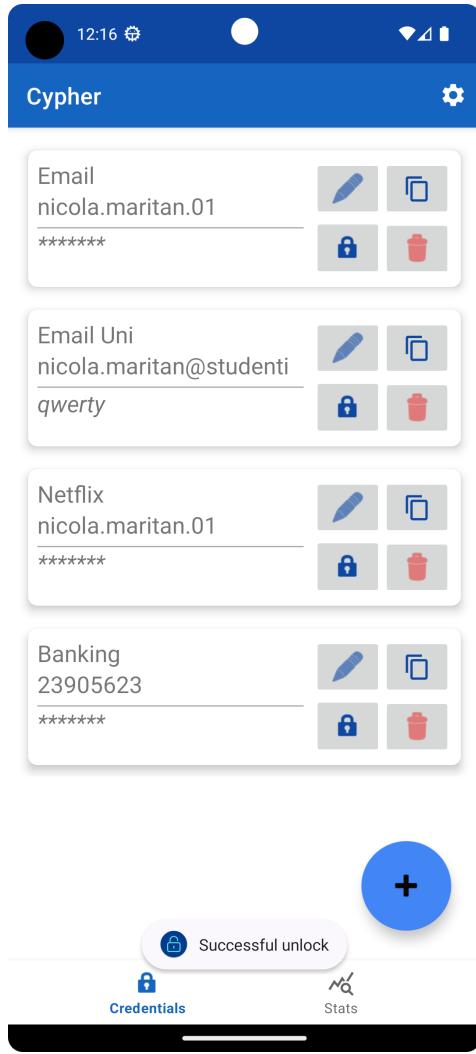


Figura 4.5: La credenziale viene sbloccata e la password è visibile. Ri cliccando sul lucchetto permette di bloccare nuovamente la credenziale.

- Fatto! (Figura 4.9) Ora le nuove credenziali compariranno nella lista di tutte le credenziali, nell'ordine stabilito nelle Impostazioni(4.1.5).

4.1.4 Statistiche

Le statistiche sono utilizzate per dare informazioni utili all'utente. Un'immagine di tale schermata è riportata in Figura 4.10 In primo luogo gli viene mostrato il numero di password salvate nell'app e di password inserite o modificate più di sei mesi fa; successivamente, viene mostrata una lista (scandita da una `RecyclerView`) di password che è consigliato cambiare: in questo caso si è deciso di dare priorità media alle password vecchie di 3 mesi e priorità alta a quelle più vecchie di 6 mesi, in modo tale da spingere l'utente a cambiarle per una sua maggiore sicurezza (questa feature è anche disponibile sottoforma di widget, vedi la Sezione 4.2 di questo Capitolo). Le credenziali sono ordinate in base alla priorità, scandita dalla data di inserimento delle stesse nell'app (per ulteriori informazioni sul tipo di dato utilizzato per la data, consulta la Sezione 4.1.3) del Capitolo corrente). Oltre all'ordine di comparsa, le

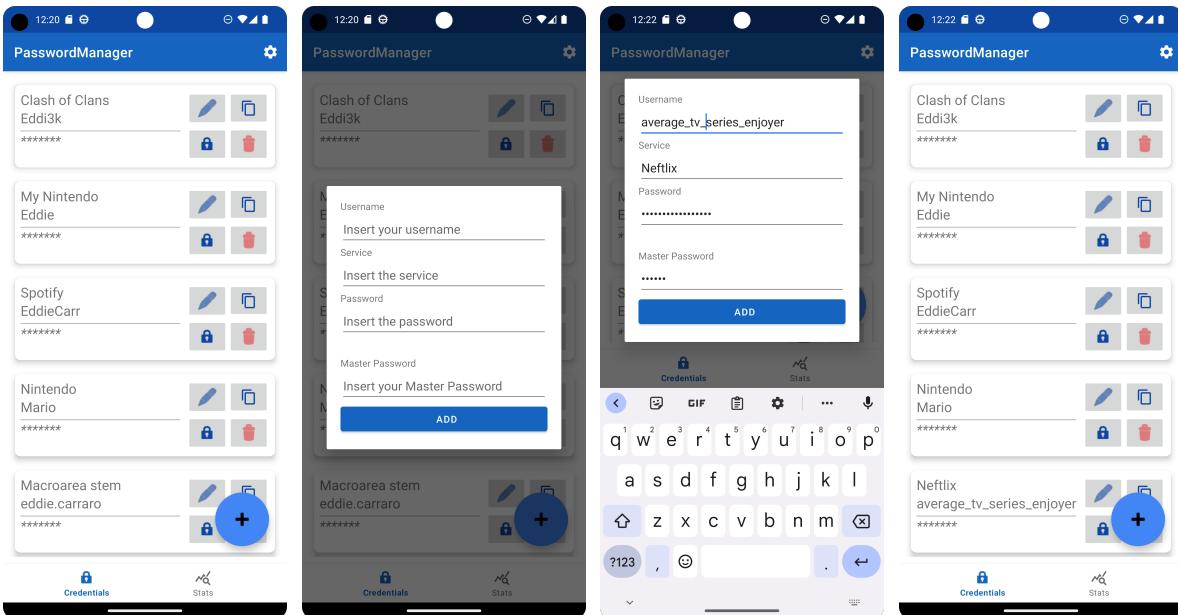


Figura 4.6: Schermata con fab

Figura 4.7: Dialog aperto

Figura 4.8: Dialog con credenziali

Figura 4.9: Credenziali aggiunte

password con priorità media e alta vengono riconosciute rispettivamente dai colori giallo e rosso, di cui si colorano l'icona triangolare di warning e la data di ultima modifica.

4.1.5 Impostazioni

La sezione "Impostazioni" è progettata per offrire agli utenti un modo semplice e intuitivo per personalizzare il comportamento e l'aspetto dell'applicazione secondo le loro preferenze. La sezione è accessibile tramite l'icona dell'ingranaggio nella barra di navigazione dell'applicazione.

All'interno della sezione, gli utenti trovano diverse opzioni consentendo loro di modificare le impostazioni relative a vari aspetti dell'applicazione. Le opzioni disponibili includono:

- **Tema dell'applicazione.** È stato implementato uno switch per consentire agli utenti di attivare o disattivare la modalità scura dell'app. La modalità scura offre una variante a contrasto elevato del tema predefinito, creando un'esperienza di utilizzo confortevole in condizioni di scarsa illuminazione;
- **Visualizzazione delle credenziali.** È stato implementato un RadioGroup che permette agli utenti di scegliere la modalità di visualizzazione delle credenziali precedentemente inserite nel database;
- **Modifica della master password.** L'utente è in grado di modificare, se lo ritiene necessario, la master password. Dopo la modifica, tutte le operazioni sulle proprie password, anche quelle già inserite, sono effettuabili con la nuova master password.

4.1.6 Implementazione delle liste

Sebbene RecyclerView.Adapter [58] sia comunemente utilizzata nelle implementazioni di una RecyclerView, l'uso del suo metodo notifyDataSetChanged() è fortemente sconsigliato [40]. Infatti, esso notifica la lista che tutto il dataset è cambiato e deve essere sostituito. Ciò

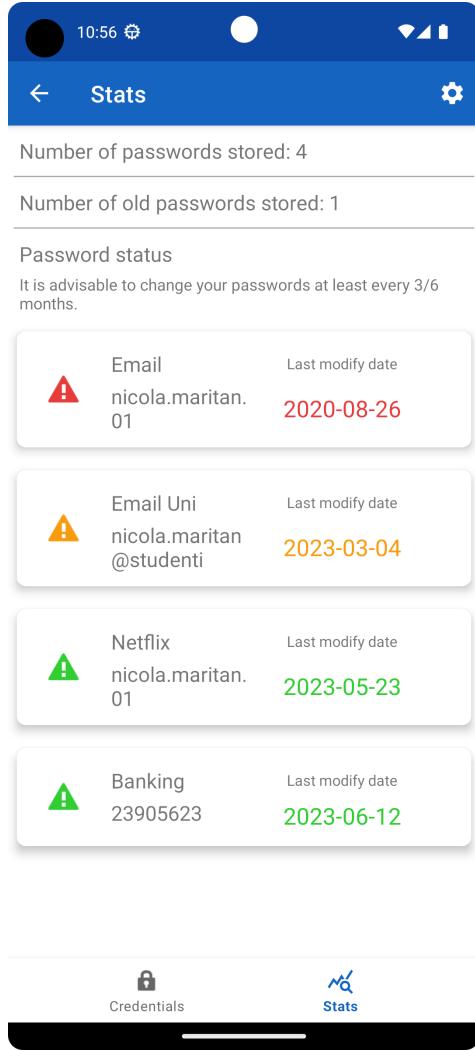


Figura 4.10: Schermata ”Statistiche” nell’app. Le credenziali aggiornate meno recentemente vengono evidenziate rispetto alle altre.

è chiaramente inefficiente, soprattutto quando il numero di elementi è alto. Per tale motivo, si è deciso di implementare l’adapter tramite `ListAdapter` [49]. Essa è in grado di aggiornare gli elementi della lista in modo efficiente, segnalando esattamente quali elementi sono stati modificati, aggiunti o rimossi. Per utilizzare tale strumento è necessario creare una sottoclassificazione di `DiffUtil.ItemCallback` [28], la quale definisce con che criteri distinguere due elementi della lista. Nel nostro caso si è definito (`com.project.passwordmanager.common`):

```
class CredentialDiffItemCallback : DiffUtil.ItemCallback<Credential>()
{
    override fun areItemsTheSame(oldItem: Credential, newItem: Credential)
    = (oldItem.id == newItem.id)

    override fun areContentsTheSame(oldItem: Credential, newItem: Credential)
    = (oldItem == newItem)
}
```

In tale modo si sono rese le modifiche alla lista più efficienti. Si noti che l’utilizzo improprio di `notifyDataSetChanged()` è segnalato dall’ambiente di sviluppo come warning.

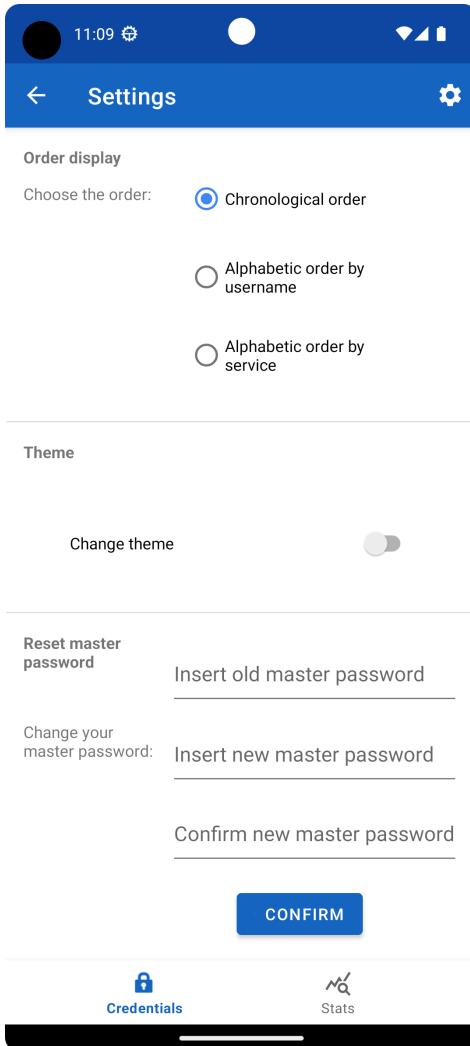


Figura 4.11: Schermata delle impostazioni (*light theme*).

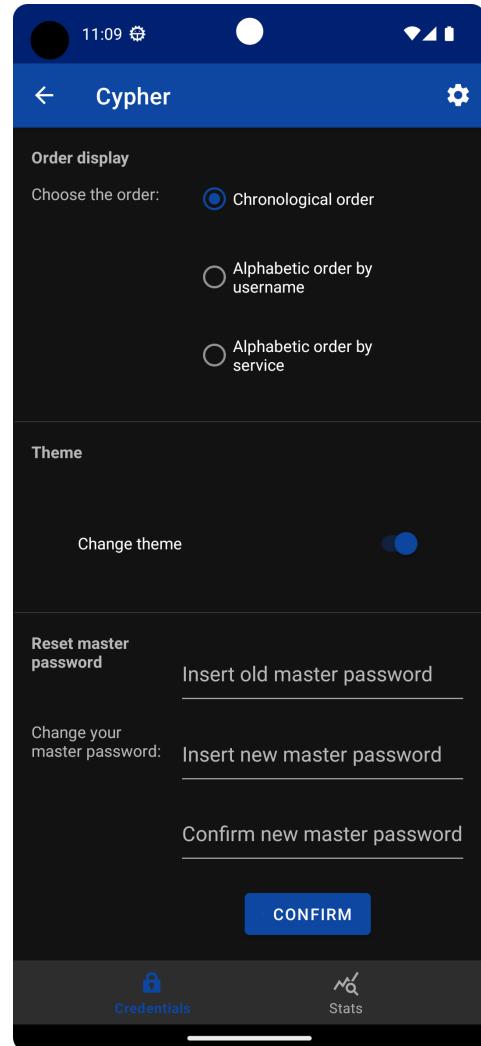


Figura 4.12: Schermata delle impostazioni (*dark theme*).

4.1.7 Model-View-ViewModel

In questa sezione si assume che i vari subpackage siano sotto `com.project.passwordmanager`. Le varie parti dell'app sono implementate seguendo il design pattern *Model-View-ViewModel* (MVVM) descritto in [40]:

- La View include il codice responsabile per la UI, come le `Activity`, i `Fragment` e i file di layout (subpackage `activities` e `fragments`).
- I ViewModel [70] sono responsabili della *business logic* e dei dati di una View (subpackage `viewmodels`).
- Il Model rappresenta i dati persistenti dell'app e elementi che li rappresentano (subpackage `model`). Nel nostro caso, il Model è composto dal database Room, il DAO, la Repository e l'Entity.

La scelta di strutturare il progetto seguendo questo design pattern porta ad una migliore organizzazione del codice e *separation of concerns* [64]. In tal modo, il codice risulta più mantenibile e comprensibile.

Inoltre, i ViewModel permettono di conservare lo stato non persistente. Alla rotazione dello schermo o alla temporanea distruzione dell'activity da parte dell'OS, i ViewModel non vengono distrutti, permettendogli di mantenere i dati che altrimenti andrebbero persi. Dunque i ViewModel forniscono una comoda ed efficacie alternativa ai meccanismi di salvataggio della UI [63].

Un ulteriore aspetto chiave dei ViewModel è la possibilità di lanciare metodi asincroni attraverso il `viewModelScope` [70], un `CoroutineScope` che segue il ciclo di vita del ViewModel. Ad esempio, attraverso:

```
viewModelScope.launch{  
    ...  
}
```

è possibile invocare metodi del DAO marchiati con `suspend` [75], i quali altrimenti, bloccherebbero il thread UI, rischiando di rendere l'app non responsiva.

4.1.8 Crittografia

La object class `Cryptography` (`com.project.passwordmanager.security`) si occupa di crittografare e decrittografare le password, utilizzando come chiave la Master Password. Essendo l'implementazione di un tale algoritmo molto complicata, si è fatto uso delle classi del package `javax.crypto` [52]. Il metodo `generateSecretKey` genera una chiave segreta di tipo `SecretKeySpec` per la crittografia AES, utilizzando l'algoritmo di hashing SHA-256 a partire dalla Master Password inserita. Tale oggetto viene utilizzato per inizializzare l'istanza di `Cipher` in `encryptText` e `decryptText`. Per l'implementazione si sono consultati diverse fonti [77, 5, 44]. Nel progetto, l'hashing è gestito dalla classe `Hashing` (`com.project.passwordmanager.security`).

4.1.9 Hashing

Cypher non salva in chiaro la Master Password per motivi di sicurezza [16]. La strategia adottata consiste nel conservare l'hash SHA-256 [67] della Master Password [41], così da rendere quasi impossibile ricavarla dalle `SharedPreferences`. Ogni autenticazione avviene confrontando lo SHA-256 della Master Password immessa con lo SHA-256 di quella effettiva.

4.1.10 Internazionalizzazione

Nel processo di sviluppo di *Cypher*, è stata data importanza all'internazionalizzazione. È stata progettata un'interfaccia utente che supporta due lingue: l'italiano e l'inglese. Ciò significa che gli utenti possono utilizzare *Cypher* nella lingua che preferiscono, rendendo l'applicazione più accessibile e ampliando notevolmente il potenziale di diffusione e adozione di dell'applicazione come soluzione affidabile per la gestione sicura delle password.

4.2 Widget

L'applicazione dispone di tre widget:

- Widget Credentials, un widget di collezioni;
- Widget Stats, un widget ibrido informativo e di collezioni;
- Widget Add, un widget di controllo.

4.2.1 Anteprime

Le anteprime dei widget sono state realizzate utilizzando le tecniche descritte in 3.6.1. Le preview sono state implementate tramite `previewLayout`, di conseguenza sono stati definiti dei layout con dei valori fittizi. Inoltre, tali valori permettono di implementare la localizzazione, mostrando alcuni valori invece di altri a seconda della lingua. In aggiunta, tale approccio permette di creare delle anteprime per il tema scuro (Figura 4.14).

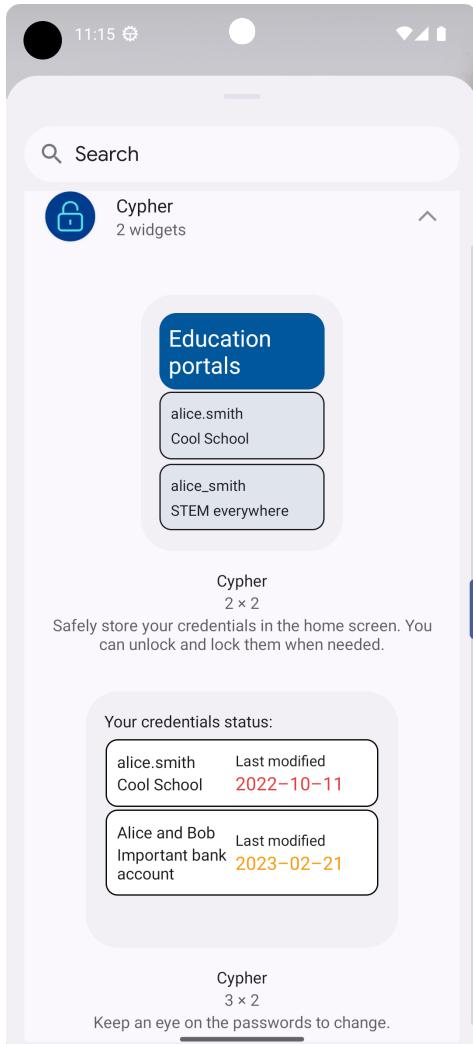


Figura 4.13: Anteprima dei widget Credentials e Stats. Essi contengono dati fittizi, per dare un’idea all’utente del loro aspetto prima di posizionarli sullo schermo.

4.2.2 Credentials widget

Il widget Credentials è un widget di collezioni che permette di conservare un insieme di password e di poterle sbloccare nel momento del bisogno. In questo modo, l’utente può ricorrere allo sblocco e consultazione di una password senza dover aprire l’app. Il widget utilizza dei layout responsivi per meglio adattarsi allo spazio disponibile, con unica limitazione la dimensione minima di 2×2 blocchi della schermata home. Ogni widget ha un widget ID unico attribuito dal sistema e sono completamente indipendenti l’uno dall’altro.

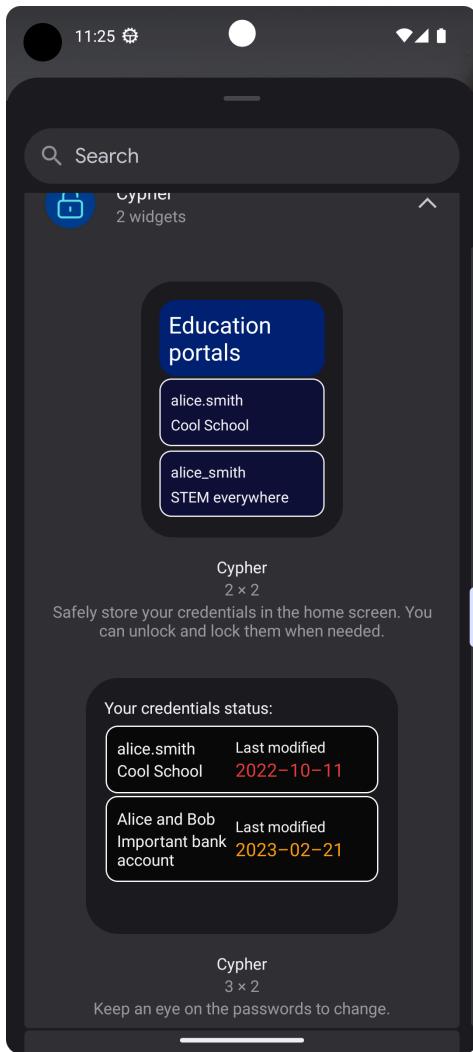


Figura 4.14: Anteprima dei widget Credentials e Stats con il tema scuro.

Unicità dei widget

Ogni widget può contenere qualunque combinazione delle password registrate in *Cypher*. Di conseguenza, l'utente ha la facoltà di decidere quante e quali password inserire nel widget, e di creare o meno altri widget con altre password. La funzionalità è stata concepita con la possibilità di raggruppare semanticamente le proprie credenziali. Infatti, l'utente può attribuire ad ogni widget un proprio nome. Un esempio di istanze indipendenti è mostrato in Figura 4.15.

Dal punto di vista tecnico, le informazioni di ogni widget sono salvate tramite `SharedPreferences`. Conservare i dati in una classe apposita (statica o meno) portava alla loro perdita nel momento in cui l'app veniva chiusa, portando a crash e malfunzionamenti da parte del widget. Di conseguenza si è optato di salvare i dati nelle `SharedPreferences`. Non si è deciso di salvare i dati a database a causa dell'overhead creatosi su di esso durante una sperimentale implementazione. I dati relativi ad un widget vengono salvati al momento della allocazione e liberate quando il widget viene eliminato dalla scheramta home. Per realizzare tale meccanismo, è fondamentale riferirsi all'id del widget. Per facilitare lo sviluppo e permettere un più facile mantenimento del codice, è stata creata la classe `WidgetPreferencesManager` (`com.project.passwordmanager.common`). Essa crea un layer di astrazione

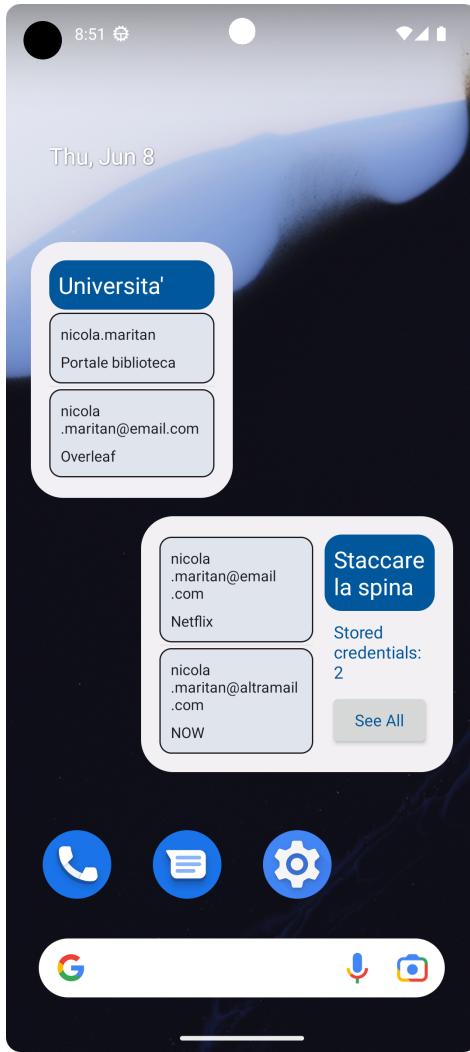


Figura 4.15: Diverse istanze del widget Credentials. Ogni istanza è a sé stante, ha un proprio nome e il proprio set di password. In questo caso le password sono raggruppate semanticamente.

sopra l'inserimento delle **SharedPreferences**. Per il suo utilizzo, è necessario passare al costruttore un **Context** e un **Int** indicante il widget ID dell'istanza su cui si vuole agire. Ad esempio, se si vuole impostare le credenziali da aggiungere a quel widget si invoca **setAddedIds(List<Int>)**, passando la lista di ID delle credenziali da aggiungere. Volendo ottenere gli ID delle credenziali inserite, si utilizza **getAddedIds() : List<Int>**.

Layout responsivi

Il widget Credentials è implementato tramite layout responsivi (3.7.1). I layout implementati sono tre:

- Normale
- Alto
- Largo

Il view mapping impiegato è il seguente:

```

mapOf(
    SizeF(80f, 80f) to smallView,
    SizeF(80f, 350f) to tallView,
    SizeF(240f, 200f) to wideView
)

```

Nelle Figure 4.18, 4.18 e 4.20 sono riportati i tre layout in questione. Si noti come il layout alto e il layout largo permettano di intravedere più elementi. In particolare, il layout alto permette di vedere il numero di elementi conservati nel widget, mentre il layout largo permette di vedere anche un pulsante che rimanda alla schermata *Credentials*. Inoltre, tutti i layout implementano una corrispettiva versione per il tema scuro. Le due versioni sono confrontabili in Figura 4.16 e 4.17.

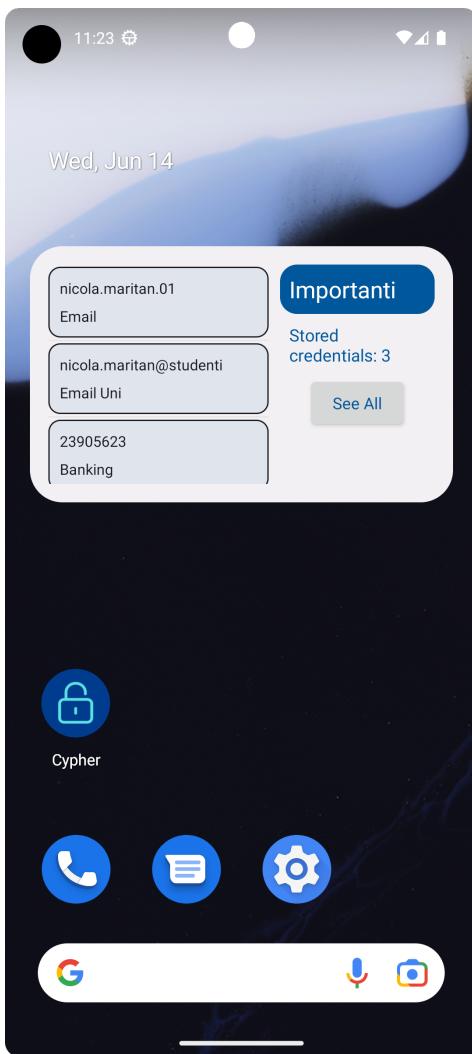


Figura 4.16: Widget Credentials (*light theme*).

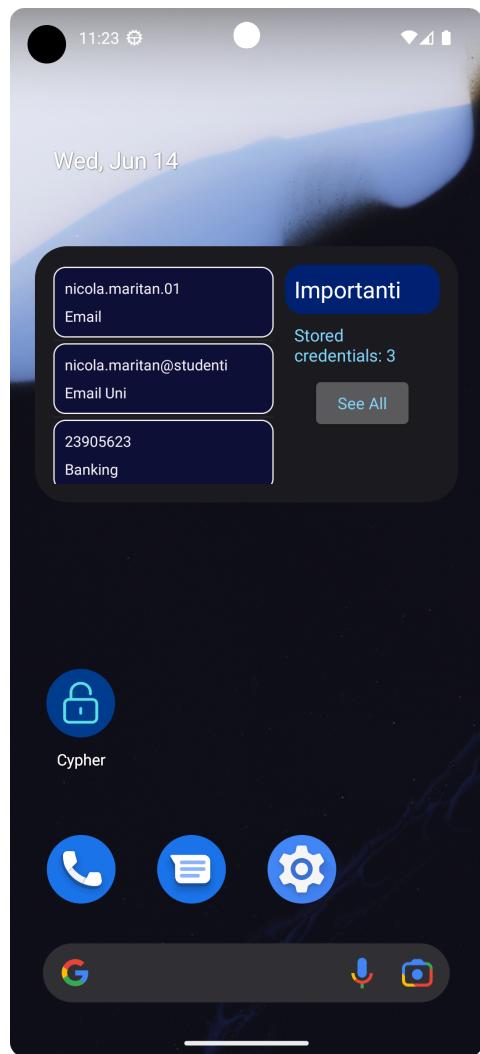


Figura 4.17: Widget Credentials (*dark theme*).

Activity di configurazione

Il widget Credentials è configurabile al momento del suo inserimento nella schermata home. È possibile personalizzare le credenziali da inserire e il nome da attribuire al widget. Un esempio di tale procedura è riportata nelle Figure 4.21 e 4.22.

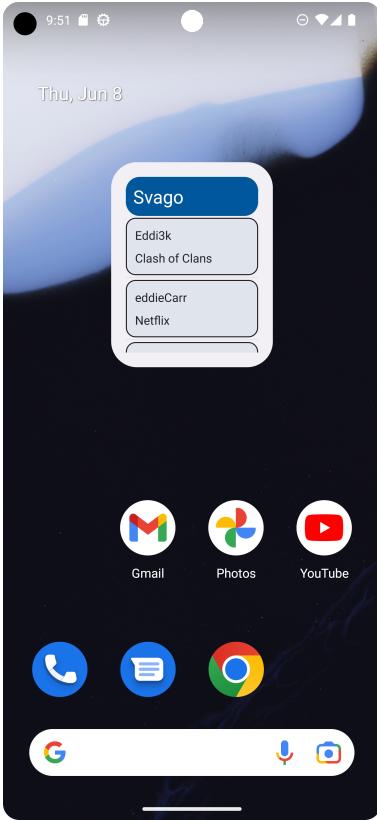


Figura 4.18: Layout normale del widget

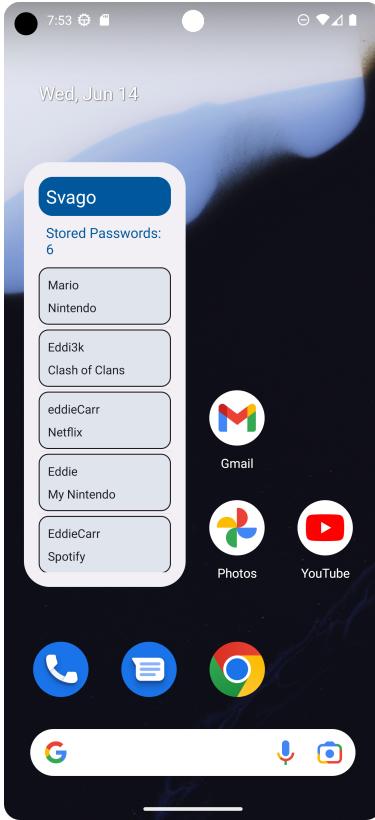


Figura 4.19: Layout alto del widget

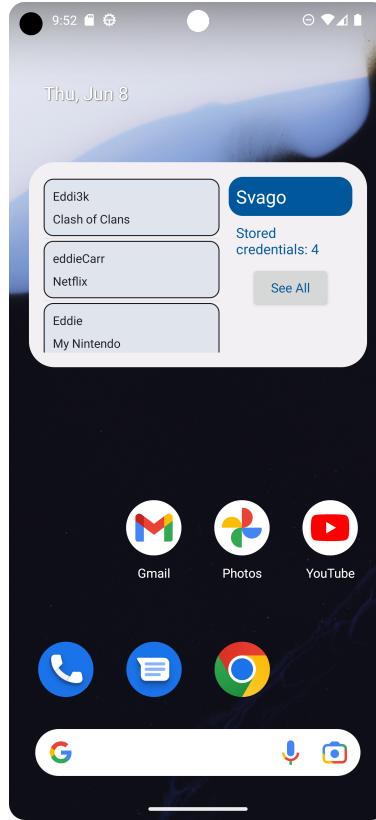


Figura 4.20: Layout largo del widget

Sblocco di una password

Per mantenere al sicuro le proprie password, esse non possono essere visibili nella home screen dell’utente. Per visualizzare una password, l’utente deve sbloccarla singolarmente cliccando sul corrispettivo elemento nel widget. Ciò apre una schermata in cui è possibile autenticarsi e visualizzare la propria password. Inoltre, è presente un pulsante di copia, con il quale l’utente può copiare la propria password nella clipboard. La procedura è riportata nelle Figure 4.23, 4.24 e 4.25. Dopo aver sbloccato la password è possibile ribloccarla nell’activity, nel caso in cui l’utente necessiti di risbloccarla a breve, senza chiudere l’app. Nonostante ciò, se l’utente non esegue la procedura di blocco, la password rimane al sicuro, e la prossima volta sarà necessario ancora una volta il suo sblocco.

Implementazione

Nel progetto è implementato con il provider `CredentialsWidget` e il service `Credential-
sWidgetService` (`com.project.passwordmanager.widgets.credentials`). Tutti i layout visualizzano la lista di credenziali inserite dall’utente. Essa è stata implementata tramite una `ListView`, i cui dati sono provvisti dal servizio `CredentialsWidgetService`. La funzione di adapter è svolta da `CredentialsCredentialsWidgetFactory`. Tale meccanismo è trattato nel dettaglio in 3.4. In particolare, il service rimane in ascolto di un `LiveData<List<Credentials>>`^[50] dell’attributo in `CredentialsRepository` [62]. Questa ottiene i dati attraverso *Data Access Object* (DAO) [26] del database. Ne consegue che la rimozione o modifica di elementi inseriti nel widget da parte dell’utente (per esempio dall’app) si ripercuote nel widget in modo automatico, il quale non mostrerà più l’elemento non più presente

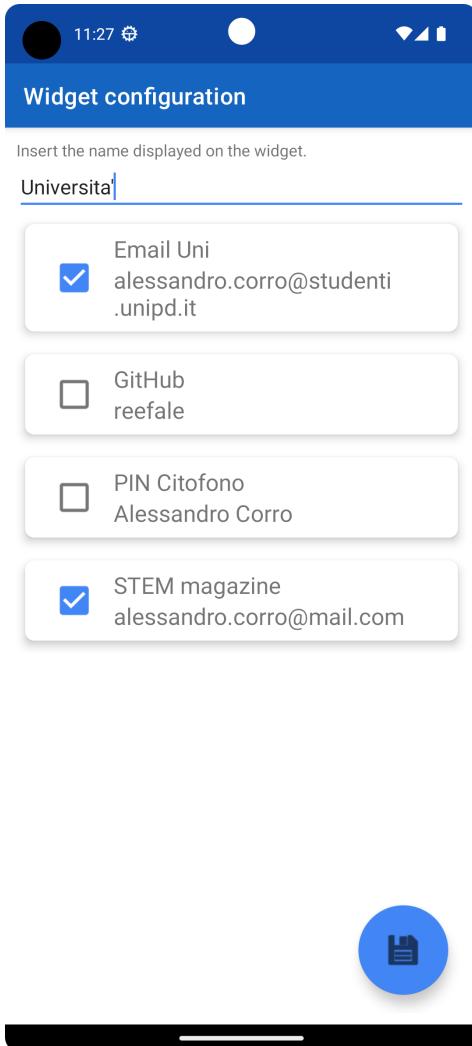


Figura 4.21: Activity di configurazione del widget.

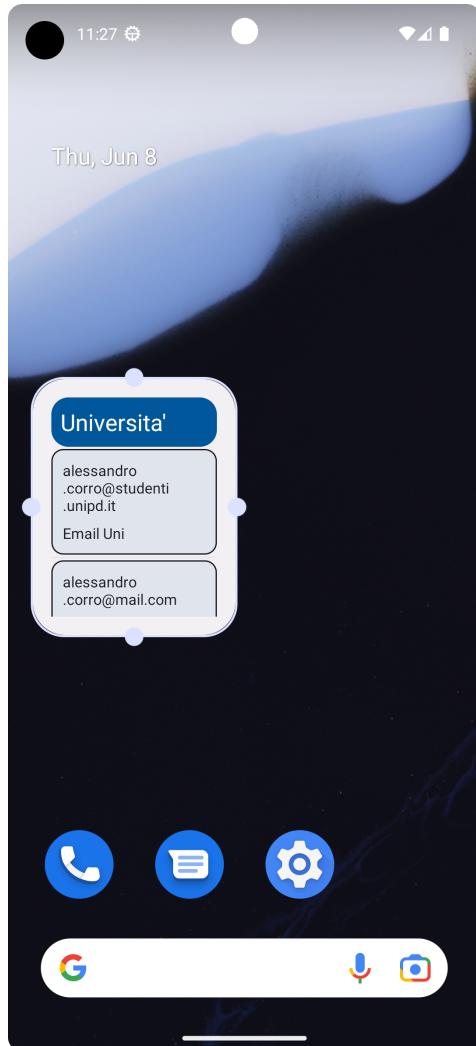


Figura 4.22: Widget piazzato dopo la configurazione.

nel database o aggiornerà le proprietà di quello modificato. Si è optato per questa soluzione invece che di quella descritta in 3.4.7 dal momento che la prima è la più flessibile nell’istanziamento dei diversi tipi di layout. Ogni elemento della lista permette di invocare la activity di sblocco trattata in 4.2.2. Ciò è stato realizzato utilizzando dei `fillInIntent` per ogni item e completandoli con un `PendingIntent`, come descritto in 3.4.5.

Il pulsante visibile nel layout largo lancia l’applicazione, permettendo di vedere tutte le credenziali salvate. A tale scopo, al momento del posizionamento del widget nello schermo e ad ogni ciclo di update, vengono creati tutti e tre i layout, ognuno con le proprie sfaccettature e funzioni, e solo uno di questi viene selezionato dal sistema e visualizzato a schermo basandosi sul view mapping. Per far ciò sono stati creati dei metodi helper statici in `CredentialsWidget` che agevolano il setup della vista. I metodi in questione sono:

- `setupItemClick(RemoteViews, Context, Int)`
- `setupWidgetName(RemoteViews, Context, Int)`
- `setupCredentialsNumber(RemoteViews, Context, Int)`
- `setupSeeAllButton(RemoteViews, Context, Int)`

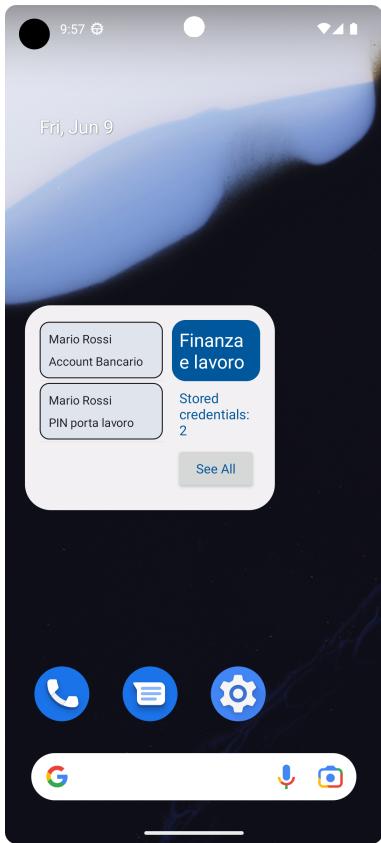


Figura 4.23: Widget visualizzato nello schermo

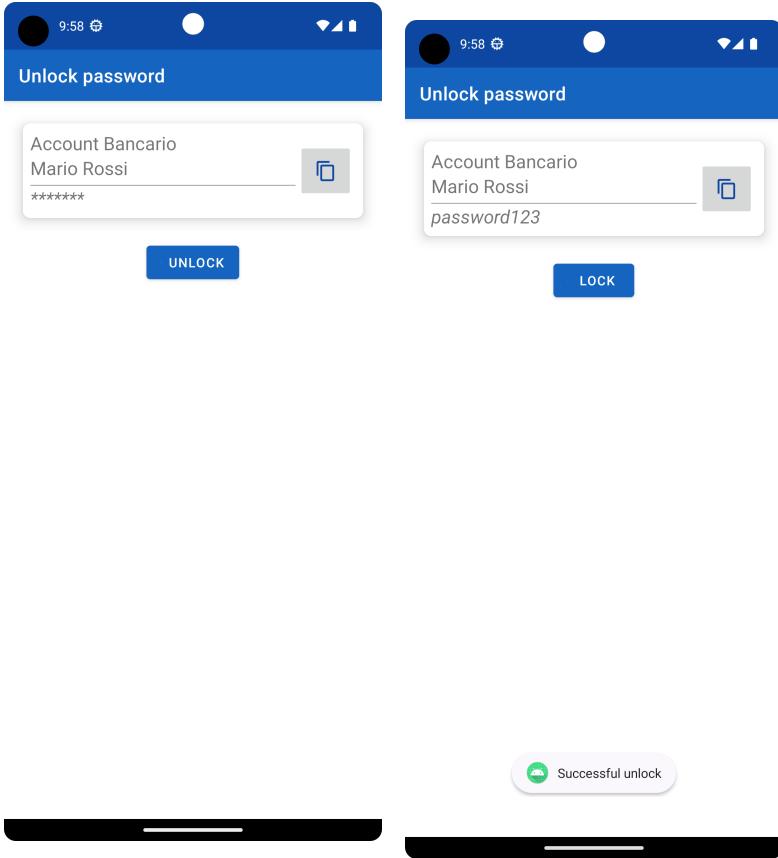


Figura 4.24: Schermata di sblocco, successiva al click del primo elemento nel widget.

Figura 4.25: Schermata dopo l'autenticazione.

Ad esempio, `setupSeeAllButton` viene invocato solo alla creazione del layout largo, mentre `setupItemClick` e `setupWidgetName` sono impiegati all'inizializzazione di tutti e tre.

4.2.3 Stats widget

Il widget descritto in questa sezione ricade nella categoria dei widget ibridi, in particolare è un widget di collezioni ed un widget informativo. Esso è una versione ridotta della schermata Stats nella `MainActivity`, dunque mostra la data di ultima modifica di una credenziale e segnala con i colori rosso e arancione quelle vecchie più di 6 e 3 mesi rispettivamente. Come nella sua controparte nella activity, anche qui vengono mostrate in cima alla lista le credenziali che necessitano di essere cambiate al più presto.

Non unicità del widget

A differenza del precedente, ogni istanza di questo widget non ha un proprio stato. In questo modo, l'utente può conoscere la data di ultima modifica di tutte le proprie credenziali, e accorgersi non appena una di queste invecchia di 3 o 6 mesi. Il fatto che ogni widget sia uguale non implica l'impossibilità di avere più istanze nella home screen. Se l'utente decide di posizionare più widget di questo tipo, questi saranno tutti funzionali, semplicemente mostreranno sempre lo stesso contenuto.

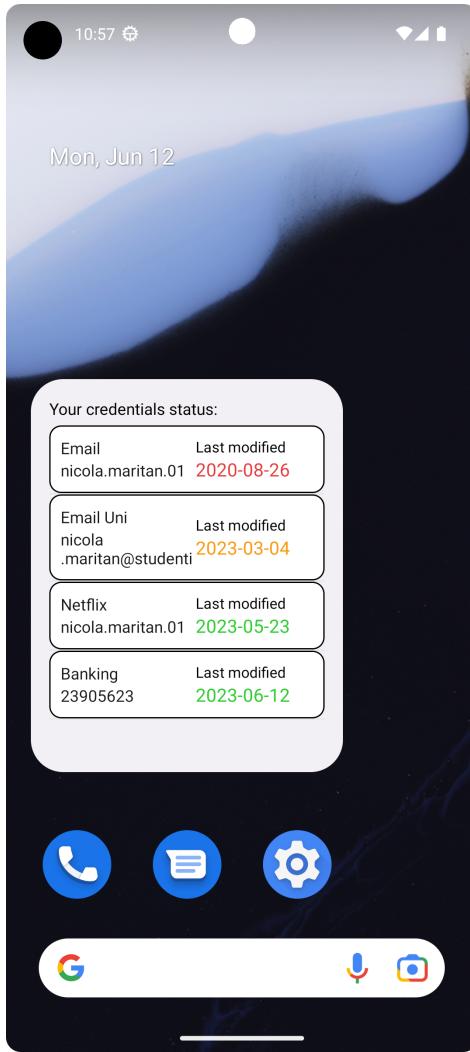


Figura 4.26: Widget delle Statistiche. Le credenziali hanno diversi date di ultima modifica.

Layout

Il widget non implementa layout responsivi, ma bensì un unico layout che viene riadattato dal sistema a seconda della dimensione del widget. Il widget ha una dimensione minima: non potrà essere più piccolo di 3×2 celle della schermata home. Solo in questo modo l'utente può sfruttare l'utilità di questo widget informativo. Per quanto riguarda la dimensione massima, non ci sono limiti di ridimensionamento. Come il widget precedente, è stato implementato una versione del layout per il tema scuro (Figura 4.27 e 4.28).

Implementazione

L'implementazione di tale widget è diversa da quella del widget Credentials. Infatti, la lista di elementi (come prima, un UI widget `ListView`) presente nella collezione è stata implementata tramite la procedura descritta in 3.4.7, dove vengono descritti anche i conseguenti benefici. L'assenza di una activity di configurazione [31] rende più semplice l'implementazione di tale approccio. Infatti, per questo widget non è stato necessario implementare alcuna sottoclasse di `RemoteViewsService`. L'unica classe implementata è `StatsWidget` (`com.project.passwordmanager.widgets.stats`). La costruzione del `RemoteCollectionI-`

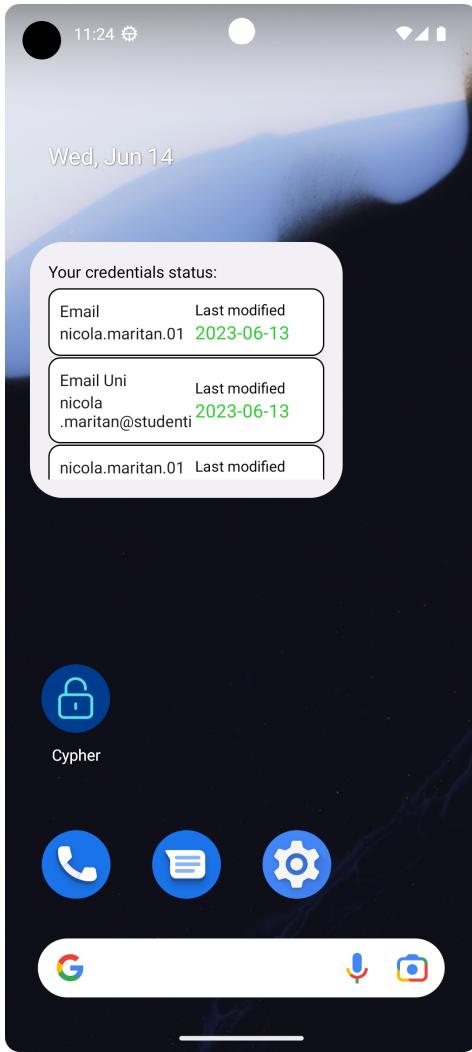


Figura 4.27: Widget delle Statistiche (*light theme*).

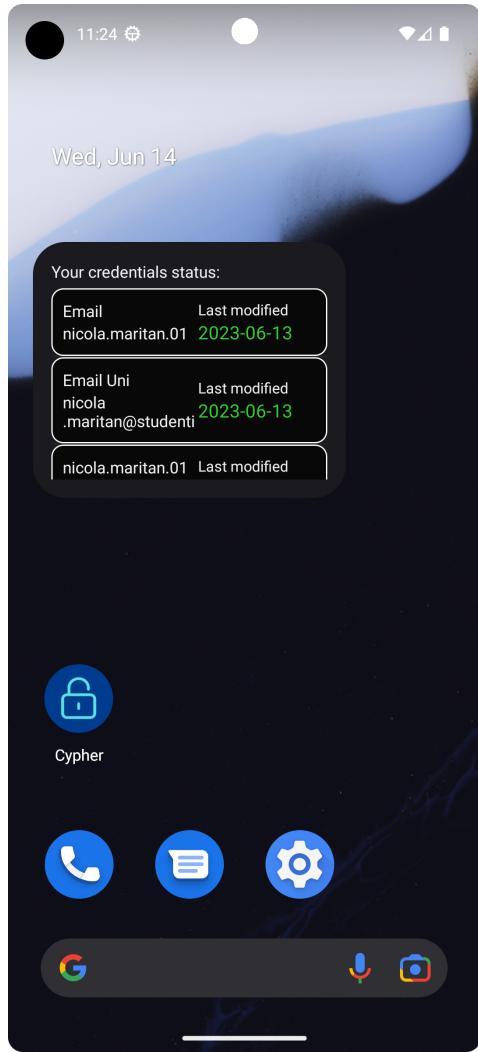


Figura 4.28: Widget delle Statistiche (*dark theme*).

temi avviene rimanendo in ascolto di `LiveData<List<Credentials>>` in `CredentialsRepository`. Come nel il widget precedente, rimozioni o modifiche al database si ripercuotono sul widget grazie al `LiveData`. A differenza del Widget Credentials, anche le nuove credenziali aggiunte saranno visibili nel Widget Stats, dal momento che il suo contenuto non viene configurato dall'utente. Il codice utilizzato per tale implementazione nel provider è analogo al seguente:

```
// Observe the credentials from the repository
credentialRepository.allCredentialSortedByDate.observeForever { credentials
->

    // Create a RemoteViews for each credential
    val builder = RemoteViews.RemoteCollectionItems.Builder()
    for (i in credentials.indices)
    {
        val credential = credentials[i]
        val itemViews =
            RemoteViews("com.example.app:id/item_credentials", R.layout.item_credentials)
        itemViews.setTextViewText(R.id.item_label, credential.name)
        itemViews.setTextViewText(R.id.item_last_modified, credential.lastModified)
        itemViews.setImageViewResource(R.id.item_image, credential.icon)
        builder.addItem(i, itemViews)
    }
}
```

```

        RemoteViews(context.packageName, R.layout.stats_widget_listview_item)

        // Layout setup, like setting text, color, etc

        builder.addItem(i.toLong(), itemViews)
    }

    // There is only one type of view.
    builder.setViewTypeCount(1)

    // Setup remote adapter
    val remoteCollectionItems = builder.build()
    views.setRemoteAdapter(R.id.list_view, remoteCollectionItems)

    appWidgetManager.updateAppWidget(appWidgetId, views)
}

```

4.2.4 Add widget

Questo widget è un widget di controllo che permette di aprire una schermata di inserimento di una nuova credenziale a partire dalla home screen. Fra i tre widget implementati, questo è il più semplice, dal momento che non deve conservare stato o mostrare liste di elementi. Di conseguenza, ogni istanza del widget è uguale alle altre, e possono coesistere liberamente nella schermata home. Si è deciso di implementare un widget di controllo che permettesse di aggiungere una nuova credenziale in modo rapido ed efficace. La procedura di inserimento è riportata in Figura 4.29, 4.30 e 4.31.

Layout

A differenza degli altri, tale widget è molto più limitato, in quanto permette il solo ridimensionamento orizzontale. In particolare, le dimensioni possibili sono 1×1 , 1×2 , 1×3 e 1×4 . Ciò è dovuto al fatto che il widget non mostra particolari informazioni, ed è pensato per essere facilmente piazzabile nella schermata dell'utente fra le applicazioni e gli altri widget. Inoltre, lo sfondo tipico dei widget in Android in questo caso è assente. In questo modo è possibile visualizzare solo il layout definito nei file XML, che in tal caso è uno sfondo blu custom. I layout del widget sono responsivi. La differenza fra i vari layout si limita al testo visualizzato e alla sua dimensione. Le viste supportate sono `smallView` (1×1 e 1×2), `normalView` (1×3), e `wideView` (1×4), ognuna corrispondente alle dimensioni sopracitate. Il view mapping impiegato è il seguente:

```

val viewMapping = mapOf(
    SizeF(150f, 80f) to smallView,
    SizeF(250f, 80f) to normalView,
    SizeF(300f, 80f) to wideView
)

```

Implementazione

Cliccando su una qualunque superficie del widget viene invocata l'activity di inserimento della credenziale. A differenza dei casi più comuni (come nel caso di Widget Credentials), però, non è un pulsante ad essere premuto, ma il layout del widget stesso. Infatti, per ottenere tale

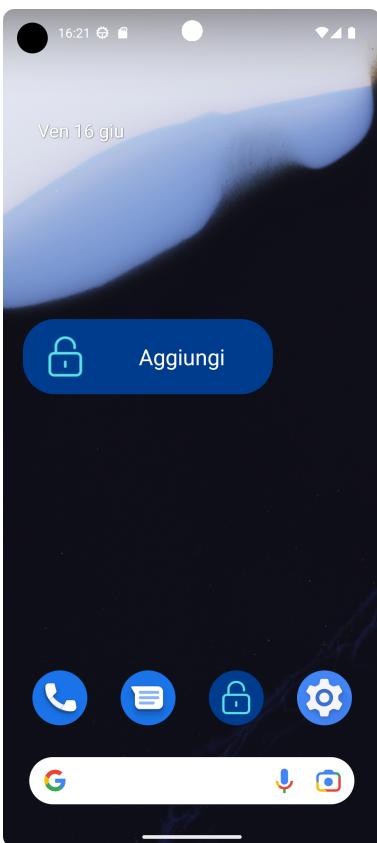


Figura 4.29: Widget visualizzato nello schermo

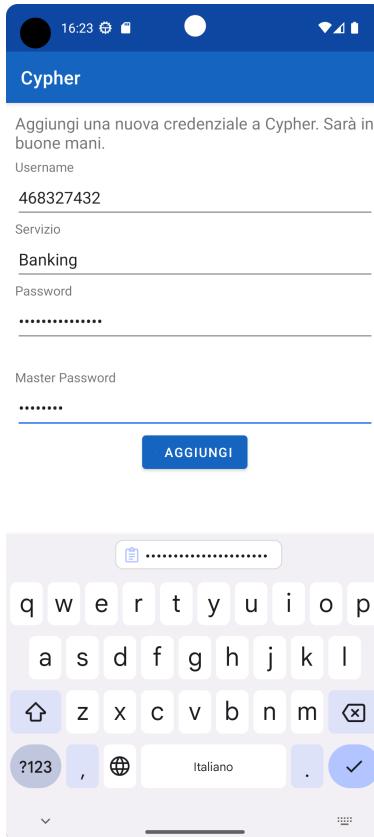


Figura 4.30: Schermata di inserimento, successiva al click sul widget.

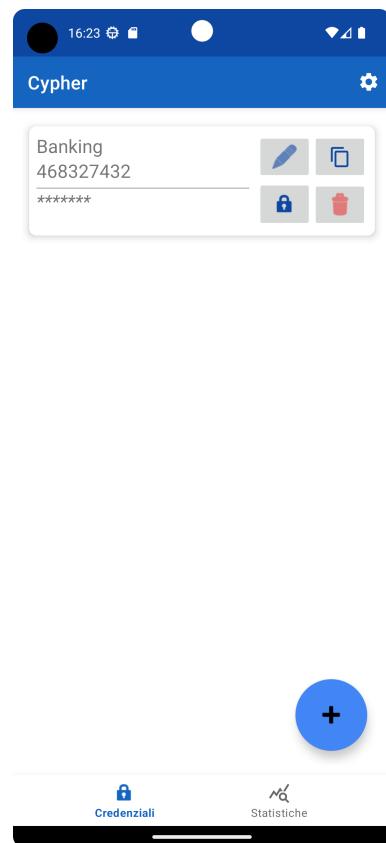


Figura 4.31: La credenziale è stata inserita correttamente.

comportamento, è necessario rendere il layout responsabile di lanciare un intent. Il codice è analogo a quello della impostazione di un `pendingIntent` per un pulsante:

```
val views: RemoteViews = RemoteViews(
    context.packageName,
    R.layout.add_credential_widget
).apply {
    setOnClickPendingIntent(R.id.add_credential_widget_layout_id, pendingIntent)
}
```

Chiaramente, tale comportamento è implementato per ognuno dei tre layout disponibili. In particolare, è necessario costruire tutti e tre i layout e assegnare i relativi `pendingIntent`, per poi selezionare il layout corretto in base alla dimensione, tramite view mapping.

4.3 Conclusioni

Cypher è stato concepito e realizzato tenendo in mente i widget che lo avrebbero accompagnato. Si sono realizzati tre widget, rappresentanti delle tre tipologie principali trattate in [10]. I widget permettono di attingere alle funzionalità chiave dell'applicazione a partire della propria schermata home in modo rapido ed efficace. Nella loro implementazione si sono sfruttati molti aspetti tecnici trattati nel Capitolo 3 e si sono seguite le linee guida descritte nel Capitolo 2. Sono state implementate delle collezioni nei due modi principali descritti nella documentazione 3.4, entrambi estraendo dati da un database Room attraverso una repository.

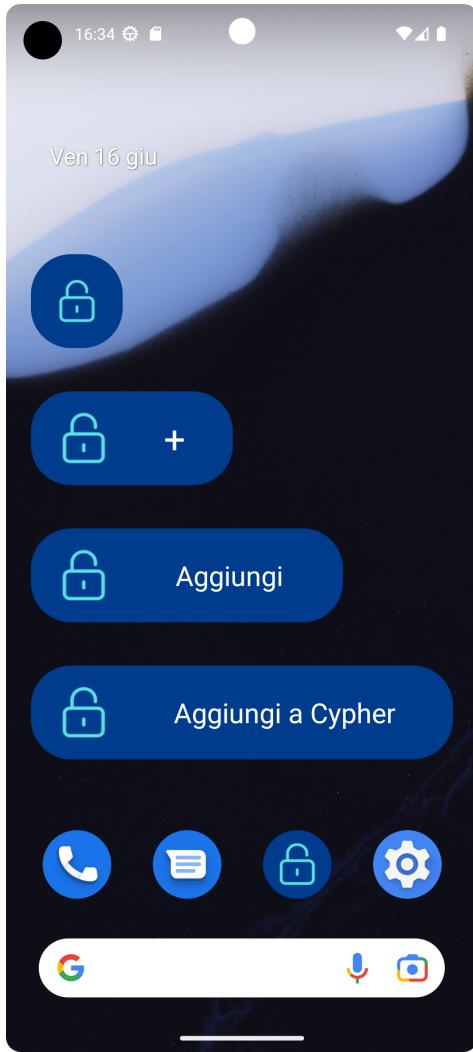


Figura 4.32: I 4 layout possibili per Widget Add.

I widget contengono elementi cliccabili e layout dinamici, che si adattano alla sua dimensione e presentano delle varianti per il tema chiaro e scuro del dispositivo. Inoltre, ogni widget è ridimensionabile in modo differente (nelle due dimensioni o solo orizzontalmente), e due di questi implementano layout responsivi. Si è implementata la possibilità di configurare un widget tramite una activity di configurazione, distinguendo il contenuto di ogni istanza. I tre widget sviluppati e l'icona dell'applicazione sono riportate in Figura 4.33.

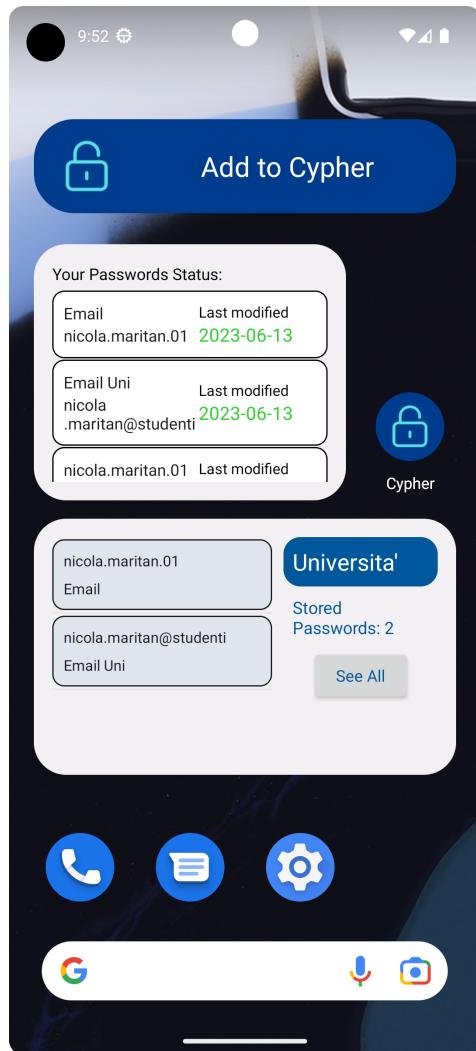


Figura 4.33: Widget e icona di *Cypher*.

Bibliografia

- [1] *A brief history of widgets*. URL: <https://www.theverge.com/2014/6/6/5786814/a-brief-history-of-widgets>.
- [2] *Accurate Previews — Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets/advanced#build-accurate-previews>.
- [3] *Add a Floating Action Button — Android Developers*. URL: <https://developer.android.com/develop/ui/views/components/floating-action-button>.
- [4] *Add apps, shortcuts and widgets to your Home screens*. URL: <https://support.google.com/android/answer/9450271?hl=en>.
- [5] *AES Encryption and Decryption in Java(CBC Mode)*. URL: <https://www.javacodegeeks.com/2018/03/aes-encryption-and-decryption-in-javabc-mode.html>.
- [6] *Android 12 widgets improvements — Android Developers*. URL: <https://developer.android.com/about/versions/12/features/widgets>.
- [7] *ANRs — Android Developers*. URL: <https://developer.android.com/topic/performance/vitals/anr>.
- [8] *App Actions built-in intents — Android Developers*. URL: <https://developer.android.com/reference/app-actions/built-in-intents>.
- [9] *App Widget Provider*. URL: <https://developer.android.com/reference/android/appwidget/AppWidgetProvider>.
- [10] *App widgets overview*. URL: <https://developer.android.com/develop/ui/views/appwidgets/overview>.
- [11] *AppWidgetHost — Android Developers*. URL: <https://developer.android.com/reference/android/appwidget/AppWidgetHost>.
- [12] *AppWidgetHostView — Android Developers*. URL: <https://developer.android.com/reference/android/appwidget/AppWidgetHostView>.
- [13] *Audio Widget pack — Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.widgets.music&hl=en&gl=US>.
- [14] *Behavior changes: Apps targeting Android 12 — Android Developers*. URL: <https://9to5google.com/2021/10/19/google-android-12-widgets/>.
- [15] *Behavior changes: Apps targeting Android 12 — Android Developers*. URL: <https://developer.android.com/about/versions/12/behavior-changes-12#notification-trampolines>.
- [16] *Best option to store username and password in android app - Stack Overflow*. URL: <https://stackoverflow.com/a/9233198>.
- [17] *Broadcast overview — Android Developers*. URL: <https://developer.android.com/guide/components/broadcasts>.

- [18] *BroadcastReceiver* — *Android Developers*. URL: <https://developer.android.com/reference/android/content/BroadcastReceiver>.
- [19] *Come mettere i widget su Android*. URL: <https://www.aranzulla.it/come-mettere-i-widget-su-android-1266043.html>.
- [20] *Content providers* — *Android Developers*. URL: <https://developer.android.com/guide/topics/providers/content-providers>.
- [21] *Create a scrollable widget for your Android app*. URL: <https://www.androidauthority.com/create-an-android-widget-1020839/>.
- [22] *Create a simple widget* — *Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets>.
- [23] *Create a strong password and a more secure account*. URL: <https://support.google.com/accounts/answer/32040?hl=en>.
- [24] *Create an advanced widget* — *Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets/advanced>.
- [25] *Create the widget layout*. URL: <https://developer.android.com/develop/ui/views/appwidgets#layout>.
- [26] *Dao* — *Android Developers*. URL: <https://developer.android.com/reference/android/arch/persistence/room/Dao>.
- [27] *Desk accessory - Wikipedia*. URL: https://en.wikipedia.org/wiki/Desk_accessory.
- [28] *DiffUtil.ItemCallback* — *Android Developers*. URL: <https://developer.android.com/reference/androidx/recyclerview/widget/DiffUtil.ItemCallback>.
- [29] *Digital Clock Widget - Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.maize.digitalClock&hl=en&gl=US>.
- [30] *Enable users to configure app widgets* — *Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets/configuration>.
- [31] *Enable users to configure app widgets* — *Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets/configuration>.
- [32] *Enhance your widget* — *Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets/enhance>.
- [33] *Gallery Widget* — *Apps on Google Play*. URL: https://play.google.com/store/apps/details?id=com.kuma.gallerywidget&hl=en_US.
- [34] *Gmail* — *Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.google.android.gm&hl=it>.
- [35] *Gmail 2021.10.31 adds revamped Material You widget*. URL: <https://9to5google.com/2021/11/22/gmail-material-you-widget/>.
- [36] *Google Calendar* — *Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.google.android.calendar&hl=en&gl=US>.
- [37] *Google Drive* — *Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.google.android.apps.docs&hl=en&gl=US>.
- [38] *Google Maps* — *Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.google.android.apps.maps&hl=en&gl=US>.
- [39] *Google Photos* — *Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.google.android.apps.photos&hl=en&gl=US>.

- [40] David Griffiths. *Head First Android Development, 3rd edition.*
- [41] *Hashing Passwords: One-Way Road to Security.* URL: <https://auth0.com/blog/hashing-passwords-one-way-road-to-security/>.
- [42] *How to add and edit widgets on your iPhone.* URL: <https://support.apple.com/en-us/HT207122>.
- [43] *How to use Samsung smart widgets.* URL: <https://www.androidpolice.com/how-to-use-samsung-smart-widgets/>.
- [44] *Inizialization Vector for Encryption — Baeldung.* URL: <https://www.baeldung.com/java-encryption-iv>.
- [45] *Intent — Android Developers.* URL: <https://developer.android.com/reference/android/content/Intent>.
- [46] *Intents and intent filters — Android Developers.* URL: <https://developer.android.com/guide/components/intents-filters>.
- [47] *Introducing home screen widgets and the AppWidget framework.* URL: <https://android-developers.googleblog.com/2009/04/introducing-home-screen-widgets-and.html>.
- [48] *JobScheduler — Android Developers.* URL: <https://developer.android.com/reference/android/app/job/JobScheduler>.
- [49] *ListAdapter — Android Developers.* URL: <https://developer.android.com/reference/androidx/recyclerview/widget/ListAdapter>.
- [50] *LiveData overview — Android Developers.* URL: <https://developer.android.com/topic/libraries/architecture/livedata>.
- [51] *Meteo and Orologio Widget - Apps on Google Play.* URL: <https://play.google.com/store/apps/details?id=com.devexpert.weather&hl=it&pli=1>.
- [52] *Package javax.crypto.* URL: <https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html>.
- [53] *PendingIntent — Android Developers.* URL: <https://developer.android.com/reference/android/app/PendingIntent>.
- [54] *Provide flexible widget layouts.* URL: <https://developer.android.com/develop/ui/views/appwidgets/layouts>.
- [55] *Provide flexible widget layouts — Android Developers.* URL: <https://developer.android.com/develop/ui/views/appwidgets/layouts>.
- [56] *Provide flexible widget layouts — Android Developers.* URL: <https://developer.android.com/develop/ui/views/appwidgets/layouts>.
- [57] *Provider info.* URL: <https://developer.android.com/reference/android/appwidget/AppWidgetProviderInfo>.
- [58] *RecyclerView.Adapter.* URL: <https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter>.
- [59] *RemoteViews — Android Developers.* URL: <https://developer.android.com/reference/android/widget/RemoteViews>.
- [60] *RemoteViewsService — Android Developers.* URL: <https://developer.android.com/reference/android/widget/RemoteViewsService>.
- [61] *RemoteViewsService.RemoteViewsFactory.* URL: <https://developer.android.com/reference/android/widget/RemoteViewsService.RemoteViewsFactory>.

- [62] *Repository Pattern* — *Android Developers*. URL: <https://developer.android.com/codelabs/basic-android-kotlin-training-repository-pattern#0>.
- [63] *Save UI states* — *Android Developers*. URL: <https://developer.android.com/topic/libraries/architecture/saving-states#onsaveinstancestate>.
- [64] *Separation of concerns* - *Wikipedia*. URL: https://en.wikipedia.org/wiki/Separation_of_concerns.
- [65] *Service* — *Android Developers*. URL: <https://developer.android.com/reference/android/app/Service>.
- [66] *Services overview* — *Android Developers*. URL: <https://developer.android.com/guide/components/services>.
- [67] *SHA-2* - *Wikipedia*. URL: <https://en.wikipedia.org/wiki/SHA-2>.
- [68] *Use collection widgets* — *Android Developers*. URL: <https://developer.android.com/develop/ui/views/appwidgets/collections>.
- [69] *View* — *Android Developers*. URL: <https://developer.android.com/reference/android/view/View>.
- [70] *ViewModel overview* — *Android Developers*. URL: <https://developer.android.com/topic/libraries/architecture/viewmodel>.
- [71] *What is stock Android? Everything you need to know about the core OS*. URL: <https://www.androidauthority.com/what-is-stock-android-845627/>.
- [72] *What is the difference between a View and widget in Android?* URL: <https://stackoverflow.com/questions/5168549/what-is-the-difference-between-a-view-and-widget-in-android>.
- [73] *Widget just got better in Android 12* — *Google*. URL: <https://blog.google/products/android/widgets-just-got-better-android-12/>.
- [74] *WorkManager* — *Android Developers*. URL: <https://developer.android.com/reference/androidx/work/WorkManager>.
- [75] *Write asynchronous DAO queries* — *Android Developers*. URL: <https://developer.android.com/training/data-storage/room/async-queries>.
- [76] *YouTube Music - Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.google.android.apps.youtube.music&hl=en&gl=US>.
- [77] zuoky/*Encryptor.java*. URL: <https://gist.github.com/zuoky/6df23bc613e457c0baf7f0f0ebc4b1f6>.