Capitolo 2

Il progetto ApoBus e la creazione dell'ambiente con Android Studio

In questo capitolo

- **Creazione del progetto in Android Studio**
- **Che cosa abbiamo realizzato**
- **Utilizzo di Gradle**
- **I sorgenti e le risorse del progetto**
- **Eseguiamo l'applicazione creata**

Questo libro vuole seguire un approccio molto pratico, che fornisca i concetti fondamentali lasciando al lettore gli approfondimenti sulla documentazione ufficiale o attraverso gli altri miei testi dello stesso editore. In questo capitolo descriveremo tutto quello che riguarda un aspetto che è diventato sempre più importante nella realizzazione delle applicazioni mobili, ovvero i tool di sviluppo, sia per la fase di scrittura del codice, sia per quella di *build*, ovvero di creazione del file di estensione `.APK` che verrà effettivamente distribuito attraverso il *Play Store*.

Nel nostro caso abbiamo deciso di utilizzare *Android Studio* (`http://developer.android.com/sdk/index.html`) il quale, giunto ormai a una versione molto stabile (1.5), è ufficialmente supportato da Google ed è dotato di una serie di strumenti che vedremo essere molto utili nello sviluppo della nostra applicazione *ApoBus*. Si tratta di un IDE ottenuto dalla specializzazione di uno strumento analogo in ambiente Java, che si chiama *IntelliJ*. È importante sottolineare come non ci occuperemo della procedura di installazione di Android Studio nei diversi ambienti, per la quale rimandiamo alla documentazione ufficiale.

Questo secondo capitolo è di fondamentale importanza, in quanto descrive tutti gli strumenti che andremo poi ad approfondire nei successivi capitoli e che rappresentano la struttura di ogni progetto *Android*. Inizieremo creando il nostro progetto ApoBus in Android Studio, descrivendo quindi il ruolo di ogni sua singola parte. La nostra applicazione inizialmente conterrà una singola schermata molto semplice, con il messaggio *Hello World*, ma sarà comunque sufficiente per la descrizione del processo

di sviluppo che va dalla scrittura del codice fino all'esecuzione dell'applicazione in un emulatore o in un dispositivo reale.
In particolare descriveremo con sufficiente dettaglio l'utilizzo di uno strumento che ha assunto una grande importanza nel mondo Android, ovvero il tool di *build Gradle* (`https://gradle.org/`). Consigliamo al lettore di non saltare questo capitolo, in quanto sarà molto utile durante il processo di sviluppo di tutta l'applicazione. Di seguito andremo a descrivere in dettaglio le tre principali componenti di un progetto, ovvero il sorgente Java, le risorse e il file di configurazione `AndroidManifest.xml`; vedremo in dettaglio il ruolo di ciascuna di esse. Concluderemo il capitolo descrivendo il processo di esecuzione della nostra applicazione, sia attraverso un emulatore sia attraverso un dispositivo reale.

Creazione del progetto in Android Studio

Non perderemo tempo a descrivere l'installazione di Android Studio, la quale dipende dalla piattaforma utilizzata: tutte le istruzioni si trovano al *link* indicato in precedenza. Come prima cosa avviamo quindi, se non già fatto, Android Studio ottenendo quanto rappresentato nella Figura 2.1. Notiamo come nella parte sinistra vi sia un classico elenco degli ultimi progetti (che potrebbe, come nel nostro caso, essere vuoto al primo avvio); nella parte destra c'è un elenco di opzioni, tra cui la creazione di un nuovo progetto che andremo a selezionare, ottenendo la finestra rappresentata nella Figura 2.2; compiliamola con le nostre informazioni, che ci accingiamo a descrivere in dettaglio. Innanzitutto ogni applicazione ha un proprio nome, che nel nostro caso è ApoBus. Per il momento utilizziamolo per quello che è, ovvero un nome. L'informazione successiva riguarda il nome del *package* associato alla nostra applicazione. Si tratta di un concetto molto più importante di quello che sembra, in quanto ogni applicazione Android può essere associata a uno e un solo *package*, il quale dovrà rimanere lo stesso per tutta la sua vita.
Il *package* è quella caratteristica della nostra applicazione che lo identifica univocamente all'interno del *Play Store*, nel quale non ci potranno mai essere due applicazioni associate a uno stesso *package*.

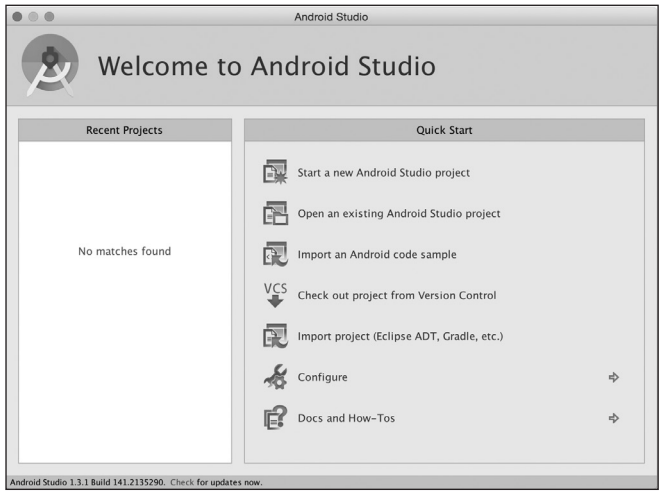

Figura 2.1 Avvio di Android Studio.

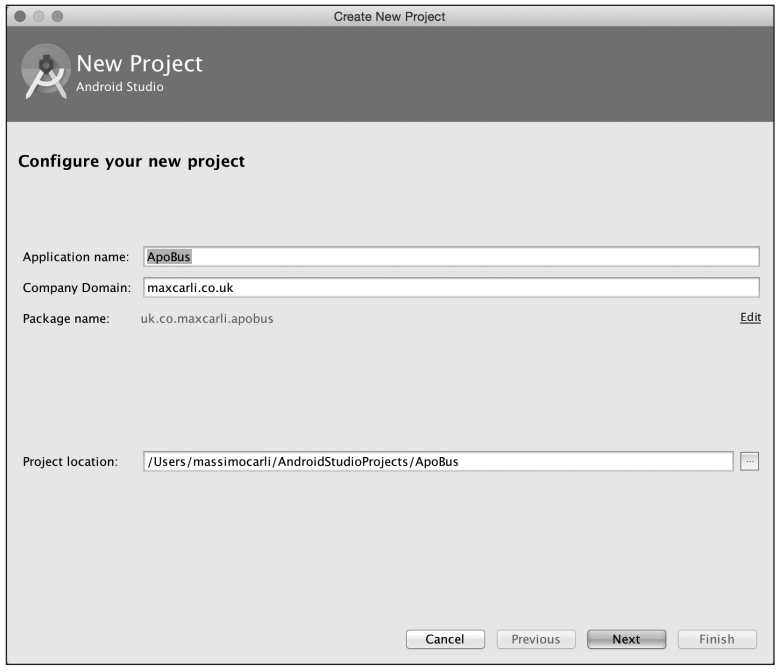

Figura 2.2 Creazione del progetto ApoBus.

Il nome del *package* dovrà seguire le convenzioni previste da Java, ovvero dovrà essere composto da parole minuscole, non riservate, separate dal punto.

> **NOTA**
>
> Quello descritto è un noto antipattern nello sviluppo Android. Una volta che si decide di realizzare una propria applicazione, è bene verificare sul *Play Store* la disponibilità del package voluto e quindi "bloccarlo", magari con un'applicazione dummy non pubblica. Questo evita spiacevoli sorprese nel momento del rilascio effettivo.

Nel caso di Android non potremo utilizzare dei *package* del tipo `com.example` o `com.android`. Le convenzioni vogliono che il *package* sia legato al dominio della nostra azienda. Se il nostro dominio è del tipo `miodominio.it` il *package* dovrà iniziare per `it.miodominio`, ovvero al contrario. Come possiamo notare, il tool ci chiede direttamente il nostro dominio, proponendo un valore per il *package* che comunque è possibile modificare selezionando l'opzione di `Edit`. L'ultimo campo riguarda la directory in cui creare il progetto; in questo caso il lettore potrà scegliere quella che più gli conviene. Selezioniamo il pulsante `Next` e arriviamo alla schermata rappresentata nella Figura 2.3 nella quale possiamo notare la possibilità di definire diversi moduli per ciascuna tipologia di applicazione. In questa fase è possibile decidere la tipologia della nostra applicazione, ovvero se debba essere eseguita in uno smartphone o tablet oppure se possa avere funzionalità accessibili attraverso dispositivi diversi come un orologio (*wearable*) oppure attraverso la TV o addirittura in auto.

Figura 2.3 Selezione della versione minima di SDK supportata.

Nel nostro caso ci accontentiamo della prima opzione, rimandando a più tardi la realizzazione di componenti aggiuntivi. Una volta selezionato il tipo di applicazione, notiamo che si ha la possibilità di selezionare la versione minima supportata dalla nostra applicazione ciascuna caratterizzata da un *API Level*. Quello di *API Level* è un altro concetto fondamentale, in quanto rappresenta una particolare versione dell'SDK della piattaforma. A ogni versione rilasciata corrisponde un *API Level* progressivo, che dovrebbe (finora è sempre stato così) garantire la retrocompatibilità. Questo significa che un'applicazione realizzata per un valore di *API Level* pari a 7 (*Eclair*) potrà essere eseguita senza problemi in dispositivi che utilizzano una versione uguale o superiore. Il valore impostato in questa fase di chiama *Mimimum SDK* e rappresenta, appunto, la versione minima di Android che la nostra applicazione dovrà supportare.

> **NOTA**
>
> La gestione di versioni diverse è, come vedremo, qualcosa di cui si deve necessariamente tenere conto e porterà all'adozione di alcuni stratagemmi, tra cui l'utilizzo di librerie di supporto. Il valore di *API Level*, insieme alle *feature* e alle dimensioni dei display supportati, rappresenta uno dei valori utilizzati dal *Play Store* per determinare se un'applicazione può essere eseguita o meno su un particolare dispositivo; in caso contrario, tale dispositivo non la visualizzerà tra quelle disponibili nello Store.

Nel nostro caso abbiamo selezionato la versione corrispondente all'*API Level 16*, che corrisponde alla prima versione della release chiamata *JellyBean*.
Il lettore potrà verificare come, dopo la selezione della minima versione supportata, il sistema visualizzi una percentuale che rappresenta, in base ai dati disponibili in quel momento, la percentuale di dispositivi in grado di eseguire la nostra applicazione. Nel

nostro caso (non visualizzato in figura) notiamo una percentuale dell'88%, che ovviamente potrebbe essere diversa per il lettore nel momento di creazione del progetto. In questa fase è sempre bene fare un compromesso tra le funzionalità che si intendono sviluppare e il lavoro che si è in grado di svolgere per garantire tutte le funzionalità su tutti i dispositivi. È indubbio, infatti, come la necessità di supportare un numero elevato di versioni differenti porti a una complicazione nello sviluppo delle funzionalità, specialmente per quello che riguarda gli aspetti grafici; la gestione della `ActionBar` è un esempio classico.

NOTA

È vero che questo testo fa riferimento alla versione di Android chiamata Marshmallow, ma se avessimo indicato questa come minima versione disponibile avremmo creato un'applicazione compatibile con meno dell'1% dei dispositivi disponibili. Vedremo più avanti come sarà possibile fare comunque riferimento a questa versione attraverso l'apposito attributo *targetSdkVersion*. Un'altra osservazione riguarda il fatto che la versione corrispondente a Marshmallow ha un *Api Level* pari a 23 e dovrebbe corrispondere alla versione 6.0 di Android, mentre nella precedente immagine vediamo ancora un riferimento a una versione 5.*X*. Ovviamente si tratta di un aspetto che potrebbe essere diverso successivamente.

Selezioniamo quindi il pulsante `Next`, arrivando alla schermata rappresentata nella Figura 2.4, nella quale ci viene data la possibilità di creare la prima schermata della nostra applicazione. Notiamo come esistano diverse possibilità, tra cui selezioniamo la più semplice, ovvero quella indicata come *Blank Activity*; in questo modo nei prossimi capitoli potremmo aggiungere i nostri componenti in modo indipendente dal particolare *IDE*, le cui funzionalità possono cambiare molto velocemente tra una versione e la successiva. Selezionando il pulsante `Next`, otteniamo la schermata rappresentata nella Figura 2.5, che ci permette di inserire le informazioni di base della nostra `Activity`, che sappiamo rappresentare una particolare schermata della nostra applicazione.
Nel prossimo capitolo vedremo come ciascuna di esse venga descritta da una particolare specializzazione, diretta o indiretta, della classe `android.app.Activity`, che potrà delegare la descrizione della UI a una particolare risorsa di *layout* descritta da un documento XML. In questa finestra non facciamo altro che specificare il nome della classe che descrive l'attività e il nome del documento XML di *layout*. In questa fase è importante ricordare che la classe è relativa al *package* che abbiamo associato al progetto. Il documento di *layout* ha solitamente un nome composto da più parole in minuscolo, separate dal simbolo *underscore* (_), ma in genere esse soddisfano le regole utilizzate per le variabili in Java. Il campo indicato come `Title` rappresenta il nome che verrà visualizzato nel menu del nostro dispositivo in corrispondenza dell'icona che andremo a selezionare per l'avvio dell'applicazione. In questa fase si tratta del valore di default, in quanto, come vedremo, tutti i valori di tipo `String` potranno essere internazionalizzati attraverso gli strumenti di gestione delle risorse e quindi resi dipendenti dalla lingua impostata nel particolare dispositivo.
L'ultimo campo permette di specificare il nome della risorsa associata al *menu* messo a disposizione da questa particolare schermata. Si tratta di un'informazione che utilizzeremo più avanti, ma che dobbiamo necessariamente definire qui, in quanto prevista come obbligatoria dal wizard. Siamo così giunti alla fine del wizard di creazione di un progetto, il quale è stato semplificato rispetto alle versioni precedenti dell'IDE rimandando a fasi successive la definizione di altri aspetti che esamineremo di volta in volta anche nel corso del nostro testo. Per il momento accontentiamoci di premere, finalmente, il pulsante `Finish`.

Figura 2.4 Selezione del componente da generare in modo automatico.

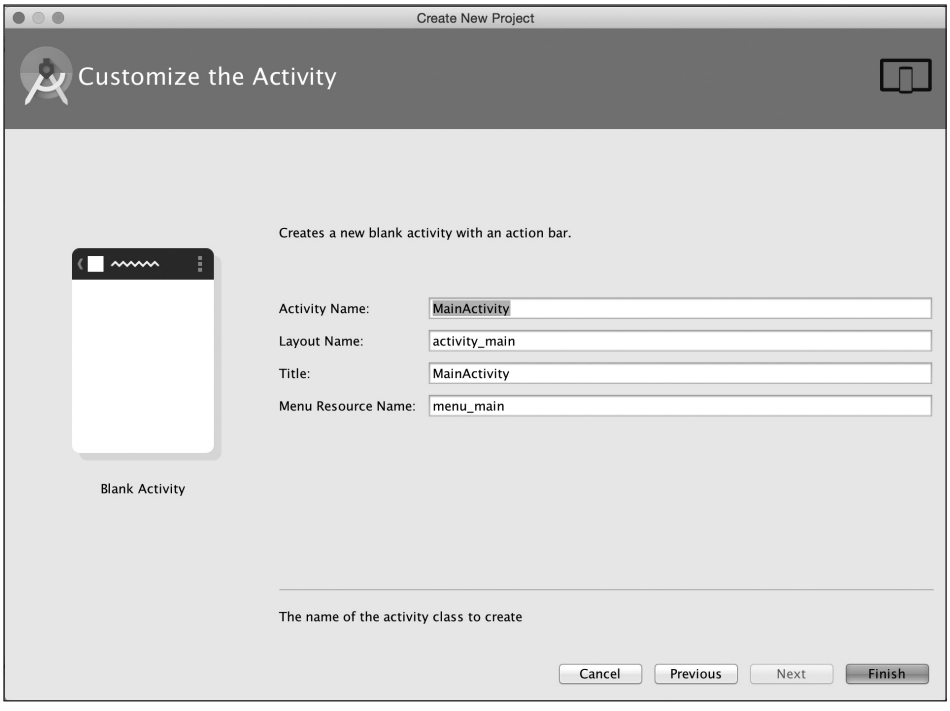

Figura 2.5 Creazione della prima Activity.

Che cosa abbiamo realizzato

A questo punto l'IDE inizierà a caricare alcune librerie di supporto relative a *Gradle*, il nuovo sistema di building utilizzato per Android in questa versione adattata di *IntelliJ* chiamata Android Studio. Al termine del caricamento otteniamo una schermata vuota, nella quale possiamo comunque notare alcuni pulsanti lungo il bordo sinistro e inferiore. Tali pulsanti permettono l'attivazione di alcune parti dell'editor. Per esempio, selezionando il pulsante indicato come *1:Project* nella Figura 2.6 si ha la visualizzazione della struttura del progetto rappresentato nella Figura 2.7.

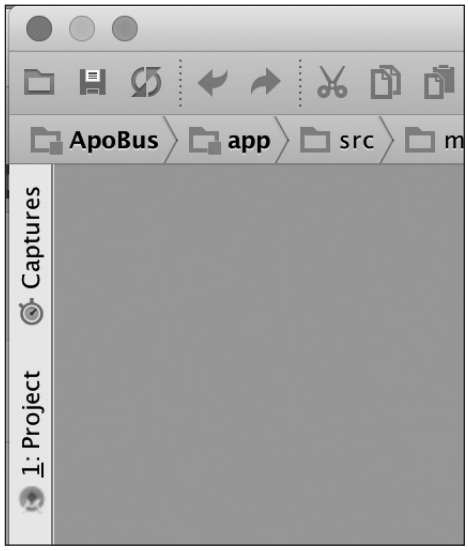

Figura 2.6 Pulsanti per l'attivazione delle parti dell'editor.

Facendo attenzione notiamo come ciascuna parte dell'editor sia caratterizzata da un nome e da un numero, che nella figura è sottolineato.
Si tratta del valore relativo alla scorciatoia che permette l'attivazione o meno del corrispondente elemento. Nel caso della struttura del progetto, l'attivazione sarà possibile attraverso la selezione del pulsante con il mouse o premendo *Alt+1* su Windows o *Cmd+1* su Mac.
In alto a sinistra notiamo la presenza del nome del progetto, che è la radice di una struttura ad albero che ci permetterà di raggiungere ogni suo elemento attraverso un menu che si chiama *Android*. È bene precisare, come vedremo meglio successivamente, come si tratti di una struttura logica, che quindi non corrisponde a un'analoga struttura a directory, alla quale è possibile accedere selezionando l'opzione *Project* indicata nella Figura 2.8. Noi utilizzeremo quella indicata come *Android*, ma ovviamente il lettore potrà utilizzare la vista che più gli aggrada. La seconda modalità rappresentata nella Figura 2.8 ci permetterà comunque di dare qualche informazione in relazione alla struttura fisica delle directory del progetto.

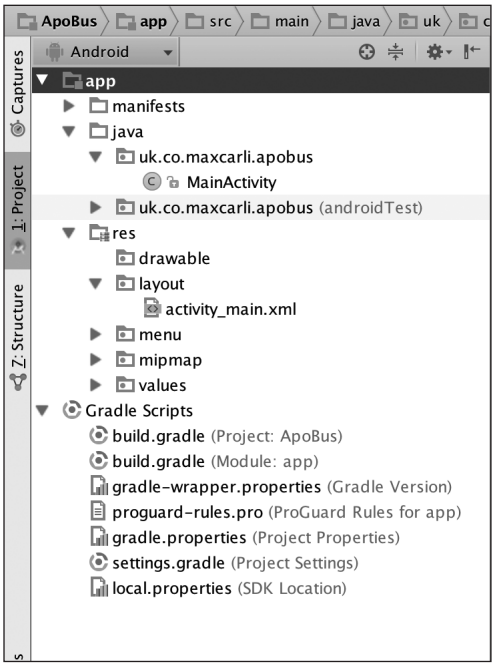

Figura 2.7 Struttura del progetto in Android Studio nella modalità Android.

A questo punto è di fondamentale importanza capire quali sono le parti del progetto, sia per quello che riguarda il codice sia per quello che riguarda la configurazione e *build* del progetto stesso. Innanzitutto notiamo come i file siano divisi in due gruppi distinti. Il primo si chiama *app* (selezionato nella Figura 2.7) e contiene tutti i file che andremo a creare ed editare per lo sviluppo vero e proprio. Al suo interno ci sono tre parti fondamentali, che impareremo a gestire nel dettaglio nei prossimi capitoli. La prima è rappresentata da una cartella che si chiama `manifests` e che contiene il file di configurazione della nostra applicazione, che si chiama `AndroidManifest.xml`. Come vedremo si tratta di un file che contiene alcune delle informazioni utilizzate in fase di installazione dell'applicazione, come l'elenco dei vari componenti, i permessi e così via. Per il momento consideriamolo un documento XML che descrive la nostra applicazione al dispositivo nel quale verrà installata. Un aspetto che potrebbe sfuggire è dato dal nome al plurale, ovvero `manifests` e non *manifest*. Questo perché, come vedremo in questo capitolo, *Gradle* ci permette di creare diverse versioni della nostra applicazione, per ciascuna delle quali sarà possibile definire diversi valori e quindi generare diversi file di configurazione `AndroidManifest.xml` che andranno tutti nella stessa cartella, che ricordiamo essere logica.

NOTA

Più file di nome *AndroidManifest.xml* non potrebbero comunque essere contenuti nella stessa cartella, in quanto file con lo stesso nome.

Come possiamo notare nella Figura 2.8 il file di configurazione `AndroidManifest.xml` è contenuto all'interno della cartella `main`, che conterrà anche i file relativi al progetto vero

e proprio.Vedremo più avanti che cosa succede nel caso in cui creassimo un file analogo nella cartella relativa ai test, ovvero quella che si chiama `androidTest`.

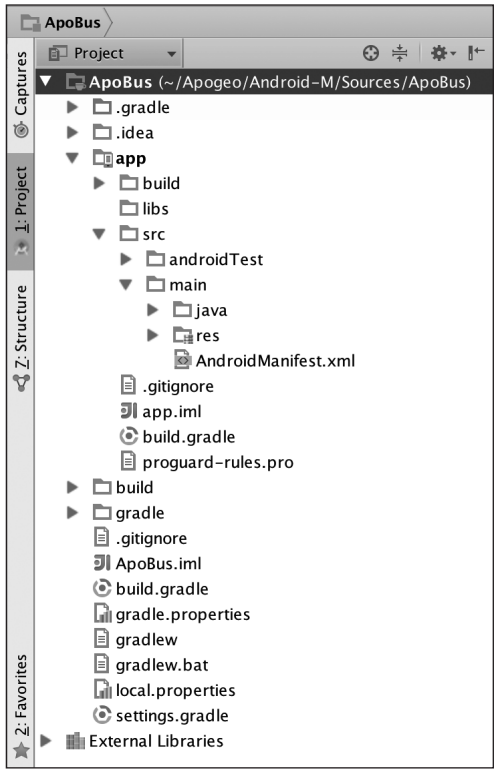

Figura 2.8 Struttura del progetto in Android Studio nella modalità Project.

Tornando alla Figura 2.7 notiamo la presenza di una cartella di nome `java` e una di nome `res`. La prima contiene tutti i sorgenti Java della nostra applicazione, mentre la seconda contiene le risorse, che impareremo a gestire fin da questo capitolo. Un aspetto interessante riguarda la modalità con cui questi file sono organizzati. Se osserviamo la parte Java notiamo come lo stesso *package* dell'applicazione sia presente due volte, anche se nella seconda è accompagnato da una label in grigio con il nome `androidTest`. Se andiamo a vedere la Figura 2.8 notiamo che `androidTest` è il nome della cartella associata ai test, la quale potrà contenere dei file per il test delle classi del *package* principale e di altri che si creeranno di volta in volta. In sintesi, la cartella `java` contiene tutti i sorgenti dell'applicazione organizzati per particolare contesto. Lo stesso vale per le risorse contenute nella cartella logica `res`. Anche qui le risorse con lo stesso nome, ma associate a contesti diversi, vengono raggruppate. Infatti, se apriamo la cartella `mipmap` notiamo quanto rappresentato nella Figura 2.9, ovvero presenza di file con lo stesso nome associati a contesti che in questo caso rappresentano, come vedremo, la densità dei display dei dispositivi che andranno a eseguire la nostra applicazione.

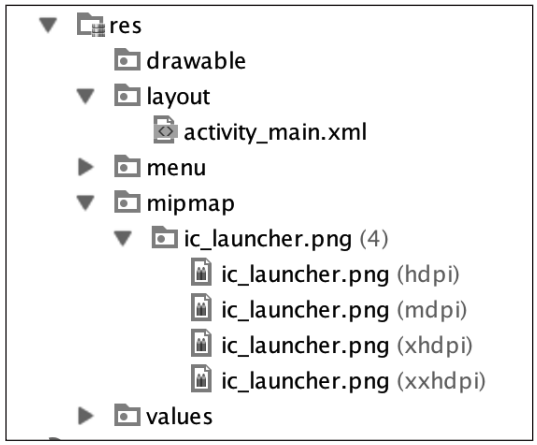

Figura 2.9 Organizzazione delle risorse nella vista Android.

È bene sottolineare come si sia parlato, per il momento, di contesti che abbiamo visto poter essere risoluzioni dei display oppure particolari versioni della nostra applicazione. Il lettore non si spaventi, in quanto tutto sarà più chiaro quando parleremo di *Gradle* e di gestione di risorse nel prosieguo di questo capitolo e nei successivi capitoli.
Dopo la sezione indicata come *app* notiamo la presenza di una cartella di nome `Gradle Scripts`, la quale contiene alcuni strumenti che sono divenuti fondamentali nel processo di sviluppo dell'applicazione e che meritano un paragrafo a parte. Prima di questo diamo un'ultima occhiata alla Figura 2.8 e in particolare alla cartella fisica `.idea`, la quale contiene una serie di configurazioni relative al nostro progetto (Figura 2.10).

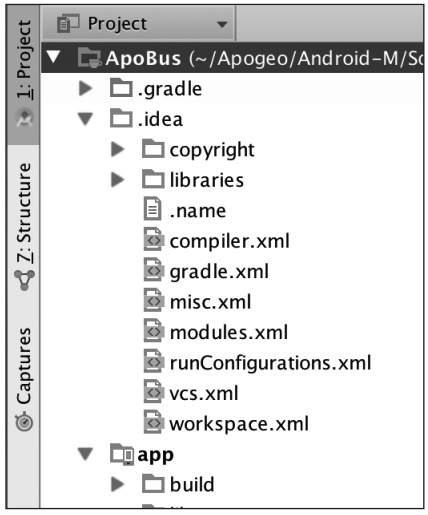

Figura 2.10 Contenuto della cartella .idea visibile in modalità Project.

Il lettore potrà verificare la presenza di alcune configurazioni relative ai vari encoding supportati, alle dipendenze, al compilatore utilizzato e così via. Si tratta di configurazione di progetto che è comunque bene includere nella parte da sottoporre a versioning. L'unica eccezione riguarda il file `workspace.xml`, il quale contiene alcune configurazioni specifiche del particolare sviluppatore, tra cui la directory di installazione, l'account per il tool di versioning e altro ancora. Si tratta, comunque, di file che non andremo a modificare, se non attraverso gli strumenti che lo stesso Android Studio ci metterà a disposizione nella parte relativa ai *settings*.

Utilizzo di Gradle

Gradle è diventato uno strumento di fondamentale importanza nello sviluppo delle applicazioni *Android*. Si tratta, in realtà, di uno strumento di *build* anche per altri tipi di applicazioni, che però Google ha personalizzato secondo le proprie esigenze attraverso la creazione di *plugin*. La caratteristica principale di *Gradle* è quella di mettere a disposizione un *Domain Specific Language* (DSL), ovvero un linguaggio specifico per un determinato dominio, che in questo caso è la gestione della fase di *build* di applicazioni con Android. Anche se non entreremo nel dettaglio, i file di configurazione di Gradle rappresentano oggetti che possono essere gestiti con un linguaggio JVM based che si chiama *Groovy*. Questo significa che chiunque può estendere e personalizzare il proprio processo di *build* estendendo i *task* offerti dai *plugin* standard. Ogni script *Gradle* è infatti equivalente al codice di un programma che viene eseguito per la fase di *build* vera e propria. Questo programma esegue sostanzialmente tre diverse fasi:

- inizializzazione;
- configurazione;
- esecuzione.

Nella prima fase Gradle leggerà tutti i file di configurazione, creando per ciascuno di questi un oggetto di tipo `Project` che, nella fase di configurazione, viene alimentato dalle informazioni relative ai vari *task* da eseguire. Per task intendiamo la compilazione, alcune verifiche sui sorgenti, esecuzione di test, creazione del file *apk* e così via. L'ultima fase è quella di esecuzione, durante la quale tutti questi task vengono effettivamente eseguiti. La fase di configurazione è importante, in quanto i *task* non sono indipendenti l'uno dall'altro, ma sono legati da vincoli di sequenzialità; non possiamo testare se prima non compiliamo e così via.
Anche per quello che riguarda *Gradle* ci aiutiamo con la struttura logica (Figura 2.7) e quella fisica (Figura 2.8) per descrivere i file creati da Android Studio in fase di creazione del progetto. Nella prima notiamo come tutti i file di *Gradle* siano contenuti in una cartella che si chiama `Gradle Scripts`, la quale contiene alcuni file che andiamo a descrivere in dettaglio, perché molto importanti durante lo sviluppo di una qualunque applicazione Android. Innanzitutto notiamo la presenza dei seguenti *tre* file:

```
settings.gradle
build.gradle
```

I file sono tre, mentre i nomi sono due, in quanto esistono due diversi file di nome `build.gradle` contenuti in cartelle diverse. Nella Figura 2.8 questo è evidente, mentre nella Figura 2.7 notiamo come i file si distinguano per la label alla loro destra, che per comodità riprendiamo nella Figura 2.11.

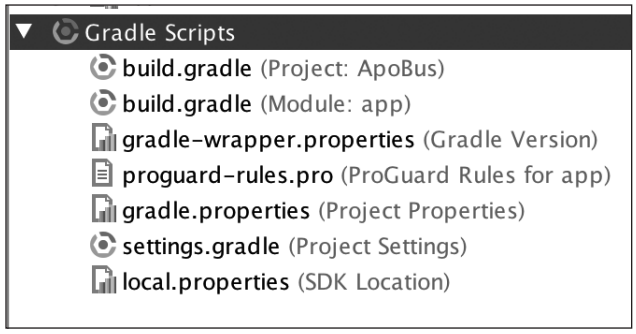

Figura 2.11 I file build.gradle nella vista Android.

Il primo ha una label che lo associa al nostro progetto ed è contenuto all'interno della cartella principale insieme al file di nome `settings.gradle`. Il secondo, invece, è associato al modulo principale che si chiama `app` ed è contenuto nella corrispondente cartella. Iniziamo quindi con la descrizione del file di nome `settings.gradle` che risulta molto semplice:

```
include ':app'
```

Da quanto detto in precedenza capiamo come questo file venga utilizzato da *Gradle* nella fase di inizializzazione per capire quali siano i progetti e i moduli da gestire e di cui leggere le configurazioni. Si tratta in sostanza di un file che permette di definire tutti i moduli della nostra applicazione.Vedremo successivamente come questo file venga modificato nel caso in cui aggiungessimo un altro modulo. È importante sottolineare come questo file non sia obbligatorio nel caso di un unico modulo, ma lo diventi nel caso in cui i moduli siano più di uno.
La cartella associata all'intero progetto contiene anche un altro file, di nome `build.gradle`, che nel nostro progetto è il seguente:

```
// Top-level build file where you can add
// configuration options common to all sub-projects/modules.
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.0'

        // NOTE: Do not place your application dependencies here;
        // they belong in the individual module build.gradle files
    }
}
```

```
allprojects {
    repositories {
        jcenter()
    }
}
```

Come dice il commento che abbiamo lasciato all'inizio del file, questo documento `build.gradle` contiene alcune configurazioni che riguardano tutti i moduli del nostro progetto. Esso è composto fondamentalmente da due parti, contenute rispettivamente all'interno dei nodi `buildscript` e `allprojects`. Il primo di questi contiene la definizione delle dipendenze, ovvero delle eventuali librerie di cui lo stesso *Gradle* necessita per il *build* della nostra applicazione. Questo significa che qui non avremo la definizione delle eventuali librerie utilizzate da ApoBus, ma delle librerie utilizzate da *Gradle* per il *build* e quindi, nel particolare, i *plugin* accennati in precedenza. Attraverso l'elemento *repositories* definiamo le sorgenti delle nostre librerie, ovvero i repository da cui ottenere le librerie stesse. *Gradle* permette in modo semplice di definire anche altre sorgenti sia remote, come `mavenCentral()`, sia locali, come `mavenLocal()`. Nel nostro esempio notiamo come i plugin per Android siano scaricati dal repository ottenuto da `jcentral()` e come lo stesso sia rappresentato dalla definizione:

```
classpath 'com.android.tools.build:gradle:1.3.0'
```

Questa definizione ci permette di dire che le classi di questo modulo saranno disponibili durante la fase di *build* del progetto e quindi verranno aggiunte al corrisponente *classpath*. Al momento la versione del *plugin* è la 1.3.0, ma ovviamente ne verranno rilasciate di successive.

> **NOTA**
>
> Per chi non conosce Java, il *classpath* rappresenta una variabile d'ambiente che contiene l'insieme delle risorse o folder all'interno delle quali andare a cercare il *bytecode* di una classe in fase di compilazione o esecuzione. Può far riferimento a un folder oppure a un file di estensione *.jar*. Nel caso si cercasse la classe *a.b.MyClass*, il compilatore o l'interprete andranno a cercare il file *MyClass.class* all'interno delle cartelle *a/b/* all'interno di ciascuna delle risorse indicate nella variabile *classpath*.

Come possiamo notare, ciascuna libreria, come vedremo più avanti per quelle specifiche della nostra applicazione, è caratterizzata da un nome del seguente tipo:

```
<package-or-company>:<name>:<version>
```

La prima parte identifica l'organizzazione o azienda che ha creato o gestisce la libreria. Solitamente utilizza un meccanismo simile a quello seguito per i *package* delle applicazioni e quindi utilizzando il dominio al contrario. Dopo i due punti (:) la seconda parte identifica il nome della libreria. In questo caso si tratta della libreria `gradle`. Le prime due parti sono quelle obbligatorie, mentre la versione è opzionale, ma molto importante.

Essa segue una convenzione che si chiama *semantic versioning* e che prevede a sua volta una struttura del seguente tipo:

```
major.minor.patch
```

Il valore di `major` identifica la versione principale e viene modificata nel caso in cui una versione non fosse più compatibile con quella precedente. Se le modifiche sono invece compatibili con le precedenti si tratterà di un aggiornamento con un valore diverso per la `minor`. Infine il campo `patch` permette di specificare l'applicazione di alcune correzioni di bug o comunque miglioramenti della versione associata ai `major` e `minor` specificati.
L'aspetto interessante del *semantic versioning*, riguarda la possibilità di fare in modo di disporre sempre dell'ultima versione disponibile, senza modificare il file di configurazione di *Gradle*. Per esempio potremmo utilizzare la seguente definizione per indicare la volontà di utilizzare sempre l'ultima versione disponibile indipendentemente dal valore di `major release`:

```
classpath 'com.android.tools.build:gradle:+'
```

È bene sottolineare come si tratti di un'operazione pericolosa, in quanto sappiamo che una particolare *major* release potrebbe essere incompatibile con quella precedente e quindi potrebbe rompere il processo di `build`. Nel caso in cui volessimo utilizzare una precisa `major` release e l'ultima `minor release`, potremmo utilizzare la seguente notazione:

```
classpath 'com.android.tools.build:gradle:1.+'
```

Se volessimo utilizzare una qualunque patch per una specifica `minor release` la notazione potrebbe essere la seguente:

```
classpath 'com.android.tools.build:gradle:1.3.+'
```

Infine possiamo addirittura indicare la volontà di utilizzare tutte le minor release di versione successiva a una indicata, semplicemente mettendo il + subito dopo la versione, come nel seguente esempio:

```
classpath 'com.android.tools.build:gradle:1.3+'
```

Se il lettore ha verificato quanto descritto in precedenza avrà sicuramente notato come Android Studio si accorga delle eventuali modifiche ai file di configurazioni, mettendo a disposizione il link rappresentato nella Figura 2.12 per l'aggiornamento effettivo della configurazione.

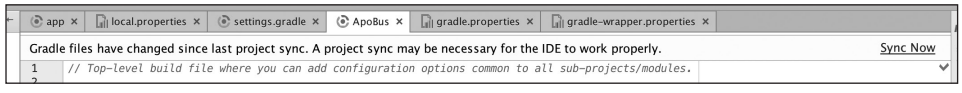

Figura 2.12 Sync Now quando si modifica un qualunque file di configurazione di Gradle.

È bene anche ricordare come questa operazione sia equivalente alla selezione del pulsante messo in evidenza nella Figura 2.13.

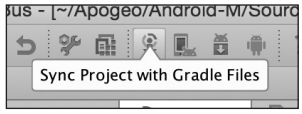

Figura 2.13 Aggiornamento dei file di configurazione di Gradle.

Tornando al nostro file di configurazione abbiamo visto come la prima parte riguardi la definizione delle dipendenze relative al tool di build stesso. Nel caso di utilizzo di altri *plugin* questo è il luogo in cui si dovranno specificare le corrispondenti dipendenze con i relativi repository.
La seconda parte è invece definita dall'elemento `allprojects`, che permette di definire tutte le informazioni relative a tutti i moduli della nostra applicazione. In questo caso viene solamente specificato il repository di riferimento, ma avremmo potuto inserire alcune delle definizioni che vedremo in dettaglio successivamente per il nostro modulo. Anche in questo caso il consiglio è quello di rendere i vari moduli il più possibile indipendenti tra loro, in modo da poterli eventualmente riciclare in altri progetti.

Il file build.gradle del modulo principale

Il terzo file di configurazione creato con il nostro progetto è quello di nome `build.gradle`, ma contenuto questa volta nella cartella associata al modulo di nome `app`, che nel nostro caso è il seguente:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.0"

    defaultConfig {
        applicationId "uk.co.maxcarli.apobus"
        minSdkVersion 16
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}
```

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:23.0.0'
}
```

La prima riga contiene la definizione dell'utilizzo del *plugin* di nome `com.android.application` il quale è disponibile in quanto contenuto nella libreria di cui abbiamo definito la dipendenza nel precedente file. Si tratta di un plugin che estende *Gradle* aggiungendo elementi e configurazioni specifici dello sviluppo Android, come quelli che vediamo all'interno dell'elemento di nome `android` il quale contiene alcune definizioni che sono fondamentali e che andremo a spiegare nel dettaglio.
Le uniche informazioni obbligatorie del modulo android sono le prime due, ovvero le seguenti:

```
compileSdkVersion 23
buildToolsVersion "23.0.0"
```

La prima (`compileSdkVersion`) permette di specificare l'API Level che andremo a considerare come effettivo per il nostro progetto. Questo significa che potranno essere utilizzati tutti gli oggetti e le definizioni previste da Android Marshmallow.

> **NOTA**
>
> Attenzione! Questo non significa che tutti i dispositivi saranno in grado di utilizzare queste nuove *feature*. È comunque responsabilità del programmatore fare in modo, come vedremo, che l'applicazione funzioni per tutti i dispositivi con Android di versione uguale o superiore a quella specificata in fase di creazione del progetto, che nel nostro caso è la 16.

La seconda (`buildToolsVersion`) permette invece di impostare la versione degli strumenti che la piattaforma Android ci mette a disposizione per le principali operazioni di *build*, come `aapt`, `zipalign`, `dix` e altri ancora. Si tratta di strumenti che vedremo, più avanti nel capitolo, come aggiornare attraverso l'*SDK Manager*.

> **NOTA**
>
> Attenzione a non confondere la responsabilità di *Gradle* con quella degli strumenti della piattaforma Android. Per esempio Gradle non creerà il file *apk* per l'installazione, ma permetterà la configurazione e l'utilizzo del corrispondente strumento della piattaforma Android.

Di seguito vi è un componente che si chiama `defaultConfig` e che contiene le configurazioni di default della nostra applicazione le quali si andranno a fondere con quelle definite nel file `AndroidManifest.xml` che, ricordiamo, descrive l'applicazione al dispositivo nel quale viene installata. Si tratta di un componente molto importante che quindi riprendiamo qui di seguito:

```
defaultConfig {
    applicationId "uk.co.maxcarli.apobus"
    minSdkVersion 16
    targetSdkVersion 23
```

```
    versionCode 1
    versionName "1.0"
}
```

La prima informazione si chiama `applicationId` e contiene, di default, il nome del *package* associato alla nostra applicazione. A questo punto è comunque importante fare alcune precisazioni. Abbiamo già detto che ogni applicazione è associata a un *package* che identifica un particolare utente del sistema Linux sottostante da cui si ereditano le garanzie di sicurezza. Come un utente non può accedere alle informazioni di un altro, così un'applicazione non può accedere alle risorse di un'altra, a meno che questa non lo permetta in modo esplicito. In ogni caso, applicazioni diverse sono associate a *package* diversi. Come vedremo nella seconda parte del capitolo, il *package* è importante, anche perché è quello a cui appartengono le classi che vengono generate in modo automatico in fase di *build*, come quelle che ci permetteranno di referenziare le varie risorse.

NOTA

Come vedremo tra poco, per ciascuna risorsa viene definita una costante di una particolare classe interna della classe *R*. Per esempio una risorsa di *layout* potrà essere associata alla costante *R.layout.my_layout*. Il package dell'applicazione sarà anche il *package* della classe *R*.

Ecco che l'informazione di `applicationId` non rappresenta il *package* dell'applicazione, ma quel valore che la identificherà nel *Play Store* al momento della pubblicazione. Ma come mai questa distinzione? Il motivo è legato al concetto di *build variant*, che ci permetterà di creare più versioni della stessa applicazione e quindi, per esempio, una versione free e una a pagamento oppure versioni che si differenziano per alcune parti, come una diversa libreria nelle dipendenze o una diversa icona. Utilizzando per ciascuna *build variant* un valore diverso per l'`applicationId` potremo fare in modo di utilizzare *package* diversi per l'identificazione dell'applicazione, mantenendo però lo stesso *package* nella generazione automatica delle risorse e quindi della classse `R`. Una modifica anche nel *package* dell'applicazione avrebbe infatti portato alla creazione di duplicazioni di difficile gestione. Ecco che l'informazione relativa all'`applicationId` ci permetterà, per esempio, di avere contemporaneamente sul nostro dispositivo versioni diverse della stessa applicazione.
Le informazioni che seguono sono molto semplici e anch'esse andranno a sovrapporsi alle corrispondenti definizioni nel file di configurazione `AndroidManifest.xml`. Attraverso `minSdkVersion` andiamo a specificare la versione minima di Android (*Api Level*) che un dispositivo dovrà supportare per poter eseguire la nostra applicazione. È un'informazione utilizzata dal *Play Store* per fare in modo che dispositivi di versioni precedenti non vedano neppure l'applicazione tra quelle disponibili. Attraverso il `targetSdkVersion` indichiamo invece la versione con cui la nostra applicazione è stata testata e sulla quale confidiamo che funzioni. Da notare come il concetto sia diverso da quello relativo alla variabile `compileSdkVersion` vista in precedenza. Infine le proprietà `versionCode` e `versionName` permettono di indicare la versione dell'applicazione, rispettivamente, attraverso un valore numerico e un nome più semplice da leggere. Il primo è importante, in quanto un'applicazione non potrà essere aggiornata sul *Play Store* da un'altra versione con un `versionCode` inferiore.

Il concetto di Build Type e Build Variant

Come abbiamo detto, le informazioni del modulo `defaultConfig` sono quelle di *default* per i vari *build type*. Ma cosa sono, più precisamente, i *build type*? Come dice il nome stesso si tratta di modi diversi di eseguire l'operazione di build della nostra applicazione. Per ciascun modulo, Gradle crea un particolare build type che si chiama `debug` e che contiene alcune impostazioni che sono utili in fase di sviluppo. Altre sono invece definite nel corrispondente file `build.gradle` attraverso un elemento che si chiama `buildTypes`, che nel nostro esempio è il seguente:

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
    }
}
```

Esso contiene la definizione di tutti i `build type` in aggiunta a quello di default che si chiama `debug`. Nel caso del progetto creato da Android Studio notiamo la presenza di un build type di nome release che contiene l'impostazione di alcune informazioni che sono utili per la versione dell'applicazione da pubblicare sul *Play Store*. In particolare l'attributo `minifyEnabled` a `false` permette di disabilitare l'eliminazione delle risorse non utilizzate, mentre l'attributo `proguardFiles` permette di impostare il file di `Proguard`, ovvero del tool di offuscamento e ottimizzazione del codice che vedremo più avanti. In questa fase non ci interessa tanto la singola proprietà, ma il fatto che sia possibile creare diverse modalità con cui possa essere eseguito il build della nostra applicazione.
Di default, abbiamo quindi due modalità; quella di debug, creata in modo automatico da *Gradle*, e quella di release, definita da Android Studio nel modo che abbiamo visto. Ma nel concreto come facciamo a gestire queste informazioni e queste diverse modalità di `build`? Android Studio ci viene in aiuto mettendo a disposizione alcune viste cui possiamo accedere attraverso l'opzione sulla sinistra in basso che si chiama `Build Variants` e che possiamo vedere nella Figura 2.14. Notiamo infatti come sia presente una tendina per il nostro unico modulo di nome `app`, che contiene appunto i nomi `debug` e `release`. Selezionando uno di questi e quindi eseguendo il `build` dell'applicazione verrà utilizzata la configurazione corrispondente.
Quello del `build type` è uno strumento molto potente, in quanto consente di gestire configurazioni diverse o librerie diverse. Per vedere come, aggiungiamo un nuovo `build type` che si chiama `perf`, perché utilizza alcune configurazioni relative, per esempio, ad alcuni test di performance che si intendono realizzare. Questo build type potrebbe, per esempio, utilizzare un diverso server per l'accesso ai dati, una chiave diversa per l'utilizzo delle mappe e una libreria per la misurazione delle performance che non si vuole utilizzare per la versione pubblicata nel *Play Store*. Andiamo quindi ad aggiungere all'interno del nostro elemento `buildTypes` il seguente:

```
buildTypes {
    release {
        minifyEnabled false
```

```
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
    }
    perf.initWith(buildTypes.debug)
    perf {
        applicationIdSuffix ".pref"
        versionNameSuffix "-perf"
        buildConfigField "String", "PERF_URL", "\"https://myperfserver/data\""
    }
}
```

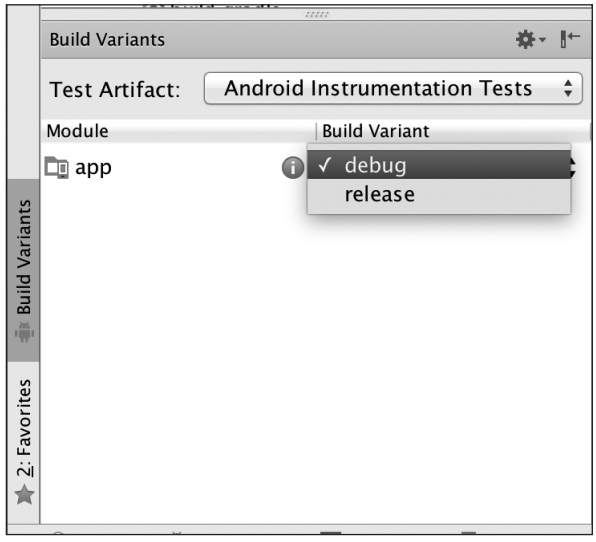

Figura 2.14 La visualizzazione delle Build Variants.

Notiamo la presenza di un elemento di nome `perf` che contiene la definizione di tre proprietà molto comode. Attraverso la proprietà `applicationIdSuffix` è possibile infatti decidere il suffisso da aggiungere all'`applicationId` definito in precedenza. Quando eseguiamo il build secondo questa configurazione, verrà creata un'applicazione che ha come `id` il *package*

```
uk.co.maxcarli.apobus.perf
```

Come abbiamo detto prima è bene ricordare come il *package* delle classi e delle risorse create sia comunque quello associato all'applicazione principale. Attraverso l'attributo `versionNameSuffix` andiamo invece a modificare il `versionName` aggiungendo il suffisso `-perf`. In entrambi i casi sono due informazioni che in fase di build andranno a sostituire le corrispondenti definite all'interno del file di configurazione `AndroidManifest.xml`. Molto interessante è infine l'attributo `buildConfigField`, che ci permette di definire il valore di una costante della classe `BuildConfig` che potremo utilizzare all'interno del codice dell'applicazione, in quanto generata in modo automatico. Se ora andiamo a vedere la finestra relativa ai *build variants* notiamo quanto rappresentato nella Figura 2.15, ovvero la presenza di un nuovo valore che si chiama, appunto, `perf`.

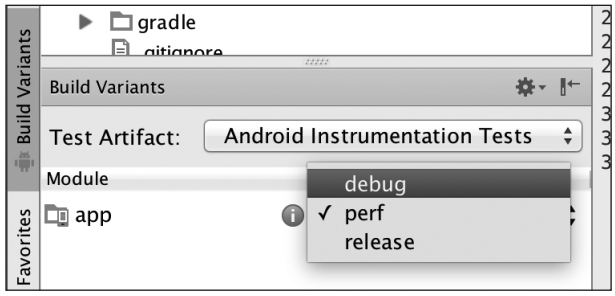

Figura 2.15 Il nuovo Build Variant di nome perf.

Il concetto di `build type` è molto di più di quanto esposto finora. Se quello definito utilizza delle librerie di misurazione delle performance significa che dovrà avere delle dipendenze che l'applicazione pubblicata non utilizza e quindi anche delle classi e risorse che non saranno utili all'applicazione nel *Play Store*. Serve quindi un meccanismo che ci permetta di definire del codice e delle risorse che sono specifiche del particolare `build type`. A ciascun `build type` può essere associato un `folder` con lo stesso nome all'interno della cartella di nome `src`. Questo folder potrà quindi contenere i sorgenti, le risorse e il file di configurazione `AndroidManifest.xml` specifico del `build type`. Supponiamo quindi di voler aggiungere una classe di nome `Performance.java`, una nuova risorsa di tipo `string` e quindi il file di configurazione specifico del nostro `build type`.

> **NOTA**
>
> Vedremo più avanti come gestire tutte le risorse supportate dalla piattaforma. Per il momento pensiamo a una risorsa di tipo string come a una label cui è possibile accedere attraverso un'opportuna costante della classe *R* generata in modo automatico.

Il nostro *file system* di partenza è quello rappresentato nella Figura 2.16, nella quale notiamo la presenza di una cartella di nome `main` che contiene i folder `java` e `res`, che contengono rispettivamente i sorgenti java e le risorse principali. Notiamo anche la presenza del file `AndroidManifest.xml`.
Se vogliamo specializzare il tutto per il nostro `build type` andiamo a creare la stessa struttura all'interno di una nuova cartella di nome `perf` come indicato nella Figura 2.17. Come possiamo notare, abbiamo creato una cartella di nome `java` che contiene lo stesso *package* dell'applicazione con una nuova classe contenuta nel file di nome `Performance.java`. È bene sottolineare come i file in queste cartelle debbano essere pensati relativamente a quelli principali contenuti nel folder che si chiama `main`. Ciascun *build type* utilizza e vede le stesse classi contenute nel folder `main` e può aggiungerne di proprie, ma non può modificarle. Questo significa che ogni classe specifica di un *build type* non può essere una versione modificata di una esistente nel `main`. Diverso è il discorso per le risorse e il file di configurazione `AndroidManifest.xml` le quali, quando possibile, vengono semplicemente fuse e quindi ne viene fatto il *merge*. Questo è possibile per le risorse di tipo valore e per il file di configurazione `AndroidManifest.xml`, ma non per altri tipi di risorse come quelle di layout e le immagini.

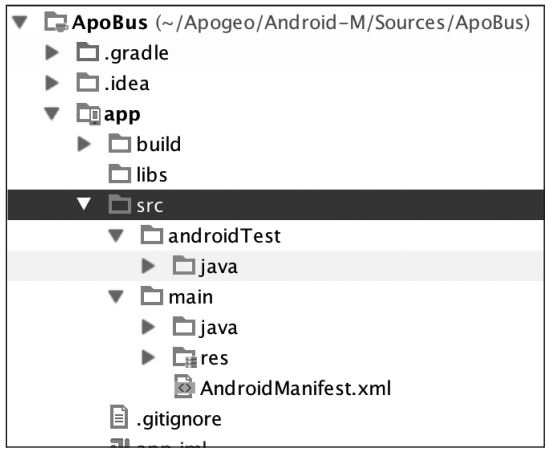

Figura 2.16 Folder associati ai vari build type.

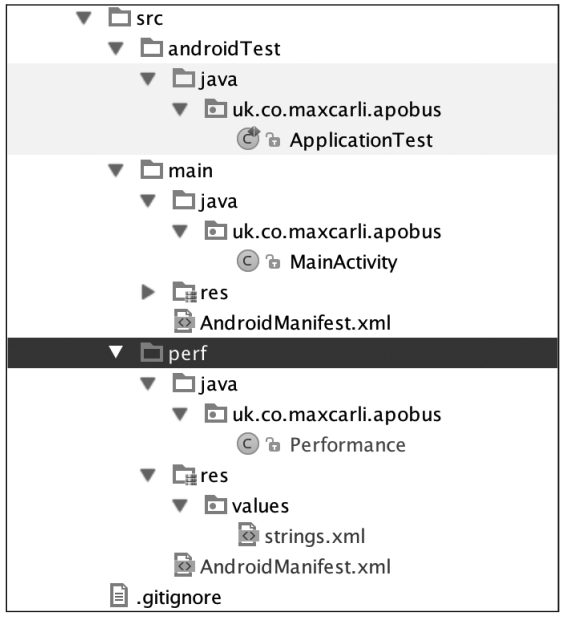

Figura 2.17 Struttura dei folder per il build type di nome perf.

Per dare un'anteprima di cosa significhi diciamo che se in `main` definiamo le seguenti risorse:

```
<resources>
    <string name="app_name">ApoBus</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
</resources>
```

e in perf solamente la seguente:

```
<resources>
    <string name="app_name">ApoBus Perf</string>
</resources>
```

è come se avessimo definito un file delle risorse fatto nel seguente modo, in cui il valore associato alla costante app_name ha preso il valore definito nel *build type* di nome perf mantenendo le altre.

```
<resources>
    <string name="app_name">ApoBus Perf</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
</resources>
```

Se andiamo a vedere il file AndroidManifest.xml notiamo come sia vuoto; andremo infatti a specificare solamente le eventuali differenze rispetto a quello principale nella cartella main. Nel codice specifico del nostro build type possiamo poi accedere a una nuova costante BuildConfig.PERF_URL che abbiamo utilizzato all'interno della classe Performance per dimostrarne l'esistenza.

```
public class Performance {

    public void method(){
        System.out.println("URL " + BuildConfig.PERF_URL);
    }
}
```

La creazione di questa costante è conseguenza della definizione che abbiamo fatto nel nostro *build type*. Per questo motivo si tratta di una costante non presente negli altri *build type* se non definita in modo esplicito.
Il lettore attento avrà notato come si sia parlato sia di *build variants* sia di *build type*. Questo perché esiste in realtà un altro concetto che si chiama *build flavor* e un *build variants* è la combinazione di un *build type* con un *build flavor*. Finora abbiamo definito tre build type (build, release e perf), ma nessun *build flavor.* A dire il vero si tratta di una differenza piuttosto sottile. Potremmo dire che *build type* e *build flavor* permettono la creazione di versioni diverse di un'applicazione secondo due dimensioni ortogonali. Mentre un *build type* permette di definire diverse configurazioni di un'applicazione, un *build flavor* permette di definire applicazioni diverse che hanno in comune una stessa base di risorse e codice. Diversi *flavor* di una stessa applicazione sono, per esempio, la versione a pagamento e quella che contiene dei banner. *Flavor* diversi si possono per esempio distinguere per l'icona nel caso in cui si riutilizzasse lo stesso codice per aziende diverse. Nel caso della nostra applicazione ApoBus, flavor diversi potrebbero far riferimento a città diverse che quindi utilizzano immagini, mappe e server diversi. Se per esempio volessimo creare due flavor per *Londra* e *Barcellona* alla fine avremmo 3×2, ovvero sei differenti *build variants*. Creare quindi questi due flavor è molto semplice, in quanto si utilizza l'elemento di nome productFlavors:

```
android {
    - - -
    defaultConfig {
        applicationId "uk.co.maxcarli.apobus"
        minSdkVersion 16
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        - - -
    }
    productFlavors {

        london {
            applicationId "uk.co.maxcarli.apobus.london"
            versionCode 2
            versionName "2.0"
        }

        barcelona {
            applicationId "uk.co.maxcarli.apobus.barcelona"
            versionCode 1
            versionName "1.0"
        }
    }
}
```

Un aspetto molto importante riguarda il fatto che le proprietà di un *build flavor* non sono le stesse di un *build type*, ma quelle che abbiamo utilizzato per il `defaultConfig`. Ecco che possiamo definire `applicationId` e identificatori diversi per ciascuno di essi. Se ora andiamo a vedere tutti i *build variants* disponibili notiamo quanto rappresentato nella Figura 2.18, ovvero tutte le possibili combinazioni di *type* e *flavor*.

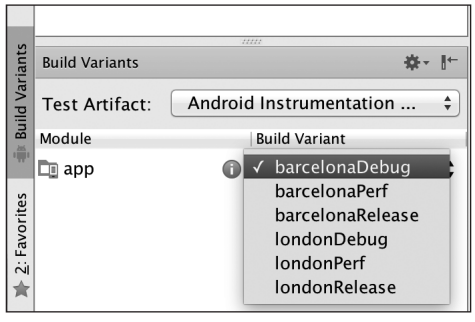

Figura 2.18 Combinazioni di build type e build variant.

Ora con `londonDebug` faremo riferimento alla versione di debug per *Londra*, mentre con `barcelonaPerf` avremo la versione di misura delle performance per *Barcellona*. Per ciascuna di queste valgono le considerazioni che abbiamo fatto in relazione alla gestione dei sorgenti e delle eventuali proprietà della classe `BuildConfig` generata automaticamente.

Se quindi volessimo definire delle configurazioni particolari per il *build variant* di nome `barcelonaRelease` basterà creare una definizione con quel nome del tipo:

```
barcelonaRelease {
    - - -
}
```

Si tratta di informazioni che ci verranno molto utili in seguito, quando andremo a realizzare la nostra applicazione nello specifico.

Gestiamo le dipendenze

Finora abbiamo visto come specializzare alcune versioni dell'applicazione in termini di risorse o configurazioni. Gradle è molto utile anche nella gestione delle dipendenze, ovvero nella dichiarazione e gestione delle librerie di cui, nelle diverse fasi di sviluppo, la nostra applicazione necessita per le proprie funzionalità. Come vedremo, esistono diverse librerie relative ad alcune funzionalità particolari come i *Google Play services*, oppure librerie che utilizziamo per il test dell'applicazione oppure quelle cui abbiamo accennato in precedenza nell'esempio, che permettono di eseguire alcune misurazioni di performance. Sia il progetto principale sia ciascuna *build variant* può definire le proprie dipendenze attraverso una definizione del seguente tipo:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:23.0.0'
}
```

Nello specifico stiamo indicando come la nostra applicazione utilizzi le librerie contenute nella cartella `/libs` del progetto oltre a quella di nome `appcompat-v7`, che ci permetterà di risolvere alcuni problemi relativi al fatto che alcune funzioni non sono disponibili in tutte le versioni di *Android* che abbiamo deciso di supportare; un esempio è dato dalla `ActionBar` o dalla `Toolbar`.

> **NOTA**
>
> La presenza della libreria di supporto è dovuta al fatto che si sia deciso di supportare le versioni della piattaforma dall'*API Level 16* in poi. Se avessimo messo come versione mimima lo stesso *Android M*, questa libreria non sarebbe stata inclusa.

In questa fase non ci dilunghiamo sulla particolare libreria, ma piuttosto sul concetto di *dependency configuration*, ovvero della modalità con cui la stessa dipendenza viene utilizzata nel progetto che notiamo essere utilizzata a fianco della dipendenza stessa. Al momento sono disponibili le seguenti configurazioni:

- `compile`;
- `apk`;
- `provided`;
- `testCompile`;
- `androidTestCompile`.

La prima, `compile`, è quella di default e permette di indicare come la libreria non venga solamente aggiunta al `classpath` che abbiamo descritto in precedenza, ma anche aggiunta all'*APK*. Si tratta quindi di classi e risorse che la nostra applicazione utilizza, ma che non sono già disponibili nei vari dispositivi. Attraverso il valore `apk` stiamo indicando come la nostra libreria non debba essa aggiunta al `classpath`, ma semplicemente aggiunta al corrispondente *apk*. Questo è il caso di librerie che utilizzano per esempio tecniche di *introspection* che permettono di creare istanze di una classe conoscendone il nome, che quindi non viene utilizzato in fase di compilazione. Attraverso il valore `provided` andiamo invece ad aggiungere la libreria al classpath senza poi inserirla nel corrispondente *apk*. Questo è il caso in cui si crei un'applicazione per un particoare dispositivo dotato di classi custom che la nostra applicazione troverà disponibili una volta installata e che quindi non è necessario inserire nell'*apk*. Le ultime due *dependency configuration* ci permetteranno di definire librerie da utilizzare in fase di test dell'applicazione. È infine bene sottolineare come tutte le dipendenze possano essere specializzate per ciascuno dei *build variants* creati.

I task e Wrapper

Nei precedenti paragrafi abbiamo visto come sia possibile definire diverse modalità di *build* dell'applicazione attraverso la definizione di *build type* e *build flavour*, le cui combinazioni portano alla definizione di *build variant*. Ma che cos'è, in pratica, una modalità di build? In che cosa si differenziano l'una dall'altra? Come sappiamo lo sviluppo di un programma Java presuppone la creazione del codice sorgente, la compilazione e quindi l'impacchettamento in un file jar. Si tratta in sostanza di una successione di passi (*task*) che un particolare tool (in questo caso *Gradle*) esegue per ottenere il risultato finale.

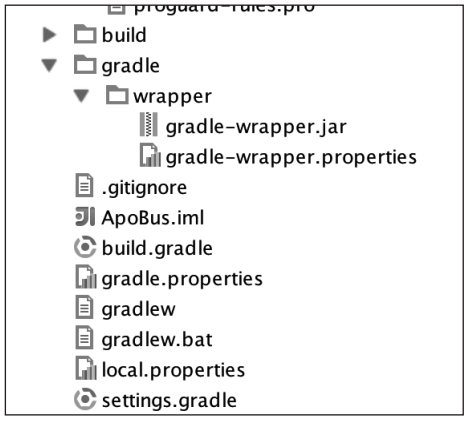

Figura 2.19 Il Wrapper per Gradle.

Come abbiamo detto in precedenza, Google ha scelto Gradle proprio per avere la possibilità di estenderlo attraverso la creazione di opportuni plugin, i quali non fanno altro che aggiungere alcuni task specifici di Android a quelli che sono forniti dalla piattaforma per Java. Prima di vedere quali siano i task disponibili e quali siano le modalità per la loro esecuzione diamo un breve cenno al *wrapper*, un'utility che ci permetterà di avere

sempre una versione di *Gradle* aggiornata. La presenza di questo strumento è visibile nella Figura 2.19, ovvero nella vista `Project` del nostro progetto in Android Studio. Nella figura possiamo notare due delle tre componenti del `Wrapper`, ovvero le corrispondenti classi nel file `.jar` e un file di configurazione di estensione `.properties`. La terza componente è un tool che possiamo utilizzare da riga di comando e che viene utilizzato da Android Studio stesso per l'esecuzione dei vari task. Si tratta di un tool che è disponibile per le varie piattaforme e ha quindi estensione diversa a seconda che sia per *Windows* (`.bat`) o *Mac* (`.sh`). Non entriamo nel dettaglio di questo strumento che fortunatamente Android Studio ci permette di tenere aggiornato attraverso opportuni messaggi e notifiche. Quello che ci interessa è la possibilità di accedere ai vari *task* che compongono il processo di build. Per fare questo esistono diverse modalità. La prima che osserviamo è quella da riga di comando.

> **NOTA**
>
> L'utilizzo di questi tool da riga di comando è di fondamentale importanza, in quanto ne permette l'uso all'interno di strumenti di continuous integration come Jenkins (`https://jenkins-ci.org/`). Si tratta di strumenti che non necessitano di un IDE come Android Studio e che talvolta non dispongono neppure di un'interfaccia grafica, ma solo di un terminale.

Nella parte inferiore dell'IDE notiamo la presenza di una label che si chiama `terminal`, che andiamo a selezionare ottenendo quanto rappresentato nella Figura 2.20.

Figura 2.20 Accesso al terminale attraverso Android Studio.

Ovviamente il lettore potrà avere un path diverso per l'applicazione oltre che accedere alla stessa cartella attraverso il proprio terminale fuori da Android Studio. A questo punto decidiamo di utilizzare il wrapper attraverso il seguente comando:

```
./gradlew -v
```

il quale ha il compito di mostrare la versione in uso. Se stiamo eseguendo il comando per la prima volta, potremmo avere la sorpresa del download della versione di gradle corrispondente e quindi della visualizzazione delle informazioni richieste. Si tratta di una prima dimostrazione di come il `Wrapper` ci permetta di gestire le versioni di Gradle.

NOTA

Il lettore curioso potrà verificare come la versione di Gradle scaricata sia quella specificata nel file di nome *gradle-wrapper.properties* visualizzato nella Figura 2.19.

Se tutto è andato per il verso giusto, noteremo la visualizzazione di informazioni simili alle seguenti:

```
Build time:    2015-05-05 08:09:24 UTC
Build number:  none
Revision:      xxxxxxxxxxxxxxxxxxxxx643f1e2d190f2c943c

Groovy:        2.3.10
Ant:           Apache Ant(TM) version 1.9.4 compiled on April 29 2014
JVM:           1.8.0_05 (Oracle Corporation 25.5-b02)
OS:            Mac OS X 10.10.4 x86_64
```

Ora vogliamo però visualizzare tutti i task disponibili. Per fare questo è sufficiente utilizzare il seguente comando:

```
./gradlew tasks
```

Lo eseguiamo dopo aver commentato le nostre definizioni di *build variants* fatte in precedenza per un motivo che sarà presto chiaro. Anche in questo caso Gradle provvederà a scaricare tutte le classi definite nelle varie dipendenze in un repository locale da utilizzare nelle esecuzioni successive e quindi visualizzerà l'elenco richiesto, che è il seguente:

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Android tasks
-------------
androidDependencies - Displays the Android dependencies of the project.
signingReport - Displays the signing info for each variant.
sourceSets - Prints out all the source sets defined in this project.

Build tasks
-----------
assemble - Assembles all variants of all applications and secondary packages.
assembleAndroidTest - Assembles all the Test applications.
assembleDebug - Assembles all Debug builds.
assembleRelease - Assembles all Release builds.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
compileDebugAndroidTestSources
compileDebugSources
compileDebugUnitTestSources
compileReleaseSources
compileReleaseUnitTestSources
```

```
mockableAndroidJar - Creates a version of android.jar that's suitable for unit tests.

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
components - Displays the components produced by root project 'ApoBus'. [incubating]
dependencies - Displays all dependencies declared in root project 'ApoBus'.
dependencyInsight - Displays the insight into a specific dependency in root project 'ApoBus'.
help - Displays a help message.
model - Displays the configuration model of root project 'ApoBus'. [incubating]
projects - Displays the sub-projects of root project 'ApoBus'.
properties - Displays the properties of root project 'ApoBus'.
tasks - Displays the tasks runnable from root project 'ApoBus' (some of the displayed tasks
may belong to subprojects).

Install tasks
-------------
installDebug - Installs the Debug build.
installDebugAndroidTest - Installs the android (on device) tests for the Debug build.
uninstallAll - Uninstall all applications.
uninstallDebug - Uninstalls the Debug build.
uninstallDebugAndroidTest - Uninstalls the android (on device) tests for the Debug build.
uninstallRelease - Uninstalls the Release build.

Verification tasks
------------------
check - Runs all checks.
clean - Deletes the build directory.
connectedAndroidTest - Installs and runs instrumentation tests for all flavors on connected
devices.
connectedCheck - Runs all device checks on currently connected devices.
connectedDebugAndroidTest - Installs and runs the tests for debug on connected devices.
deviceAndroidTest - Installs and runs instrumentation tests using all Device Providers.
deviceCheck - Runs all device checks using Device Providers and Test Servers.
lint - Runs lint on all variants.
lintDebug - Runs lint on the Debug build.
lintRelease - Runs lint on the Release build.
test - Run unit tests for all variants.
testDebugUnitTest - Run unit tests for the debug build.
testReleaseUnitTest - Run unit tests for the release build.

Other tasks
-----------
jarDebugClasses
jarReleaseClasses
lintVitalRelease - Runs lint on just the fatal issues in the Release build.
```

Come possiamo notare, si tratta di un elenco piuttosto lungo, che il tool ci espone in gruppi. Per ogni task viene poi visualizzata una breve descrizione. A dire il vero quelli elencati non sono nemmeno tutti i task, ma solamente i principali. Lasciamo al lettore l'esecuzione del seguente comando:

```
./gradlew tasks --all
```

In questo caso notiamo la presenza di altri task, come nel seguente frammento, nel quale possiamo vedere il riferimento al modulo dell'applicazione di nome `app:` e delle versioni dei task associate a ciascuno dei *build type*. Questo è appunto il motivo per cui abbiamo rimosso i build variant da noi definiti; avrebbero portato alla generazione di un elenco di task molto lungo.

```
app:assembleDebug - Assembles all Debug builds. [app:compileDebugSources]
    app:dexDebug
    app:packageDebug
    app:preDexDebug
    app:validateDebugSigning
    app:zipalignDebug
app:assembleRelease - Assembles all Release builds. [app:compileReleaseSources]
    app:dexRelease
    app:packageRelease
    app:preDexRelease
```

Il lettore avrà capito che per l'esecuzione di un particolare task sarà sufficiente eseguire il comando:

```
./gradlew <nome task>
```

dove il nome del task è uno di quelli elencati. Interessante l'utilizzo di una sorta di organizzazione gerarchica nel nome del task stesso. Attraverso i comandi:

```
./gradlew assembleDebug
./gradlew assembleRelease
```

possiamo eseguire i task di creazione dell'APK rispettivamente per la versione di *debug* e per quella di *release*. La stessa operazione è possibile attraverso il solo comando:

```
./gradlew assemble
```

che esegue il task di *assemble* per ciascuno dei *build variant* definiti nel file di configurazione. Nel caso in cui non fossimo sicuri di quali task si eseguano con un particolare comando è possibile utilizzare l'opzione `dry run`, la quale non esegue il *task* vero e proprio, ma dà indicazione delle dipendenze tra task elencandone la sequenza. Se eseguiamo il seguente comando:

```
./gradlew assembleDebug --dry-run
```

otteniamo la seguente sequenza, nella quale possiamo notare la presenza della parola SKIPPED, che indica appunto come il *task* non sia stato eseguito:

```
:app:preBuild SKIPPED
:app:preDebugBuild SKIPPED
:app:checkDebugManifest SKIPPED
:app:preReleaseBuild SKIPPED
:app:prepareComAndroidSupportAppcompatV7222OLibrary SKIPPED
:app:prepareComAndroidSupportSupportV42220Library SKIPPED
:app:prepareDebugDependencies SKIPPED
:app:compileDebugAidl SKIPPED
:app:compileDebugRenderscript SKIPPED
:app:generateDebugBuildConfig SKIPPED
:app:generateDebugAssets SKIPPED
:app:mergeDebugAssets SKIPPED
:app:generateDebugResValues SKIPPED
:app:generateDebugResources SKIPPED
:app:mergeDebugResources SKIPPED
:app:processDebugManifest SKIPPED
:app:processDebugResources SKIPPED
:app:generateDebugSources SKIPPED
:app:processDebugJavaRes SKIPPED
:app:compileDebugJavaWithJavac SKIPPED
:app:compileDebugNdk SKIPPED
:app:compileDebugSources SKIPPED
:app:preDexDebug SKIPPED
:app:dexDebug SKIPPED
:app:validateDebugSigning SKIPPED
:app:packageDebug SKIPPED
:app:zipalignDebug SKIPPED
:app:assembleDebug SKIPPED

BUILD SUCCESSFUL
```

Il lettore a questo punto si potrebbe chiedere come mai l'esecuzione di un solo task presupponga l'esecuzione di una lunga serie di altri task; quello richiesto è infatti l'ultimo della sequenza. Questo è dovuto al fatto che i vari task sono legati da una relazione di dipendenza. Riprendendo sempre lo stesso esempio notiamo come la creazione dell'APK dell'applicazione presupponga l'esecuzione di altri task, come quello di compilazione, di merge delle risorse e file di configurazione, di validazione e altro ancora. Noi non ci dilungheremo nei dettagli, ma come ultima cosa andiamo a vedere quali siano i task principali che sono anche i più utili durante lo sviluppo vero e proprio.
Nella parte iniziale, dedicata a Gradle, abbiamo detto come il plugin di Android sia un'estensione di quello dedicato a Java, che a sua volta estende quello di base. Quest'ultimo mette sempre a disposizione i seguenti due task:

```
clean
assemble
```

Il primo permette di ripulire l'ambiente, mentre il secondo permette di assemblare il risultato finale. Queste sono operazioni utili a un qualunque progetto realizzato con

una qualunque tecnologia. Il plugin dedicato allo sviluppo Java ha poi aggiunto, oltre al concetto di `SourceSet`, altri due task:

```
check
build
```

Il primo è molto importante, in quanto permette di svolgere tutta una serie di controlli da eseguire prima della creazione del risultato finale; tra questi abbiamo per esempio l'esecuzione dei test. Infine il task di build non fa altro che eseguire prima il *check* e quindi l'*assemble*. I task definiti dal plugin Java possono essere poi specializzati in base al particolare ambiente come del resto avviene con il plugin Android. Il task *assemble* ha come risultato la creazione dell'APK dell'applicazione, mentre quello di *check* esegue, insieme agli eventuali test, alcuni strumenti di verifica con *Lint* (`http://developer.android.com/intl/es/tools/help/lint.html`). Si tratta di uno strumento molto utile, che permette di esaminare il codice eseguendo controlli relativi sia alla presenza di eventuali errori sia all'attinenza a eventuali standard di scrittura del codice. A seconda del tipo di problema, *Lint* può far fallire il task, impedendo la creazione dell'*APK*; in ogni caso fornisce una serie di report sotto forma di documenti HTML all'interno della cartella `app/build/outputs`. Ma Android Studio come ci aiuta in tutto questo? Se andiamo a vedere nella parte inferiore destra del nostro IDE notiamo la presenza di un tab di nome *Gradle*, selezionando il quale otteniamo il risultato rappresentato nella Figura 2.21.

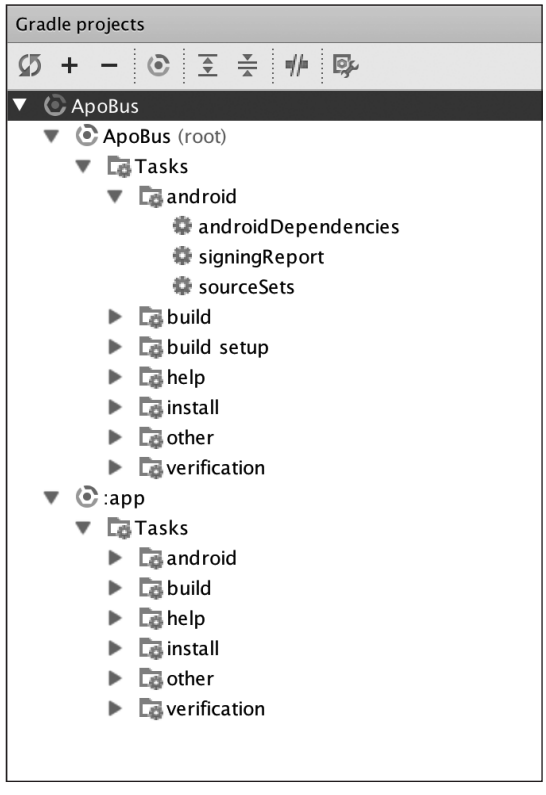

Figura 2.21 I task di Gradle in Android Studio.

Notiamo come i vari task siano organizzati secondo la struttura vista in precedenza da riga di comando. Attraverso la selezione di tasto destro del mouse è poi possibile eseguire singolarmente ciascuno di questi task, come abbiamo fatto in precedenza attraverso il nostro terminale.

Utilizzo di uno strumento di versioning

Dopo la creazione del progetto, Android Studio potrebbe aver visualizzato un alert come nella Figura 2.22, attraverso il quale viene richiesto se utilizzare o meno un sistema di versioning, che nel nostro caso è Git (`https://git-scm.com/`).
L'utilizzo di uno strumento di questo tipo è ormai obbligatorio nella realizzazione di un qualunque progetto, anche nel caso in cui vi fosse un unico programmatore. L'utilizzo di Git, come di altri strumenti analoghi, necessiterebbe troppo spazio, per cui non ci dilungheremo oltre. Se il lettore non dispone di alcun strumento di questo tipo potrà selezionare l'opzione `Ignore`.

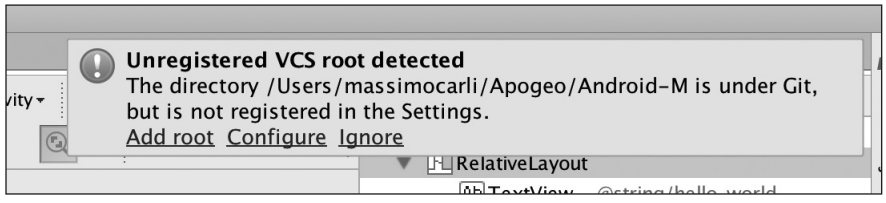

Figura 2.22 Aggiungiamo la root del progetto a Git.

I sorgenti e le risorse del progetto

Dopo aver descritto in dettaglio i file di configurazione di Gradle è giunto il tempo di dedicarci agli aspetti di sviluppo vero e proprio, cominciando da quanto generato da Android Studio in fase di creazione del progetto. Torniamo quindi nella vista in modalità Android, ottenendo quanto rappresentato nella Figura 2.23.
In particolare possiamo notare la presenza di tre importanti sezioni relative a:

- le risorse;
- il codice sorgente Java;
- il file di configurazione `AndroidManifest.xml`.

La creazione di un'applicazione consisterà, appunto, nella creazione degli opportuni file sorgenti Java, risorse e nella relativa configurazione all'interno del file `AndroidManifest.xml`. Di seguito vedremo il ruolo di ciascuna di queste componenti, per poi entrare nel dettaglio nel corso dei prossimi capitoli.

Le risorse

Le risorse rappresentano una parte di fondamentale importanza di ciascuna applicazione Android e sono contenute all'interno della cartella di nome `res`. Come possiamo notare nella Figura 2.23 esistono diversi tipi di risorse, ciascuna contenuta in una sua cartella.

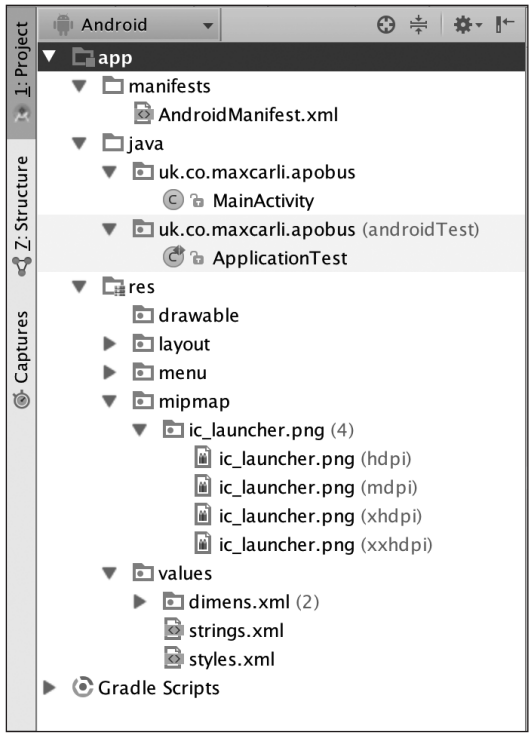

Figura 2.23 La vista Android per i file del nostro progetto.

Per il momento pensiamo alle risorse solamente come a particolari file di configurazione o a immagini accessibili dal nostro codice e dotate della possibilità di essere opportunamente selezionate in base al particolare dispositivo che esegue la nostra applicazione. Un esempio di questa loro caratteristica è visibile nel nostro progetto osservando le risorse di tipo `mipmap` contenute all'interno del folder onomimo. Si tratta di immagini che vengono utilizzate come icone della nostra applicazione sul display del dispositivo. Come possiamo osservare nella Figura 2.23 esistono quattro versioni dello stesso file, ciascuna caratterizzata da una label che esprime, in questo caso, la densità del display. Per essere sintetici vedremo come un dispositivo con display classificato di densità media (`mdpi`) sceglierà quelle particolari risorse annotate con la label `mdpi`, mentre uno classificato di densità alta (`xhdpi`) sceglierà quelle annotate come `xhdpi`, e così via. I criteri utilizzati dalla piattaforma nella selezione delle risorse da impiegare prendono il nome di qualificatori e sono mostrati come indicato nella Figura 2.23, ovvero tra parentesi. In realtà ciascuna risorsa è contenuta in una cartella, il cui nome riprende quello associato al tipo di risorsa e al particolare qualificatore. Quelle relative alle risorse di tipo `mipmap`, per esempio, sono contenute nelle cartelle indicate nella Figura 2.24:
La selezione della risorsa opportuna per ogni dispositivo permette una gestione ottimale delle capacità del dispositivo stesso. Pensiamo per esempio al caso in cui un dispositivo abbia la necessità di effettuare il resize di un'immagine di dimensioni maggiori del dovuto. In questo caso il danno sarebbe doppio, in quanto legato a uno spreco di memoria (immagine troppo grande) e di elaborazione (il resize), con conseguente esaurimento della risorsa a noi più cara, ovvero la batteria.

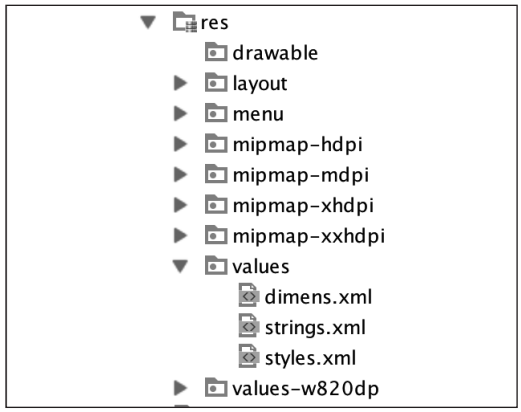

Figura 2.24 La struttura a directory per risorse associate a qualificatori diversi.

Mentre per le altre tipologie di risorse dedicheremo ampio spazio, vogliamo dare qualche informazione in relazione alle risorse di tipo mipmap che, come abbiamo detto, sono immagini. In Android le immagini sono considerate risorse di tipo `Drawable`, ma dalla versione 4.3 (*Api Level 17*) è stato deciso di creare una nuova tipologia che si chiama appunto *mipmap*. Senza entrare troppo nel dettaglio, si tratta di immagini che utilizzano un formato che le rende ottimizzate in fase di renderizzazione nel caso in cui questa necessiti di un ridimensionamento. Questo le rende adatte a un utilizzo come icone, in quanto alcuni dispositivi utilizzavano immagini di risoluzione superiore per poi rimpicciolirle al fine di mantenere un'ottima risoluzione.
Un'altra tipologia di risorse di fondamentale importanza è rappresentata dai *layout*, che sono contenuti in una cartella con lo stesso nome. Anche queste risorse vengono scelte in base all'utilizzo di qualificatori tra cui, per esempio, quelli relativi all'orientamento del dispositivo o alle dimensioni dello schermo. In questa fase ci interessa sottolineare che cosa sia un *layout* e quali siano gli strumenti che abbiamo a disposizione per la loro gestione.

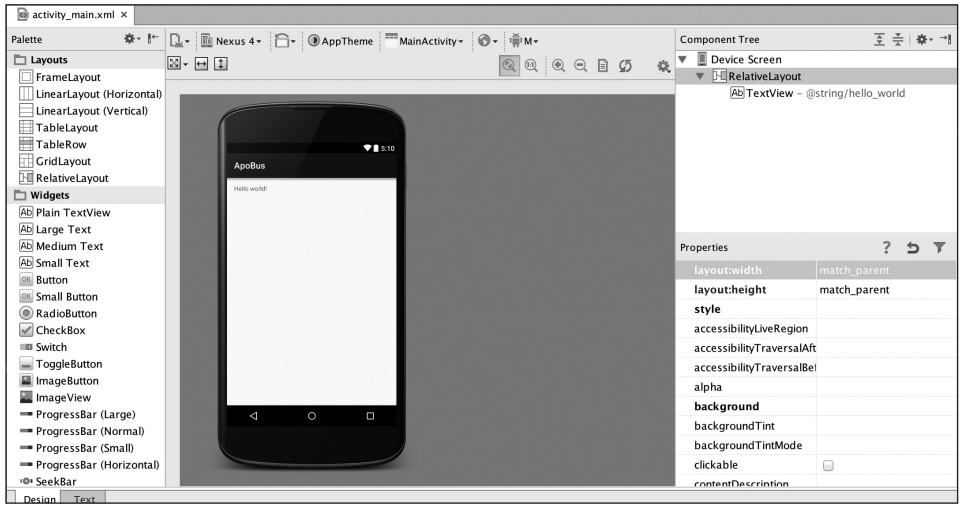

Figura 2.25 La gestione dei layout nell'editor.

Come dice il nome stesso, si tratta di risorse che permettono di definire in modo dichiarativo le interfacce della nostra applicazione, quella che viene spesso indicata come UI (User Interface). Per il momento prendiamo quello che è stato realizzato automaticamente dal nostro plug-in in fase di creazione del progetto. Selezioniamo il file `activity_main.xml` nella cartella `/res/layout` ottenendo la visualizzazione di alcune finestre (Figura 2.25). Come possiamo osservare, l'interfaccia dell'IDE è divisa sostanzialmente in tre colonne, dove l'ultima è suddivisa in due righe. La prima colonna è associata di default a un pulsante di nome `Palette` che contiene tutti i componenti visuali che possiamo trascinare all'interno della preview dell'interfaccia nella parte centrale; questa contiene, nella parte inferiore, due schede, di nome `Design` e `Text`. La prima è selezionata di default e permette la visualizzazione di una preview, mentre la seconda permette di vedere il codice sorgente XML. Prima di vedere come appare questo documento XML, notiamo come nella parte destra in alto vi sia la corrispondente visualizzazione ad albero. Tutti gli elementi definiti all'interno del documento di *layout* vengono visualizzati in modalità gerarchica. Questo tool è utile nel caso in cui sia necessario vedere quali componenti ne contengono altri e così via. Nella parte destra inferiore abbiamo infine il pannello delle proprietà. Selezionando un componente se ne possono modificare le proprietà nella modalità ormai classica di qualunque IDE.
Come già detto, una risorsa di layout è un documento XML il cui sorgente può essere modificato selezionando l'apposito tab di nome `Text` nella parte inferiore sinistra ottenendo quanto rappresentato nella Figura 2.26.

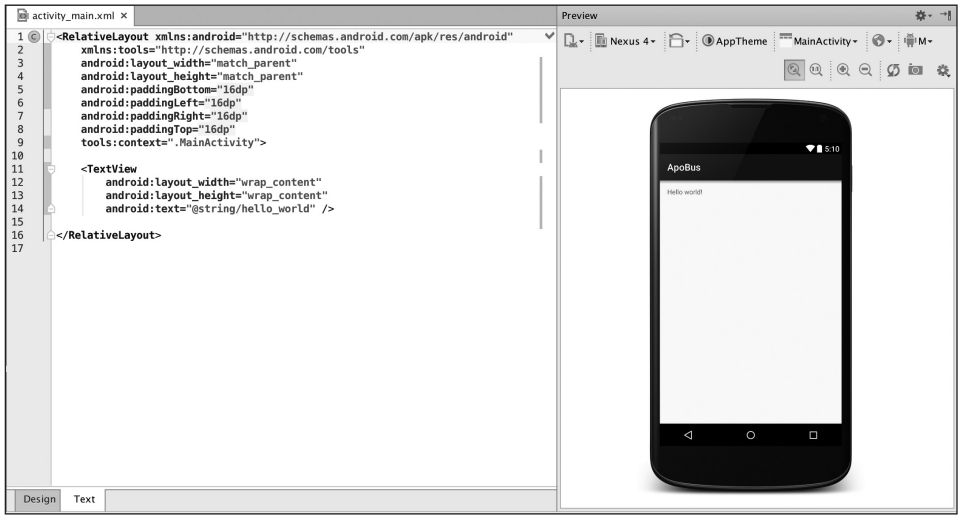

Figura 2.26 Documento di layout come XML.

Come possiamo notare, l'interfaccia cambia ancora e mostra nella parte sinistra il codice sorgente del layout e nella parte destra ancora una preview del risultato. La preview è uno strumento molto utile, in quanto ci permette di avere una prima idea di come apparirà la nostra applicazione su display con caratteristiche diverse. Per fare questo è infatti possibile selezionare il menu a tendina e scegliere un modello di riferimento tra quelli elencati nella Figura 2.27.

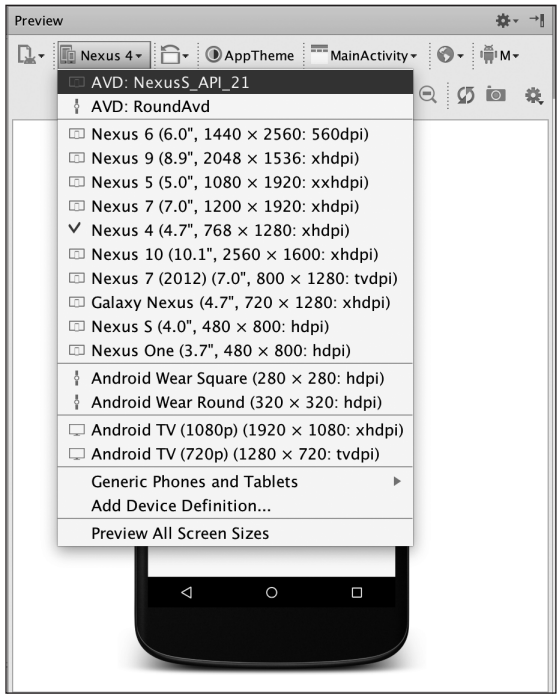

Figura 2.27 Preview per i principali dispositivi.

Come possiamo vedere in figura vi è un elenco di dispositivi, classificati per dimensione e risoluzione del display. Nel caso in cui volessimo però avere una preview basata su una specifica versione di Android possiamo utilizzare un altro menu a tendina e precisamente quello rappresentato nella Figura 2.28 nel quale notiamo la presenza della versione 6.0 di Android, a differenza di quanto visto in precedenza; probabilmente per un problema di label.

È interessante notare come in entrambi i menu vi sia la possibilità di visualizzare più *preview* contemporaneamente, in modo da facilitare eventuali confronti. Nel caso si selezionasse l'opzione *Preview All Screen sizes* si otterrebbe quanto rappresentato nella Figura 2.29.

Si tratta di una funzione molto importante, che permette di ridurre notevolmente il tempo sui dispositivi reali; era infatti facile incappare spesso in UI non volute su particolari tipologie di dispositivi. Inoltre possiamo selezionare un elemento nella preview e il cursore verrà posizionato automaticamente nel punto corrispondente nel sorgente nella parte sinistra. Si tratta di funzioni che alla lunga risulteranno molto utili. Vedremo che lo stesso strumento si potrà utilizzare per testare le UI a seguito della modifica di altri qualificatori classici, come quello della lingua, dell'orientamento e della versione di API Level disponibile.

Torniamo al nostro documento di *layout*, nel quale abbiamo evidenziato alcuni elementi molto importanti:

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

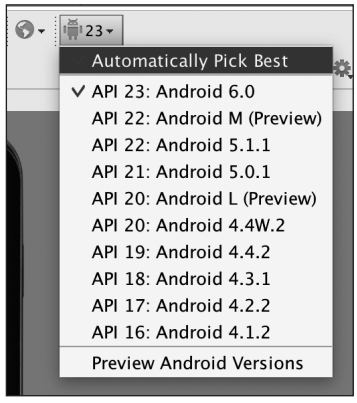

Figura 2.28 Preview basata sull'Api Level.

Figura 2.29 Preview multipla per diversi dispositivi.

Innanzitutto notiamo come si tratti di un documento XML che contiene alcuni elementi che permettono di descrivere gli elementi grafici con la loro dimensione e posizione. Come vedremo più avanti esistono diversi tipi di componenti grafici, a cui corrispondono altrettanti elementi *XML*. Alcuni di questi componenti corrispondono a componenti di base, come la `<TextView/>`, mentre altri hanno funzione di contenitore come il `<RelativeLayout/>`. Notiamo come ciascun elemento disponga di alcuni attributi che si differenziamo per *namespace*.

> **NOTA**
>
> Per chi non ha dimestichezza con XML possiamo dire che un namespace è un identificatore di una sorta di alfabeto, ovvero un insieme di elementi e attributi che si possono utilizzare all'interno di un documento. Ciascun namespace è caratterizzato da un *Uniform Resource Identifier* (URI) che, sebbene abbia l'aspetto di un indirizzo web, non corrisponde necessariamente a una pagina accessibile attraverso il browser; si tratta, come dice il nome stesso, appunto di un identificatore.

Nel nostro documento notiamo la presenza di due *namespace* associati alle label `android` e `tools` che permettono di contestualizzare gli attributi utilizzati nel documento stesso. Tutti gli attributi che iniziano per `android:` saranno quindi relativi ad aspetti legati alla piattaforma, mentre quelli che iniziano per `tools:` sono associati a funzionalità di Android Studio. L'utilizzo di un documento XML ha infatti senso solamente se esiste un *parser* che ne estrae le informazioni e che le interpreta in relazione al *namespace* utilizzato. Come possiamo notare, gli attributi ci permettono di valorizzare alcune proprietà dei componenti definiti nel documento. Per esempio, l'attributo `android:text` della `TextView` permette di specificare il testo da visualizzare al suo interno, che notiamo seguire una sintassi particolare, ovvero:

```
android:text="@string/hello_world"
```

Possiamo generalizzare attraverso una notazione del tipo

```
@<tipo risorsa>/nome risorsa
```

Si tratta di un aspetto fondamentale di tutta l'architettura di Android, che abbiamo voluto affrontare immediatamente. Dall'interno di un documento XML di *layout* (ma vedremo che lo stesso varrà nel caso di altri tipi di documenti) possiamo fare riferimento al valore di una risorsa attraverso una sintassi del tipo indicato. Se andiamo a cercare il valore di questa risorsa è sufficiente visualizzare il file `strings.xml` nella cartella `res/values`:

```
<resources>
    <string name="app_name">ApoBus</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
</resources>
```

Nel nostro caso, attraverso la sintassi `@string/hello_world` si fa riferimento al valore dato dalla stringa `Hello world!`. Il lettore si potrebbe chiedere quale possa essere il motivo di questa sintassi; la risposta cade sempre sul concetto di qualificatori.

Uno di questi, spesso associato alle risorse di tipo `String`, è quello legato alla lingua impostata nel dispositivo. Per darne dimostrazione selezioniamo con il pulsante destro la cartella delle risorse e selezioniamo la voce `New > Android resource file` come indicato nella Figura 2.30. A questo punto otteniamo la finestra rappresentata nella Figura 2.31, nella quale possiamo specificare le informazioni della risorsa che intendiamo creare tra cui il nome del file, il tipo di risorsa e la *build variant* associata. È bene sottolineare come si stia creando un file che potrà essere associato a una o più risorse a seconda del tipo; un file potrà essere associato a un'immagine, ma potrà contenere, per esempio, le definizioni di più `String`.

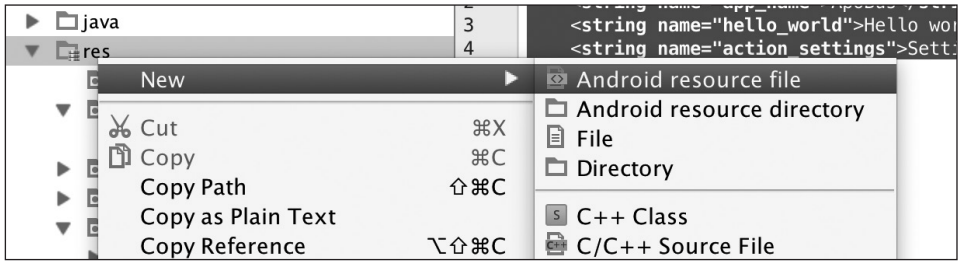

Figura 2.30 Creazione di una nuova risorsa.

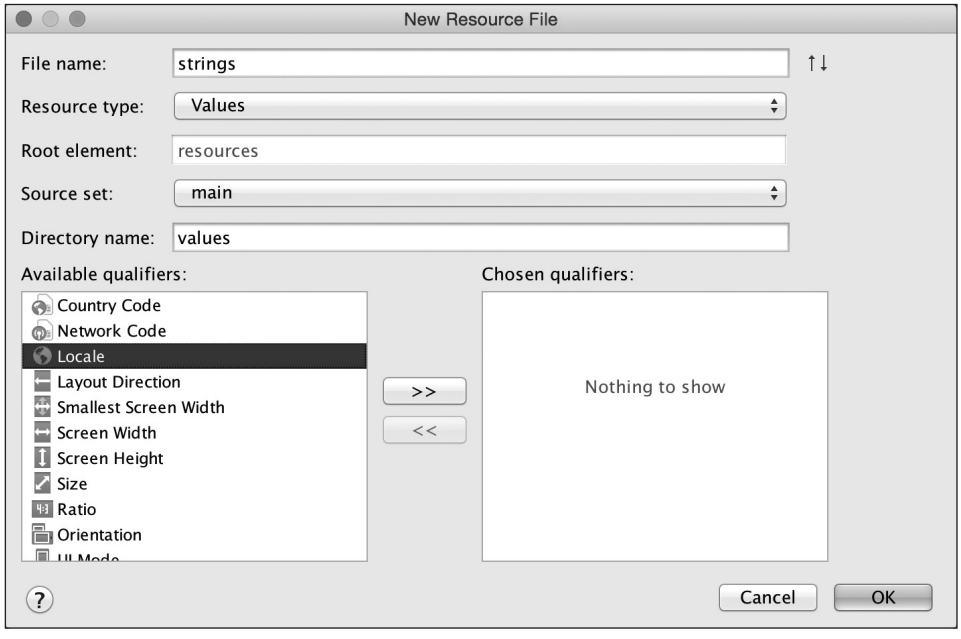

Figura 2.31 Informazioni associate alla risorsa da creare.

Nella parte inferiore della schermata possiamo notare la presenza di un elenco di qualificatori. Nel nostro caso vogliamo creare la versione italiana di alcune risorse di tipo string, per cui andiamo a selezionare, come in figura, la label `Locale`; selezioniamo quindi il pulsante con le frecce verso destra arrivando alla schermata rappresentata nella Figura 2.32.

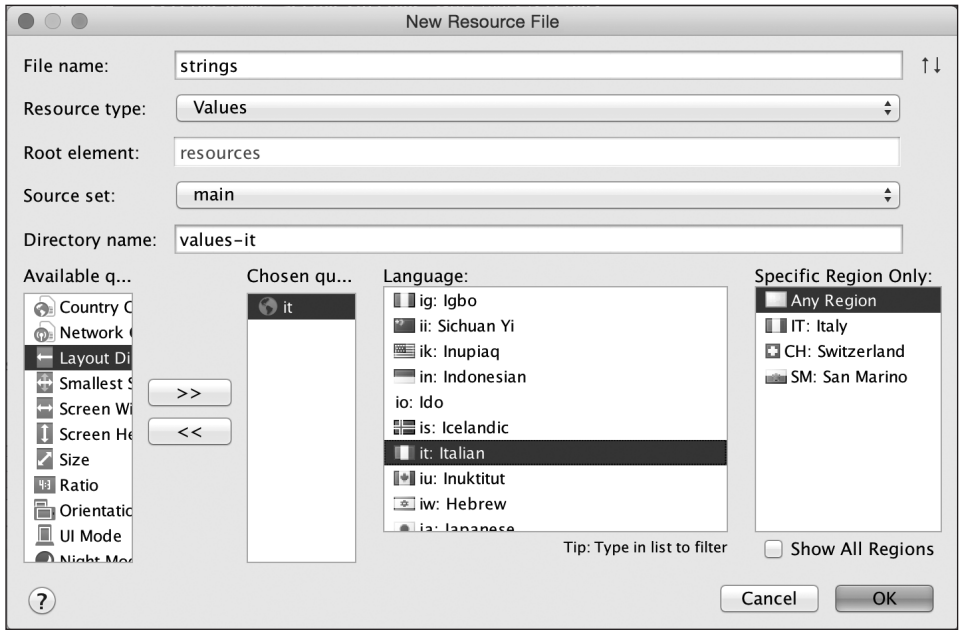

Figura 2.32 Selezioniamo il locale per la nostra risorsa.

A questo punto selezioniamo la nostra lingua e quindi il pulsante OK. Il risultato è la creazione di un nuovo file di nome `strings.xml` associato questa volta al *locale* italiano, come possiamo vedere nella Figura 2.33.

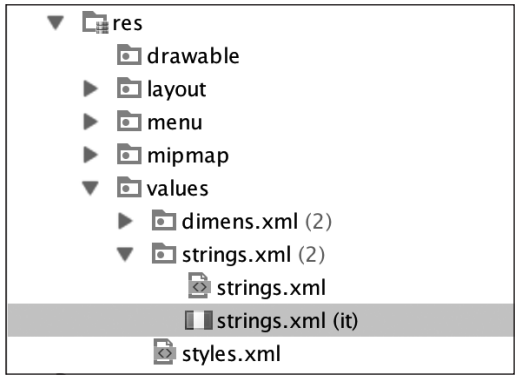

Figura 2.33 La creazione del file strings associato al locale italiano.

Come accennato in precedenza il file verrà inserito in una cartella che si chiama `values-it` all'interno di `/res`, ma viene visualizzato come in figura se si utilizza la vista *Android*. A questo punto possiamo definire le stesse risorse associando alle stesse chiavi valori diversi, come nel seguente documento:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
    <string name="app_name">ApoBus</string>
    <string name="hello_world">Ciao Mondo!</string>
    <string name="action_settings">Impostazioni</string>
</resources>
```

A questo punto nel nostro *layout* faremo sempre e comunque riferimento alla risorsa identificata dalla sintassi `@string/hello_world`, ma a runtime il dispositivo andrà a prendere il valore corrispondente alla lingua impostata. Come accennato si tratta di una caratteristica di tutte le risorse, come si può vedere nella seconda definizione evidenziata in precedenza:

```
android:paddingLeft="@dimen/activity_horizontal_margin"
```

dove la risorsa questa volta è di tipo `dimension`. In quel caso il qualificatore potrà essere eventualmente legato alle dimensioni del display oppure più probabilmente alla sua risoluzione. Concludiamo questa parte introduttiva relativa alle risorse utilizzando la *preview* per la visualizzazione delle label appena create attraverso l'opzione visibile nella Figura 2.34.

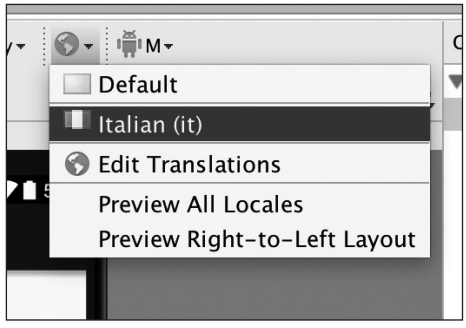

Figura 2.34 Utilizzo della preview nel caso di Locale diversi.

Lasciamo al lettore la verifica di cosa venga visualizzato nel caso in cui si selezioni la lingua italiana o quella di default. Si tratta comunque di una funzione molto utile, specialmente nel caso in cui le traduzioni portino a modifiche della UI a causa di label troppo lunghe o troppo corte.

I sorgenti Java

Nel precedente paragrafo abbiamo parlato delle risorse e della loro importanza. Ma a cosa servono e, soprattutto, dove si utilizzano? Un primo esempio ci è dato dalla classe Java `MainActivity`, generata in modo automatico da Android Studio a seguito della nostra scelta indicata nella Figura 2.4. Il codice sorgente è il seguente e ne approfittiamo per riprendere brevemente i concetti visti nel primo capitolo.

```
package uk.co.maxcarli.apobus;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
```

```
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

Ricordando innanzitutto che un'`Activity` è la descrizione di una schermata dell'applicazione, notiamo come si tratti di una classe che estende indirettamente l'omonima classe del `package android.app`. La classe `AppCompatActivity` è infatti una classe che eredita da `Activity` e permette la gestione, tra le altre cose, della `ActionBar` anche in versioni per la quale non era definita.

> **NOTA**
>
> Nel caso in cui avessimo scelto Marshmallow come versione minima, la nostra attività avrebbe esteso direttamente l'omonima classe *Activity*.

Più avanti vedremo in dettaglio il ciclo di vita di questi componenti; per il momento osserviamo come il *layout* da noi creato e che vogliamo assegnare alla nostra schermata sia stato associato a una costante di una classe che si chiama `R`. Questa è la seconda e fondamentale proprietà delle risorse, ovvero di generare, per ciascuna di esse, una costante di una classe `R` che ne permetta il riferimento dall'interno del codice Java.

In realtà per ciascuna tipologia di risorsa verrà generata un'opportuna classe statica interna, che nel caso del *layout* si chiama, appunto, `R.layout`, mentre nel caso delle stringhe si chiama `R.string`. Per vedere cosa viene generato possiamo verificare il contenuto della classe `R` all'interno della cartella dei file temporanei (Figura 2.35).

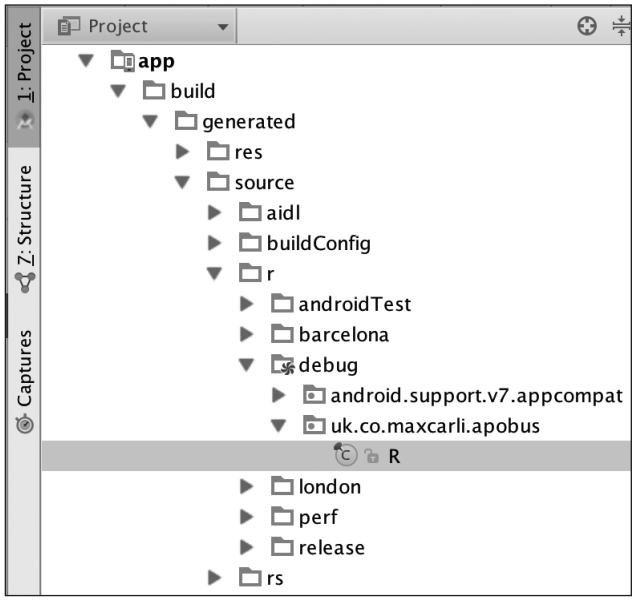

Figura 2.35 La classe R generata in modo automatico in fase di build.

Come possiamo notare si tratta di una classe che appartiene allo stesso package dell'applicazione; viene generata in modo automatico e che quindi non possiamo modificare; se lo facessimo perderemmo le nostre modifiche al successivo *build*. Se andiamo a vederne il contenuto notiamo come si tratti di una classe molto lunga a seguito dell'utilizzo della libreria di supporto. Possiamo notare comunque la presenza delle costanti relative alle risorse di tipo `layout`, `string` e `dimension` viste in precedenza, che abbiamo estratto di seguito:

```
package uk.co.maxcarli.apobus;
public final class R {
    - - - -
    public static final class dimen {
        public static final int activity_horizontal_margin=0x7f070010;
        public static final int activity_vertical_margin=0x7f070041;
    }
    - - - -
    public static final class layout {
        public static final int activity_main=0x7f040019;
    }
    - - - -
    public static final class menu {
        public static final int menu_main=0x7f0d0000;
    }
```

```
    - - - -
    public static final class mipmap {
        public static final int ic_launcher=0x7f030000;
    }
    public static final class string {
        public static final int action_settings=0x7f06000d;
        public static final int app_name=0x7f06000e;
        public static final int hello_world=0x7f06000f;
    }
    - - - -
}
```

Vedremo come l'utilizzo delle risorse sia di fondamentale importanza e come la piattaforma disponga di moltissimi strumenti che si aspettano come possibili valori dei propri parametri quelli associati alle costanti precedenti. Quello che ci interessa sottolineare al momento riguarda solamente la possibilità di poter referenziare e utilizzare le varie risorse attraverso le costanti della classe `R` generata in modo automatico. Nella classe abbiamo quindi messo in evidenza come sia stata utilizzata la costante `R.layout.activity_main`, per il riferimento al documento di *layout*, oppure la costante `R.menu.menu_main`, nel caso di una risorsa di *menu*. Caratteristica fondamentale di queste costanti è quella di rimanere sempre le stesse indipendentemente dagli eventuali qualificatori applicati. Per quanto visto nel caso delle `String`, faremo quindi riferimento sempre alla label associata alla costante `R.string.hello_world` sia che si faccia riferimento alla versione italiana sia a quella di *default*. La costante è sempre la stessa, mentre il valore sarà quello corrispondente alle caratteristiche e configurazioni del particolare dispositivo.

Il file di configurazione AndroidManifest.xml

La terza componente che andiamo a esaminare è contenuta invece in una cartella di nome `manifests`. Come notato anche in precedenza il nome è al plurale, in quando sarà possibile vedere tutte le eventuali versioni associate alle varie *build variants*.
Nel caso in cui avessimo mantenuto e reso attivo il *built type* di nome `perf` avremmo ottenuto, per esempio, la vista rappresentata nella Figura 2.36, dove la presenza di più file di configurazione `AndroidManifest.xml` è visualizzata in modo esplicito.

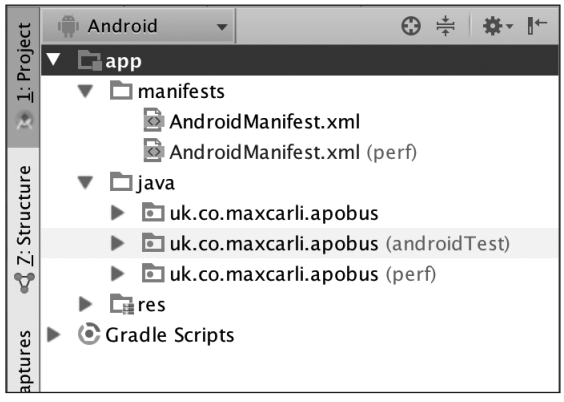

Figura 2.36 Esempio della presenza di più file di configurazione.

Ma che cosa è questo file di configurazione che abbiamo già citato più volte? Si tratta di un documento *XML* che descrive al dispositivo l'applicazione in termini dei componenti che essa contiene e di come questi collaborino tra loro e con il sistema stesso. Nel nostro esempio, quello che si chiama anche *deployment descriptor* è il seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="uk.co.maxcarli.apobus" >
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Notiamo che si tratta di un documento XML la cui `root` è rappresentata dall'elemento `<manifest/>` che contiene un attributo fondamentale che si chiama `package` e che identifica in modo univoco l'applicazione e che quindi dovrà rimanere lo stesso per tutta la sua vita. Insieme al *package* questo elemento dovrebbe contenere anche le informazioni relative al `versionCode` e al `versionName` che abbiamo visto all'interno del file di configurazione di *Gradle*. In realtà queste informazioni vengono aggiunte proprio in fase di *build* e contengono i valori specificati per il particolare *build type*.

> **NOTA**
>
> Si tratta di un'operazione di merge tutt'altro che banale; per i dettagli si rimanda alla documentazione ufficiale.

Lo stesso discorso vale per le informazioni relative agli attributi `minSdkVersion` e `targetSdkVersion` che possono essere specificate attraverso un elemento del tipo `<uses-sdk/>`, all'interno di `<manifest/>`, anch'esso aggiunto in fase di *build* in dipendenza del particolare *build type*. A questo punto è possibile specificare le informazioni relative all'applicazione vera e propria attraverso l'elemento `<application/>`, il quale contiene alcuni attributi i cui valori sono rappresentati da riferimenti ad alcune risorse. Nel precedente paragrafo abbiamo visto come sia possibile accedere alle risorse attraverso opportune costanti della classe `R`. In questo caso abbiamo invece un assaggio di come sia possibile fare riferimento a una risorsa dall'interno di un file di configurazione. Come vedremo in dettaglio nel prossimo capitolo, si utilizza una notazione del tipo

```
@[package]:<tipo risorsa>/<nome risorsa>
```

che nel caso di una risorsa di tipo String può essere la seguente

```
@string/app_name
```

in quanto il *package* è opzionale. Notiamo come si faccia riferimento a delle risorse come modo per delegare al dispositivo la selezione del valore a esso corrispondente. Nel caso dell'icona, per esempio, attraverso la notazione

```
@mipmap/ic_launcher
```

faremo riferimento all'immagine corrispondente alla risoluzione del dispositivo, mentre attraverso

```
@string/app_name
```

faremo riferimento alla label per la corrispondente lingua.
Il file `AndroidManifest.xml` permette di descrivere l'applicazione al dispositivo in termini di componenti quali `Activity`, `Service`, `Content Provider` e `BroadcastReceiver` e soprattutto della modalità con cui questi collaborano con quelli delle altre applicazioni o di sistema. Il tutto avverrà attraverso opportuni elementi all'interno di `<application/>`.
In questa primissima versione dell'applicazione notiamo come la nostra `Activity` sia stata definita attraverso l'elemento `<activity/>` e come questa sia stata associata a una particolare azione definita attraverso un elemento `<action/>`. Quello degli *intent* e degli *intent filter* è uno dei concetti fondamentali di Android, che non vogliamo però affrontare in questa fase. Per il momento diciamo solamente che la nostra `Activity` è stata associata a un evento relativo alla selezione dell'icona dell'applicazione nella *home* del dispositivo. Questo è anche il significato dell'utilizzo della `<category/>` di nome `LAUNCHER` nella stessa definizione. Quando installeremo l'applicazione e selezioneremo la corrispondente icona nella *home* del dispositivo, verrà generato un evento (*intent*) che verrà ascoltato dal sistema, il quale manderà in esecuzione la nostra `Activity`, con la conseguente visualizzazione del *layout* associato.

Eseguiamo l'applicazione creata

Il lettore troverà forse strano che sia già possibile eseguire l'applicazione creata senza scrivere codice. In realtà, in fase di creazione del progetto, abbiamo preparato tutto quello che ci serve, ovvero un'`Activity` dotata di *layout* in grado di essere lanciata dalla *home* del dispositivo. Sebbene sia di fondamentale importanza testare le applicazioni sui dispositivi reali, l'ambiente Android mette a disposizione una serie di emulatori istanziabili attraverso un AVD (*Android Virtual Device*), che rappresenta una possibile configurazione di cui un dispositivo reale può essere dotato. A tale proposito procediamo con la creazione di una particolare istanza di emulatore, relativa alla versione corrispondente alla preview di *Android M* indicata appunto come *MNC*. Prima di fare questo vogliamo assicurarci di avere tutto quello che ci serve, attraverso uno strumento che si chiama *SDK Manager* a cui possiamo accedere selezionando l'icona indicata nella Figura 2.37:

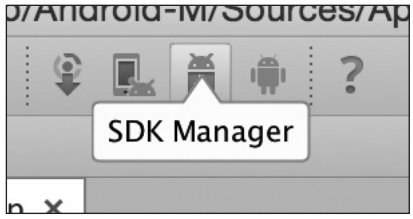

Figura 2.37 Accesso al SDK Manager da Android Studio.

Si tratta di uno strumento che ci permetterà di gestire tutti gli strumenti relativi alle varie versioni della piattaforma, nonché le immagini dei corrispondenti emulatori. Selezionando il pulsante indicato nella figura viene visualizzata la schermata rappresentata nella Figura 2.38, che notiamo contenere tre diversi tab. Il primo è quello di nome *SDK Platforms* e contiene un elenco delle versioni di Android disponibili al download. Nel nostro caso specifico possiamo notare come sia disponibile un aggiornamento relativo a Marshmallow, per cui lo andremo a selezionare. Prima di confermare andiamo però a selezionare il tab *SDK Tools*, che vediamo nella Figura 2.39.

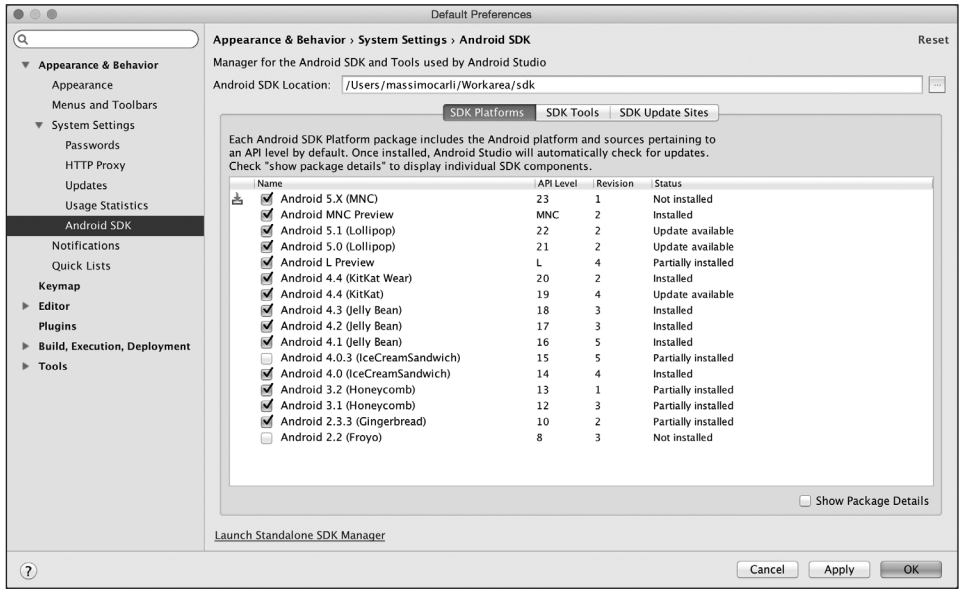

Figura 2.38 SDK Manager per la gestione delle diverse versioni di Android disponibili.

Anche in questo caso andiamo a selezionare quelle che ci interessano: al momento sono quelle di cui vediamo un aggiornamento in figura. In particolare notiamo come sia importante avere sempre una versione dei tool e dei repository delle librerie di supporto. Per l'esecuzione degli emulatori è importante anche scaricare e installare il seguente elemento:

```
Intel x86 Emulator Accelerator (HAXM Installer)
```

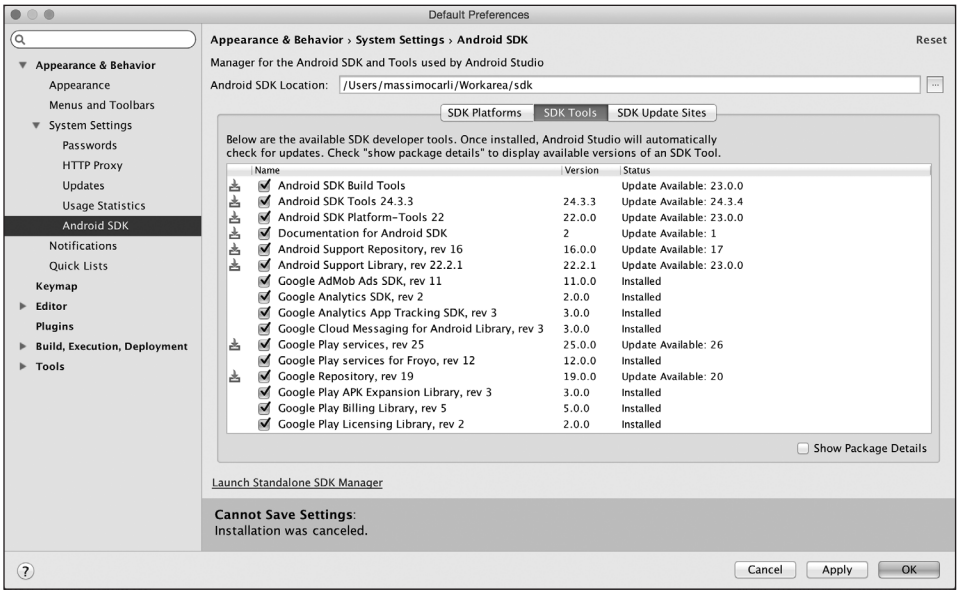

Figura 2.39 SDK Manager per la gestione dei tools.

Nel caso ci dimenticassimo sarà il tool di creazione degli emulatori a ricordarci di scaricare questo tool.
Il terzo tab contiene un elenco dei siti da cui queste informazioni vengono scaricate, ai quali è possibile aggiungere eventualmente quelli che permettono l'accesso alle versioni beta. A questo punto selezioniamo il pulsante `Apply` oppure `OK`, notando come i file selezionati vengano scaricati e installati. È importante sottolineare come questa sia un'operazione molto importante, che può avere come conseguenza l'aggiornamento di alcuni file di configurazione. È importante, infatti, che la versione degli strumenti specificati attraverso la proprietà `buildToolsVersion` nel file di configurazione di *Gradle* siano sempre allineati con quelli scaricati in questa fase.

> **NOTA**
>
> Il lettore non si preoccupi nel caso in cui *Andorid Studio* richiedesse nuovamente l'installazione dei tools attraverso un messaggio di warning. Accettiamo tranquillamente e sincronizziamo il file di *Gradle*.

Siamo quindi pronti alla creazione dell'AVD attraverso un tool che si chiama, appunto, *AVD Manager* a cui possiamo accedere attraverso il pulsante indicato nella Figura 2.40, che ci porta alla schermata rappresentata nella Figura 2.41.

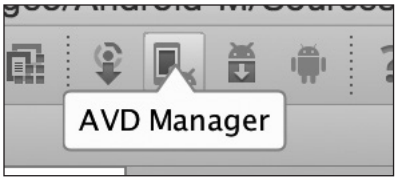

Figura 2.40 Accesso al AVD Manager da Android Studio.

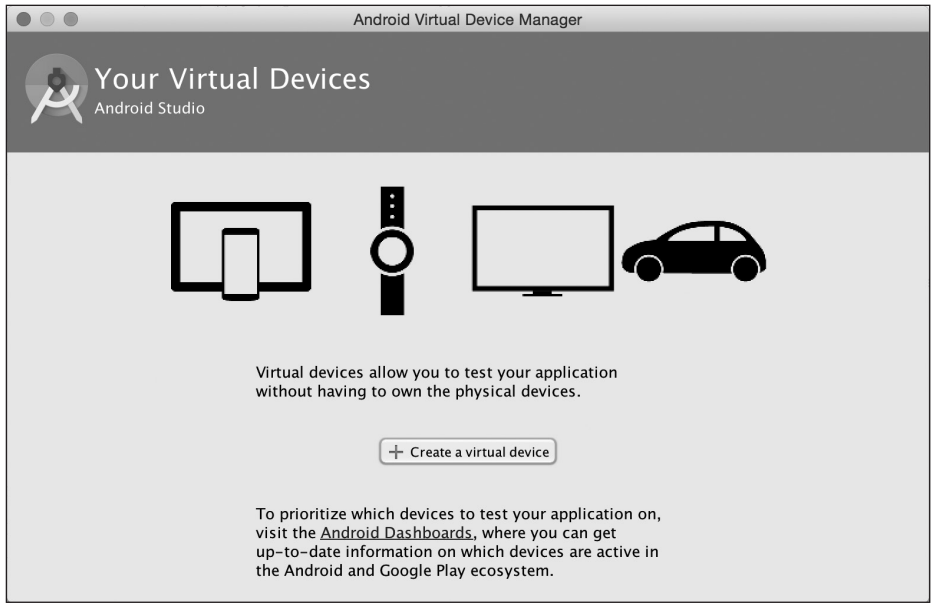

Figura 2.41 AVD Manager.

Come possiamo vedere è possibile creare diversi emulatori per smartphone, dispositvi wearable, TV e auto, in linea con le ultime versioni di Android.
Più avanti vedremo come gestire i dispositivi wearable, mentre per il momento selezioniamo il pulsante centrale `Creare a virtual device`, ottenendo la schermata rappresentata nella Figura 2.42 nella quale possiamo selezionare il modello e le caratteristiche hardware e software del dispositivo che intendiamo emulare.

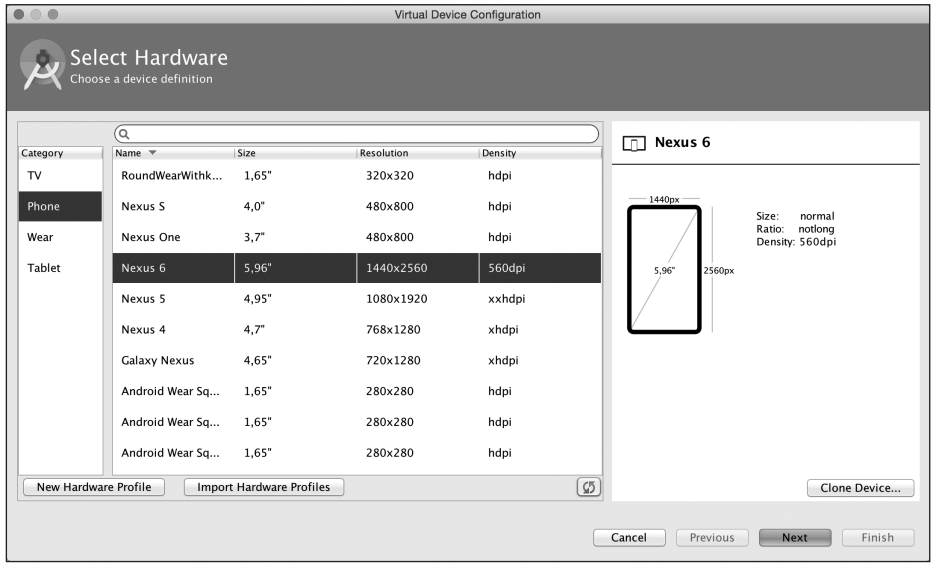

Figura 2.42 Selezioniamo il dispositivo con l'AVD Manager.

Nel nostro caso abbiamo selezionato un *Nexus 6*, per il quale possiamo osservare le informazioni relative alle dimensioni dello schermo e densità. Ovviamente la scelta del dispositivo non presuppone la scelta della particolare versione di Android, che andiamo invece a scegliere con la schermata successiva (Figura 2.43) alla quale arriviamo attraverso il pulsante `Next`.

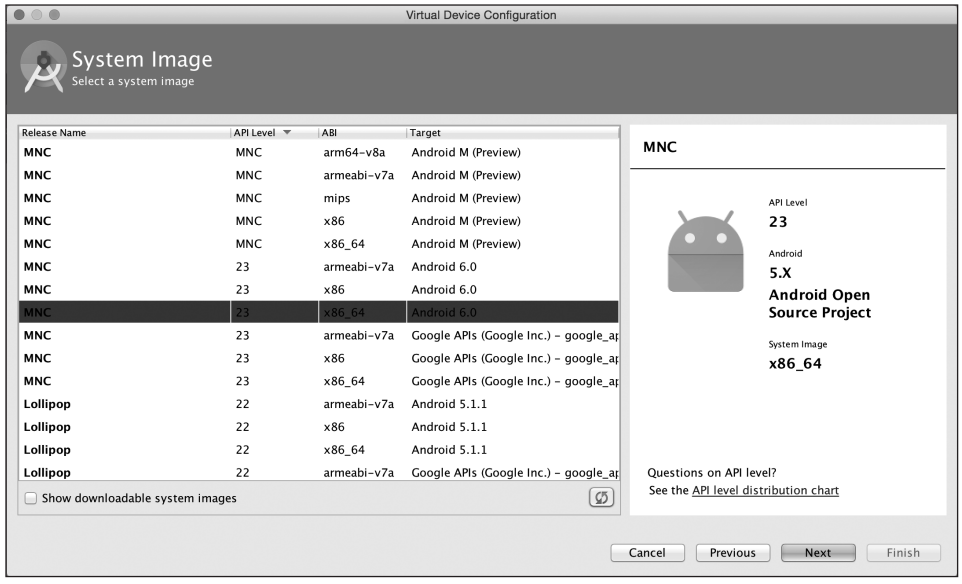

Figura 2.43 Selezioniamo la versione di Android del nostro AVD.

Nella figura sono visualizzate diverse versioni che dipendono da quelle scaricate in precedenza attraverso l'*SDK Manager*. Nel nostro caso selezioniamo la versione relativa all'*Api Level 23* con un processore *x86_64*. Selezionando ancora il pulsante *Next* arriviamo alla schermata rappresentata nella Figura 2.44, nella quale possiamo dare un nome al nostro AVD e scegliere le ultime impostazioni relative all'orientamento o alla possibilità di selezionare la quantità di memoria disponibile attraverso la corrispondente interfaccia riportata nella Figura 2.45, alla quale si arriva selezionando il pulsante *Show Advanced Settings*, in basso a sinistra.
Come è facile intuire, attraverso questo strumento è possibile definire in maniera piuttosto dettagliata le impostazioni del dispositivo che si intende emulare. Nel nostro caso abbiamo creato un dispositivo con Marshmallow, ma ovviamente dovremo creare anche dispositivi con caratteristiche hardware diverse, oltre che con *Api Level* diversi e corrispondenti a quelli per i quali abbiamo dichiarato la compatibilità.
Le impostazioni relative alla quantità di memoria disponibile sono infatti molto importanti nel caso in cui si volessero testare le applicazioni in dispositivi con capacità ridotte. Lo stesso vale per il tipo di connessione disponibile. Identico discorso si può fare per i dispositivi di input, nel caso volessimo testare l'utilizzo o meno della tastiera o di un qualche puntatore o pennino. Infine possiamo decidere se il nostro device dispone di una memoria fisica interna ed esterna (*SD Card*).

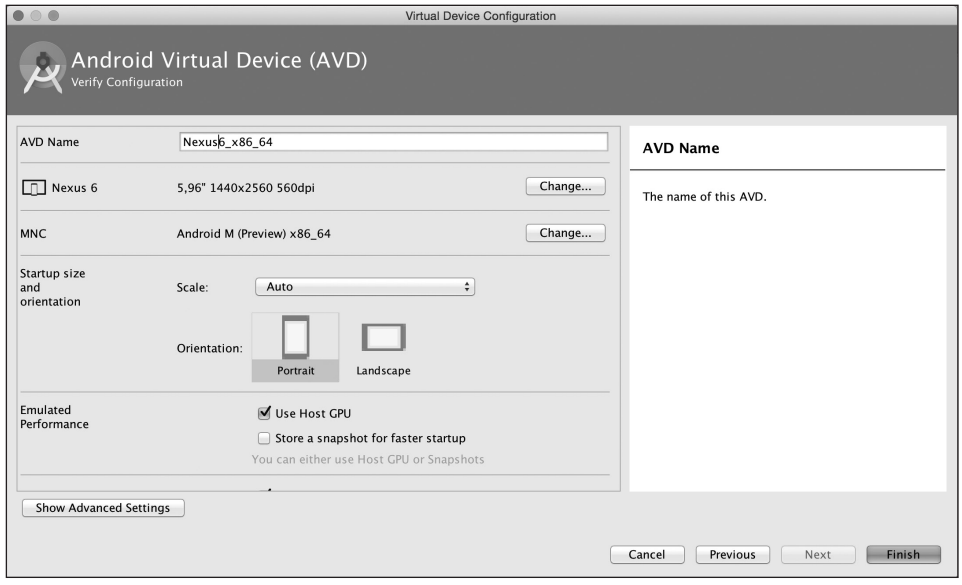

Figura 2.44 Le ultime impostazioni dell'AVD.

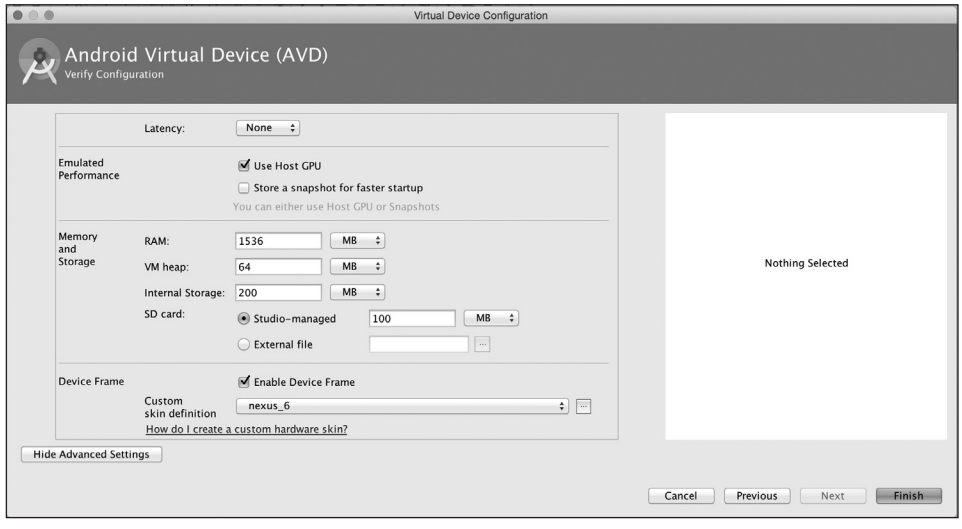

Figura 2.45 Impostazioni avanzate.

Siamo finalmente giunti al punto in cui il pulsante `Finish` è abilitato e può quindi essere premuto per arrivare alla schermata rappresentata nella Figura 2.46, che elenca tutti gli AVD creati.

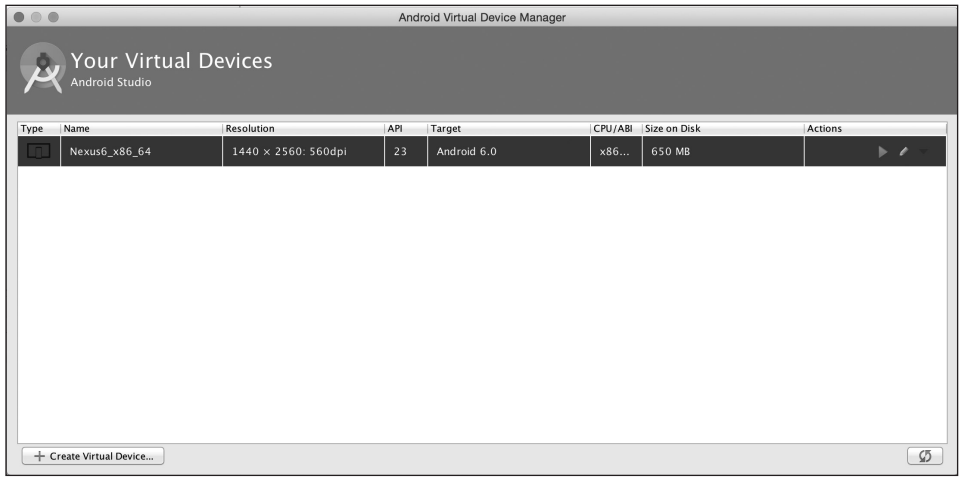

Figura 2.46 Schermata con tutti gli AVD creati.

Per ciascun AVD notiamo la presenza di alcune icone sulla destra, che possiamo selezionare per l'avvio o per la modifica delle corrispondenti configurazioni. Selezionando il triangolino di color verde possiamo avviare il nostro emulatore. Si tratta di un'operazione abbastanza onerosa che, specialmente la prima volta, richiederà un po' di tempo. Per velocizzare le esecuzioni successive, l'emulatore ha messo a disposizione l'opzione *Shapshot*, che permette di rendere persistente una configurazione di memoria che può essere ripristinata in modo veloce all'avvio successivo. Infine esiste un flag che consente di abilitare o meno la *Graphics Process Unit*, ma si tratta di una configurazione utile soprattutto quando si sviluppano giochi che utilizzano gli elementi grafici in modo molto pesante. Lanciamo quindi l'emulatore, ottenendo quanto rappresentato nella Figura 2.47 che è relativa al Nexus 6 che abbiamo scelto in precedenza.
A questo punto il nostro emulatore è in funzione, per cui non ci resta che lanciare la nostra applicazione che, come avevamo visto nelle diverse preview, dovrebbe semplicemente visualizzare il messaggio *Hello World*. Torniamo quindi in Android Studio e osserviamo la parte della barra degli strumenti (Figura 2.48), dove notiamo la presenza del nostro modulo di nome `app` e di un'opzione che si chiama *Edit Configurations*, che ci permetterà di definire alcune impostazioni custom da utilizzare in fase di esecuzione della nostra applicazione.
Nel caso di più applicazioni, avremmo avuto maggiori opzioni. Selezioniamo quindi il modulo di nome `app` e selezioniamo l'icona con la freccia verde rivolta a destra. A questo punto *Gradle* si mette in moto, eseguendo il *task* `install` che permette di eseguire il *build* completo dell'applicazione e quindi l'installazione nel dispositivo; nel nostro caso è rappresentato dall'emulatore avviato in precedenza, che comunque potrà essere selezionato attraverso l'interfaccia rappresentata nella Figura 2.49:

Figura 2.47 Emulatore del Nexus 6 in esecuzione.

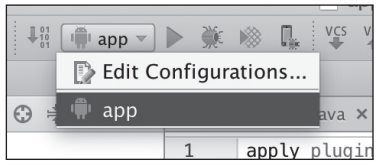

Figura 2.48 Eseguiamo la nostra applicazione.

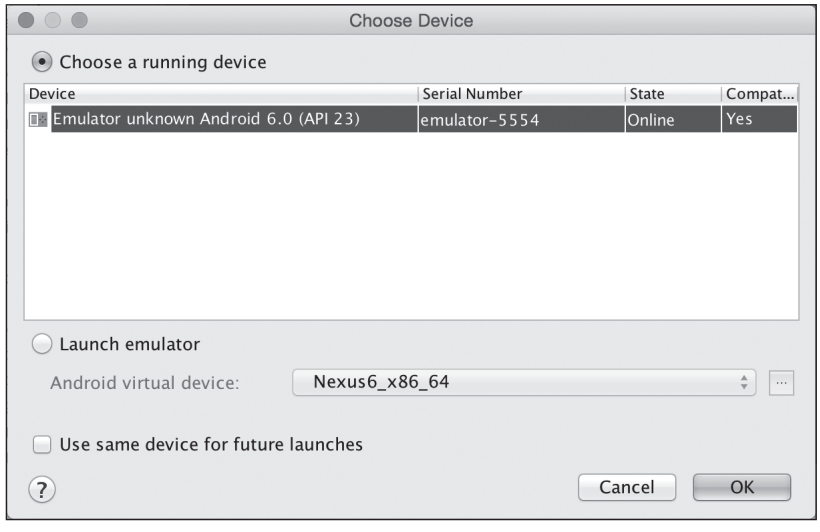

Figura 2.49 Selezione dell'AVD o dispositivo connesso per l'esecuzione dell'applicazione.

Selezionando il pulsante OK si ha finalmente l'esecuzione dell'applicazione, che apparirà nel nostro emulatore come indicato nella Figura 2.50.

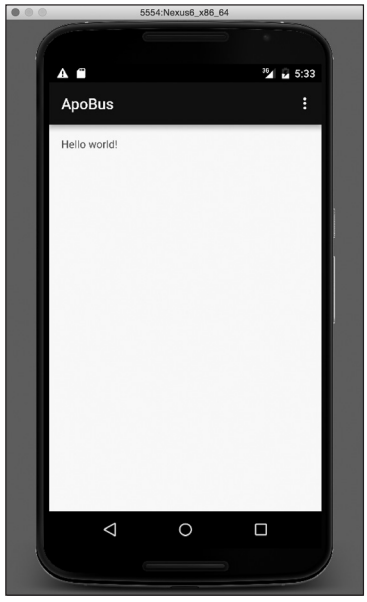

Figura 2.50 Applicazione in esecuzione nel nostro AVD.

Nella Figura 2.48 abbiamo visto la presenza di una voce di nome Edit Configurations, selezionando la quale si arriva a un'interfaccia (Figura 2.51) che contiene, nella parte destra, tre tab molto importanti.
Il primo si chiama General e permette di selezionare il modulo da eseguire, se utilizzare l'Activity principale definita nel file di configurazione AndroidManifest.xml e altre opzioni relative al particolare emulatore o dispositivo da utilizzare. Il secondo tab si chiama Emulator e permette (Figura 2.52) di specificare quale velocità di connessione simulare e il tempo di latenza. Attraverso delle checkbox è possibile abilitare o meno la cancellazione dei dati a ogni esecuzione oppure l'animazione del boot del dispositivo.
Infine il terzo tab si chiama *Logcat* e fa riferimento alle impostazioni di un tool che vedremo nel prossimo capitolo e che permette, appunto, la gestione del log dell'applicazione.

Esecuzione in un dispositivo reale

La possibilità di utilizzare un emulatore per le diverse versioni di dispositivi con caratteristiche differenti tra loro è sicuramente un aspetto molto importante, che però non può sostituire completamente il test su dispositivo reale. Per questo motivo diamo un breve cenno all'esecuzione della nostra applicazione in un dispositivo connesso al nostro PC attraverso un cavetto *USB*. Non entriamo negli aspetti relativi all'eventuale installazione di driver sulla nostra macchina, ma su quelle che sono le operazioni da fare sul nostro dispositivo. La prima cosa da fare riguarda l'abilitazione del menu *Developer Options* nell'applicazione delle impostazioni.

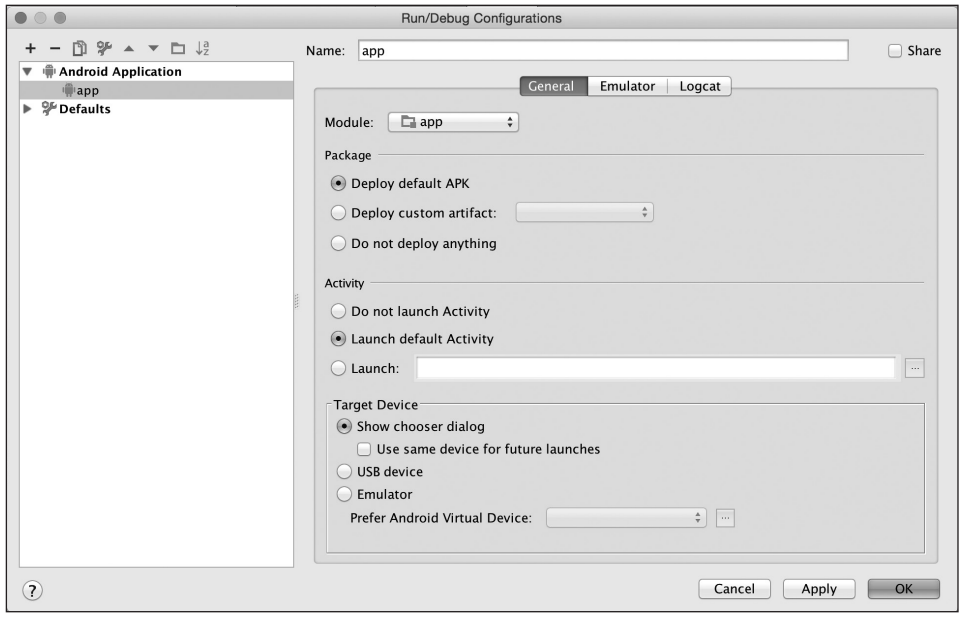

Figura 2.51 Personalizzazione per l'esecuzione di un'applicazione.

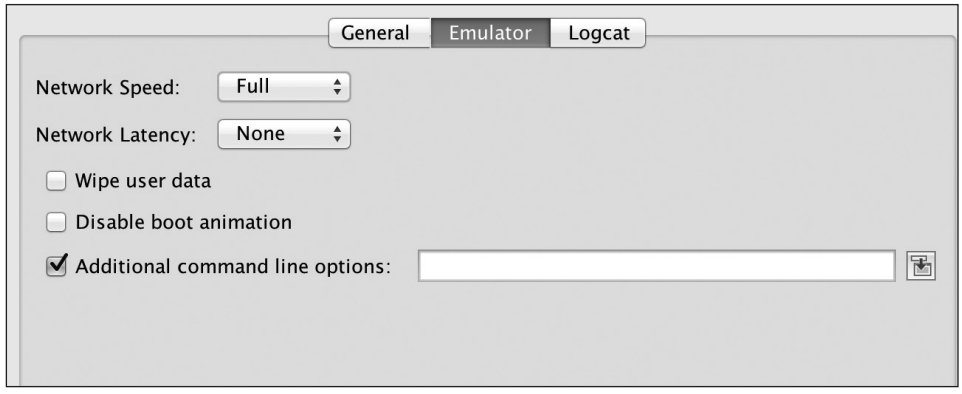

Figura 2.52 Impostazioni per l'emulatore.

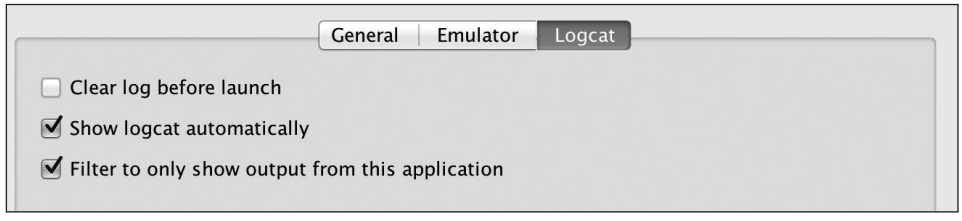

Figura 2.53 Impostazioni relative al log.

Si tratta di un menu che inizialmente è invisibile a cui si può accedere selezionando un certo numero di volte consecutive una delle opzioni già presenti, ovvero quella associata al *Build Number* nel menu corrispondente alla voce *About Phone*.

NOTA

Qualche dispositivo potrebbe avere una modalità di abilitazione diversa, per cui invitiamo il lettore a consultare la documentazione del proprio dispositivo.

A questo punto, nell'applicazione dei Settings, compare la voce *Developer Options*, che andiamo a selezionare. Il lettore potrà vedere moltissime configurazioni che andremo a esaminare più nel dettaglio quando parleremo di performance. Per il momento ci concentriamo su *Debugging*, che al momento è disabilitata come possiamo vedere nella Figura 2.54. Con il dispositivo collegato andiamo ad abilitare quella funzione, ottenendo la richiesta rappresentata nella Figura 2.55 che andiamo a confermare. A questo punto il dispositivo potrebbe chiedere un'ulteriore conferma in relazione al PC cui ci si collega, visualizzando il corrispondente *Mac Address*. Dopo aver accettato questa eventuale nuova richiesta, il nostro dispositivo è quasi pronto. Manca infatti un'ultima abilitazione, relativa alla possibilità di eseguire delle applicazioni di terze parti, ovvero applicazioni che non sono scaricate dal *Play Store*. A dire il vero non ci preoccupiamo di trovare questa opzione, che si trova all'interno del menu *Security*, in quando quando proveremo a eseguire la nostra applicazione ci verrà mostrata una finestra di dialogo che ci chiederà appunto di permettere l'esecuzione di questo tipo di applicazioni.

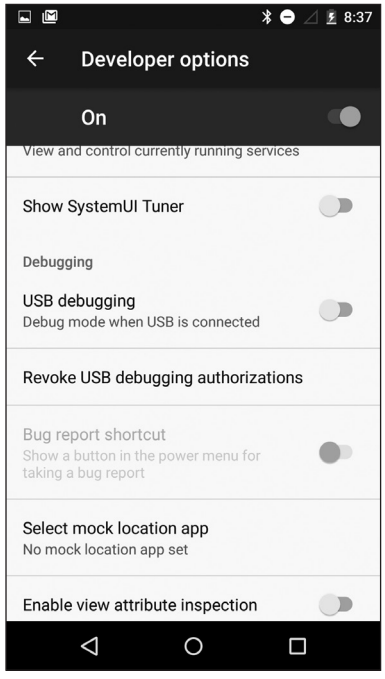

Figura 2.54 Opzione relativa al debug.

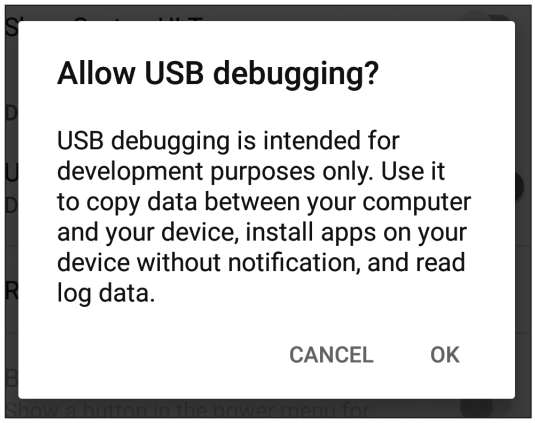

Figura 2.55 Conferma del debug via USB.

A questo punto il dispositivo è connesso e sarà visibile all'interno nella stessa finestra rappresentata nella Figura 2.49 insieme agli eventuali emulatori. Basterà selezionarlo ed eseguire l'applicazione che apparirà nel nostro dispositivo.

Conclusioni

Siamo giunti al termine di questo capitolo, che ci ha portato all'esecuzione della nostra prima applicazione *Android* senza scrivere alcuna riga di codice. Abbiamo infatti utilizzato il wizard di Android Studio e ci siamo soffermati su altri aspetti che sono fondamentali nella realizzazione di ogni applicazione *Android*. Dopo aver creato il progetto ci siamo soffermati sulla descrizione dei file di configurazione di *Gradle*, ovvero del tool di build che Google ha scelto e deciso di personalizzare. Abbiamo visto che cosa sia un *build type*, un *flavor*, una *build variant* e abbiamo imparato a gestirli all'interno del nostro *IDE*. In questa fase abbiamo anche visto come gestire le varie dipendenze. Si tratta di concetti che saranno utili anche nei prossimi capitoli.

Nella seconda parte abbiamo visto quali siano i ruoli delle parti fondamentali di un'applicazione, ovvero i sorgenti Java, le risorse e il file di configurazione `AndroidManifest.xml`. Si tratta delle componenti fondamentali di un'applicazione Android, che saranno trattati nei prossimi capitoli.

Abbiamo concluso il capitolo descrivendo quali siano i passi da seguire per l'esecuzione dell'applicazione in un emulatore (AVD) o in un dispositivo reale. Ora abbiamo tutto quello che ci serve per iniziare la nostra applicazione ApoBus.