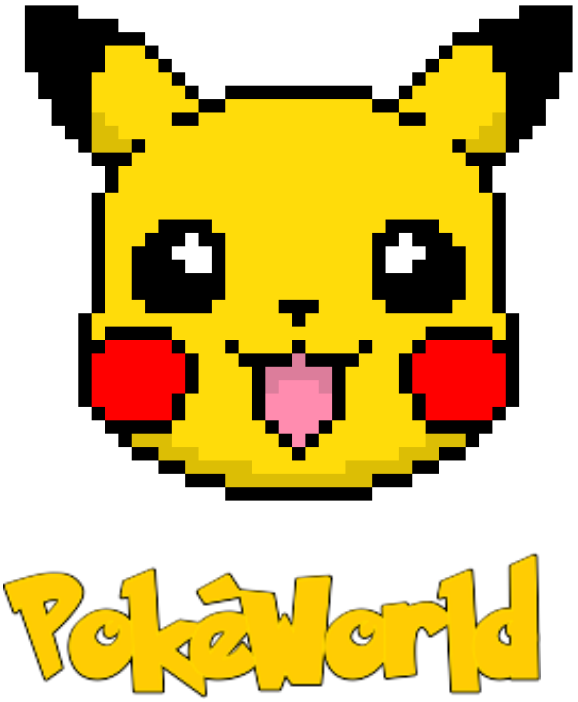


Progetto per il corso di Mobile Programming A.A. 2023/2024

Il team FALE

Cacace Elisa 0309987

Carloni Luca 0286148

Masci Francesco 0306888

Vassallo Alessandro 0295078

PokéWorld

1. Introduzione

L'applicazione "PokeWorld" è uno strumento innovativo pensato per tutti gli appassionati del mondo Pokémon. Essa contiene tutte le informazioni dettagliate su ogni Pokémon esistente, fornendo così agli utenti un accesso rapido e intuitivo a un vasto e completo dizionario del mondo Pokémon. Grazie a "PokeWorld", gli utenti possono esplorare le caratteristiche, le abilità, gli oggetti e altre informazioni essenziali riguardanti ciascun Pokémon, rendendo l'applicazione una risorsa indispensabile sia per i neofiti che per gli esperti del franchise. Questo progetto nasce dalla volontà di creare un'applicazione facile da usare, capace di raccogliere e presentare in modo organizzato e accattivante tutte le informazioni relative ai Pokémon, facilitando così la consultazione e l'apprendimento per gli appassionati di ogni età.

2. Requisiti di sistema

Per garantire il corretto funzionamento dell'applicazione, si raccomanda che i dispositivi soddisfino i seguenti requisiti minimi di sistema:

- **Sistema operativo**: L'applicazione è compatibile con dispositivi Android che eseguono la versione 9.0 (API 28) o successive.
- **Connessione a internet**: Sebbene non sia indispensabile, la disponibilità di una connessione internet consente di sfruttare appieno tutte le funzionalità offerte dall'applicazione.
- **Display**: Si consiglia l'utilizzo di un dispositivo con schermo di almeno 5 pollici, al fine di garantire una corretta visualizzazione degli elementi grafici e dell'interfaccia utente.
- **Memoria RAM**: È richiesta una memoria RAM minima di 2 GB per assicurare prestazioni fluide e una fruizione ottimale dell'applicazione.
- **Spazio di archiviazione**: Lo spazio di archiviazione minimo necessario è di 200 MB.

3. Tecnologie e strumenti di progettazione

Nel progetto sono state utilizzate diverse tecnologie e librerie le cui principali sono:

- `Kotlin`: Kotlin è stato il linguaggio di programmazione scelto per lo sviluppo dell'applicazione.

- `Android Studio`: è l'ambiente di sviluppo integrato (IDE) ufficiale per lo sviluppo di applicazioni Android. È stato utilizzato per scrivere, testare e eseguire il debug dell'applicazione, offrendo strumenti avanzati come l'editor di codice, emulatori e strumenti di monitoraggio delle prestazioni. La sua integrazione con Gradle ha facilitato la gestione delle dipendenze e il processo di build dell'applicazione.

- **Jetpack Compose**: il toolkit moderno di Google per la creazione di interfacce utente native su Android. Nel nostro progetto, è stato utilizzato per progettare e implementare l'interfaccia utente in modo dichiarativo, migliorando la leggibilità del codice e riducendo la complessità rispetto ai tradizionali layout XML.

- **Navigation Component**: il Navigation Component di Android Jetpack è stato utilizzato per gestire la navigazione all'interno dell'applicazione. Questo componente ha semplificato l'implementazione delle transizioni tra le varie schermate ed ha garantito un approccio centralizzato dell'applicazione.

- **Room Database**: è una libreria di persistenza di dati che fa parte di Android Jetpack. È stato utilizzato per gestire il database locale dell'applicazione. Room ha facilitato l'interazione con il database SQLite attraverso un'API di alto livello, garantendo al contempo la sicurezza e l'integrità dei dati tramite il controllo delle query e l'uso di entità e DAO (Data Access Object). Room consente inoltre la gestione delle diverse versioni del database, facilitando l'aggiornamento del database stesso per l'introduzione di nuove funzionalità o per l'aggiunta di nuovi Pokémon.

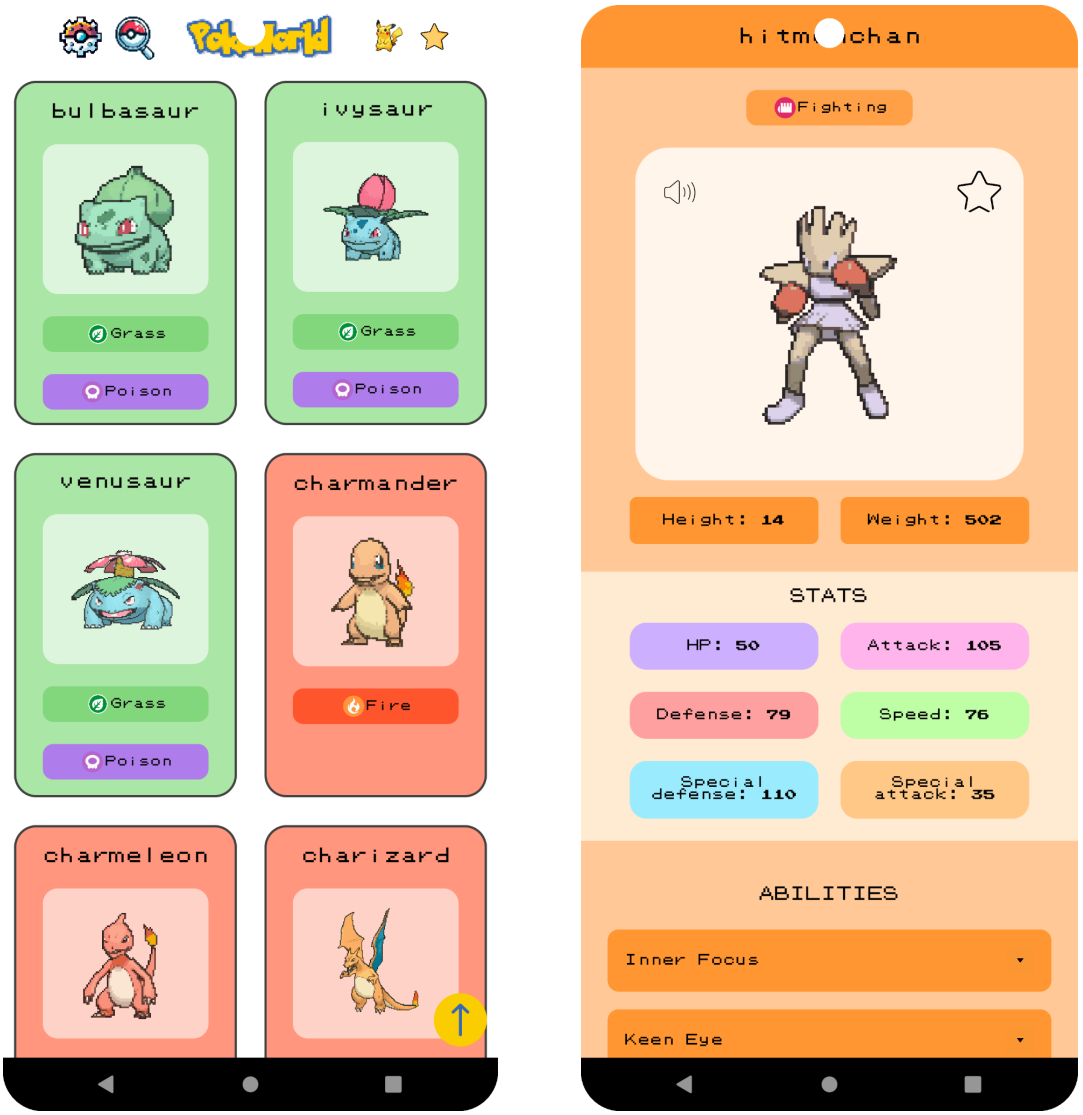
- **Kotlin Coroutines**: le Coroutines di Kotlin sono state utilizzate per gestire in modo efficiente le operazioni asincrone all'interno dell'applicazione, come l'interazione con il database.

- **Coil**: è una libreria moderna per il caricamento delle immagini, progettata specificamente per Android e integrata con Kotlin Coroutines. Coil è stato utilizzato per gestire il caricamento e la visualizzazione delle immagini dei Pokémon in modo efficiente e reattivo. Grazie a Coil, è stato possibile implementare il caricamento delle immagini senza bloccare l'interfaccia utente. La sua integrazione con Jetpack Compose ha ulteriormente semplificato la gestione delle immagini all'interno dell'interfaccia utente dichiarativa.

- **Gson**: è una libreria Kotlin, sviluppata da Google, molto apprezzata per la sua capacità di convertire oggetti Kotlin in JSON e viceversa. È stata impiegata per gestire la serializzazione e deserializzazione di liste di traduzioni all'interno del database.

4. Schermate

L'applicazione si articola in due schermate principali, interconnesse e complementari.
La prima schermata funge da indice, presentando una lista esaustiva di tutti i Pokémon, organizzati in modo intuitivo e visivamente accattivante.



L'interfaccia utente presentata è ispirata al mondo dei videogiochi classici e presenta una galleria dinamica di carte digitali, ciascuna dedicata a un Pokémon, che evoca l'estetica delle carte collezionabili fisiche.
Selezionando una card, l'utente viene reindirizzato alla seconda schermata, dedicata alla scheda dettagliata del Pokémon prescelto. L'articolazione delle due schermate garantisce un'esperienza utente fluida e intuitiva, permettendo all'utente di esplorare liberamente il Pokéverso.

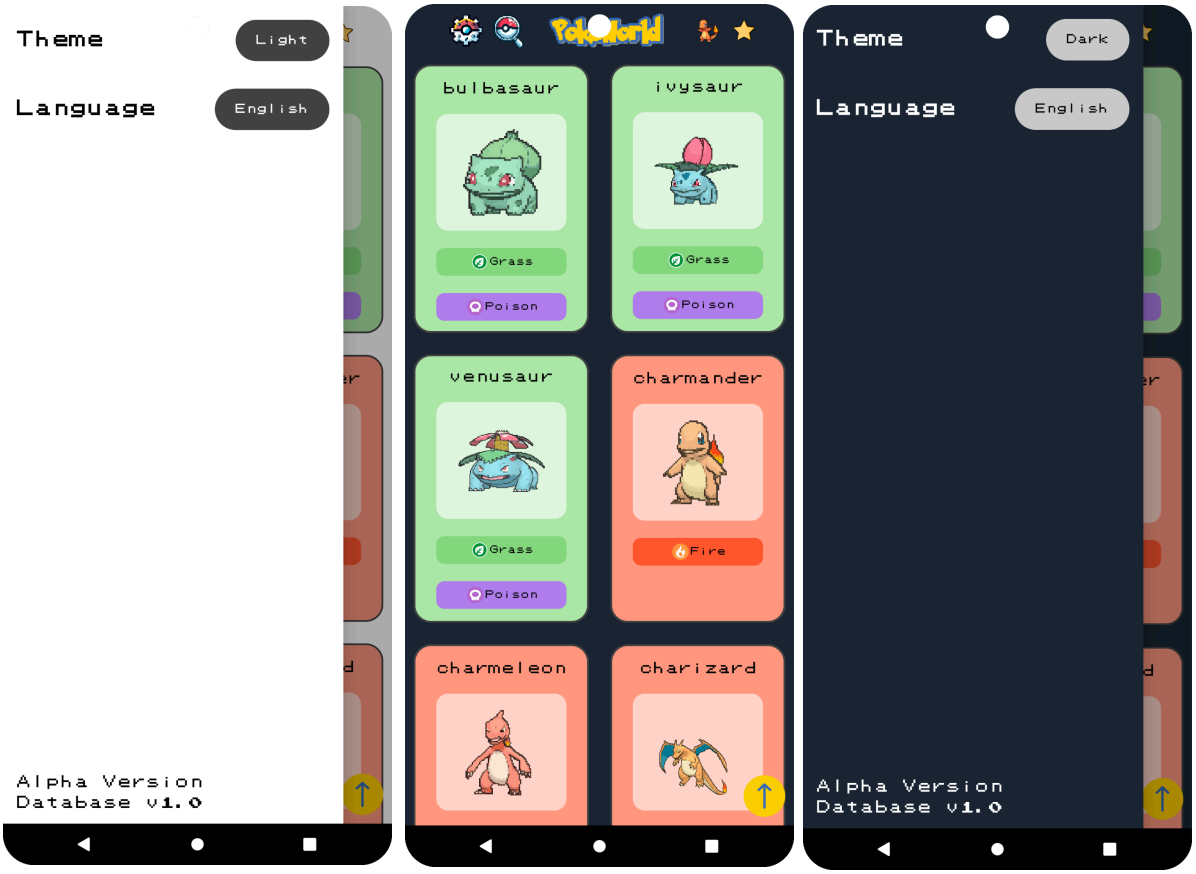
L'elemento grafico distintivo risiede nel design pixelato: i loghi, le icone, i testi e le animazioni GIF sono realizzati con una grafica pixelata, evocando i primi videogiochi e aggiungendo un tocco nostalgico. Il font che è stato utilizzato, chiamato Pokemon GB, replica il font utilizzato nei giochi originali Pokémon per il Game Boy.
Inoltre, per le immagini dei Pokémon da mostrare, si è scelto di utilizzare delle GIF pixelate fornite dalla PokéAPI.
Questa scelta stilistica non solo si allinea con il tema del franchise Pokémon, ma arricchisce l'esperienza utente, offrendo un richiamo alla nostalgia dei primi fan della serie.

4.1. Schermata principale

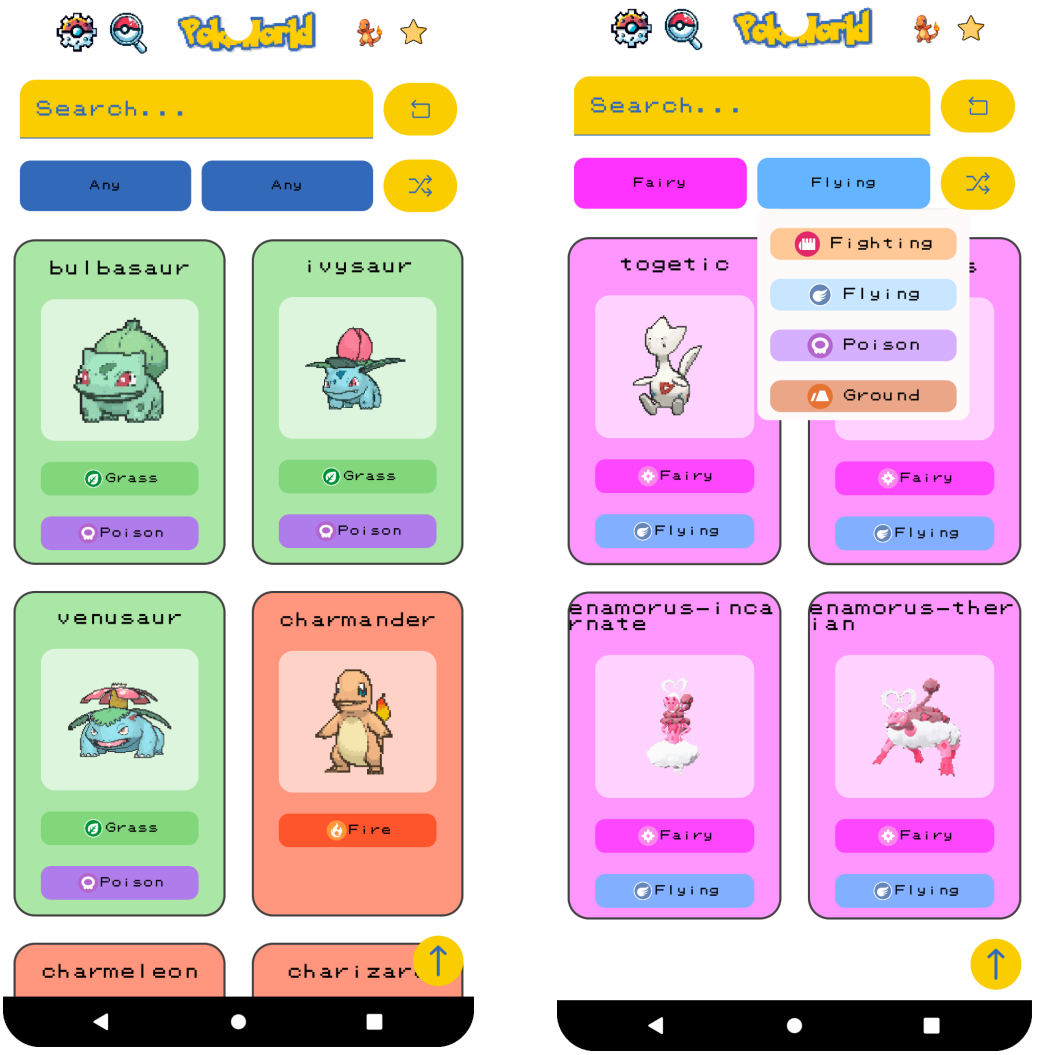
La schermata principale si presenta come una griglia dinamica di carte digitali, ciascuna dedicata a un Pokémon specifico.

Gli elementi che la compongono sono:

- la **lista Pokémon:** ogni elemento della lista è una “Pokémon Card”, che contiene il nome del Pokémon, una GIF animata che lo mostra, il suo tipo primario e, se presente, anche quello secondario. Di default la griglia mostra un minimo di due Card per riga, in funzione della dimensione dello schermo del dispositivo. Tuttavia, le colonne si adattano alla densità dei pixel, per cui per permettere una corretta visione anche sui dispositivi più piccoli, sarà visualizzata una sola Card per riga.
- l’**icona Back To Top:** siccome la lista dei Pokémon è molto lunga, questo tasto permette all’utente di ritornare facilmente all’inizio della lista.
- l’**icona Pokémon preferito:** mostra un’immagine statica del Pokémon correntemente selezionato come preferito; cliccando su di essa, si viene reindirizzati alla schermata di dettaglio di quel Pokémon.
- il **bottone Impostazioni:** cliccando su di esso, si aprirà la sezione delle impostazioni. Qui viene mostrata la versione dell’app e del database e viene permesso all’utente di cambiare la lingua o il tema dell’applicazione.



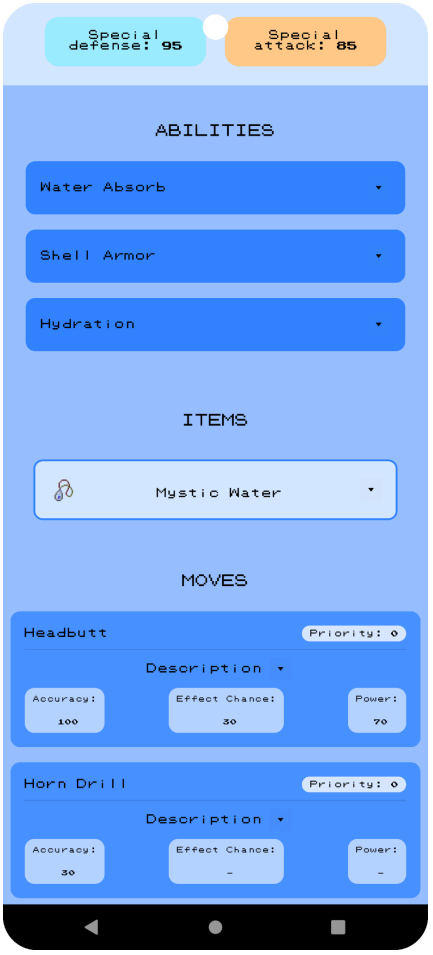
- il **bottone Ricerca e Filtraggio:** permette di espandere/collassare la sezione per la ricerca ed il filtraggio dei Pokémon in base a tipo primario e/o secondario; questa sezione contiene anche i tasti Random e Reset che servono, rispettivamente, per impostare casualmente i filtri sui tipi dei Pokémon e per resettare filtraggio/ricerca ripristinando la visione della lista completa dei Pokémon.



4.2. Schermata dettaglio

La schermata dedicata al dettaglio di un singolo Pokémon presenta un'interfaccia utente ricca di informazioni. Al centro della schermata, una GIF animata raffigura il Pokémon in stile pixelato. Insieme all'animazione, sono mostrate anche le informazioni di base, quali peso, altezza e tipo primario e, se presente, secondario. Due icone intuitive consentono all'utente di impostare/togliere il Pokémon come preferito e di riprodurre il suo caratteristico suono. Un'apposita sezione mostra le statistiche del Pokémon, offrendo una rapida panoramica delle sue abilità in combattimento. Le sezioni dedicate alle abilità, alle mosse e agli oggetti posseduti dal Pokémon completa il quadro, fornendo all'utente un approfondimento completo sulle caratteristiche e le potenzialità della creatura.
Ogni elemento in ciascuna sezione (abilità, oggetti, mosse) può essere espanso, così da mostrare informazioni aggiuntive.


Water
Ice
Height: 25
Weight: 2200
STATS
HP: 130
Attack: 85
Defense: 80
Speed: 60
Special defense: 95
Special attack: 85
ABILITIES
Water Absorb
Shell Armor


Special defense: 95
Special attack: 85
ABILITIES
Water Absorb
Shell Armor
Hydration
ITEMS
Mystic Water
MOVES
Headbutt
Priority: 0
Description
Accuracy: 100
Effect Chance: 30
Power: 70
Horn Drill
Priority: 0
Description
Accuracy: 30
Effect Chance: -
Power: -


ABILITIES
Water Absorb
Shell Armor
A hard shell protects the Pokémon from critical hits.
Hydration
ITEMS
Mystic Water
Cost: 1000
Effect: An item to be held by a Pokémon. This teardrop-shaped gem boosts the power of Water-type moves.
MOVES
Headbutt
Priority: 0
Description
The user sticks out its head and attacks by charging straight into the target. This may also make the target flinch.
Accuracy: 100
Effect Chance: 30
Power: 70

5. Navigazione

La struttura di navigazione dell'applicazione è stata realizzata utilizzando il componente `NavHost` di Jetpack Compose.

Il `NavHost` gestisce la navigazione tra le diverse schermate dell'applicazione.

Si definiscono due composable, "pokemon_list_screen" e "pokemon_details_screen", che rappresentano le schermate dell'applicazione e la destinazione iniziale viene impostata come "pokemon_list_screen".

La composable "pokemon_details_screen" riceve l'ID del Pokémon come argomento della route, consentendo di visualizzare i dettagli del Pokémon selezionato. Per definire gli argomenti che possono essere passati alla composable "pokemon_details_screen" viene utilizzato `navArgument`.

La funzione `onPokemonClicked` nella composable "pokemon_list_screen" naviga alla schermata dei dettagli del Pokémon selezionato utilizzando `navController.navigate`.

```
NavHost(
   navController = navController,
   startDestination = "pokemon_list_screen",
) {
   composable("pokemon_list_screen") {

       PokemonListScreen(
           onPokemonClicked = {
navController.navigate("pokemon_details_screen/$it") },
           pokemonListViewModel = viewModel(factory = factory),
           isDarkTheme = isDarkTheme,
           onThemeToggle = { isDark ->
               AppCompatDelegate.setDefaultNightMode(if(isDark)
AppCompatDelegate.MODE_NIGHT_YES else AppCompatDelegate.MODE_NIGHT_NO)
               isDarkTheme = isDark
           },
           language = language,
           onLanguageChange = { newLanguage ->
               language = newLanguage
           }
       )

   }
   composable(
       route = "pokemon_details_screen/{pokemonId}",
       arguments = listOf(navArgument("pokemonId") {
           type = NavType.IntType
       })
   ) {
       // ID del pokemon passato come argomento della route
       val pokemonId = it.arguments!!.getInt("pokemonId")
```

```
        PokemonDetailsScreen(
            pokemonDetailViewModel = viewModel(factory = factory),
            pokemonId = pokemonId
        )
    }
}
```

6. PokeAPI

Per lo sviluppo dell'applicazione, è stato necessario utilizzare le risorse disponibili su PokéAPI. Questo sito offre un servizio di API RESTful, che è stato impiegato per raccogliere tutte le informazioni essenziali per l'app. L'interfacciamento con le API non avviene direttamente tramite l'applicazione; al contrario, è stato sviluppato uno script Python esterno all'app, utilizzato esclusivamente in fase di sviluppo per raccogliere automaticamente i dati necessari e inserirli in un database appositamente progettato per rispecchiare la struttura dell'applicazione.

Lo script si occupa, inoltre, della conversione dei dati ottenuti dalle API nel formato richiesto dall'applicazione, adattandoli allo schema del database utilizzato. Questo database è stato successivamente integrato nella cartella "assets" del progetto Android, consentendo all'app di disporre di un database locale, che non richiede aggiornamenti tramite internet.

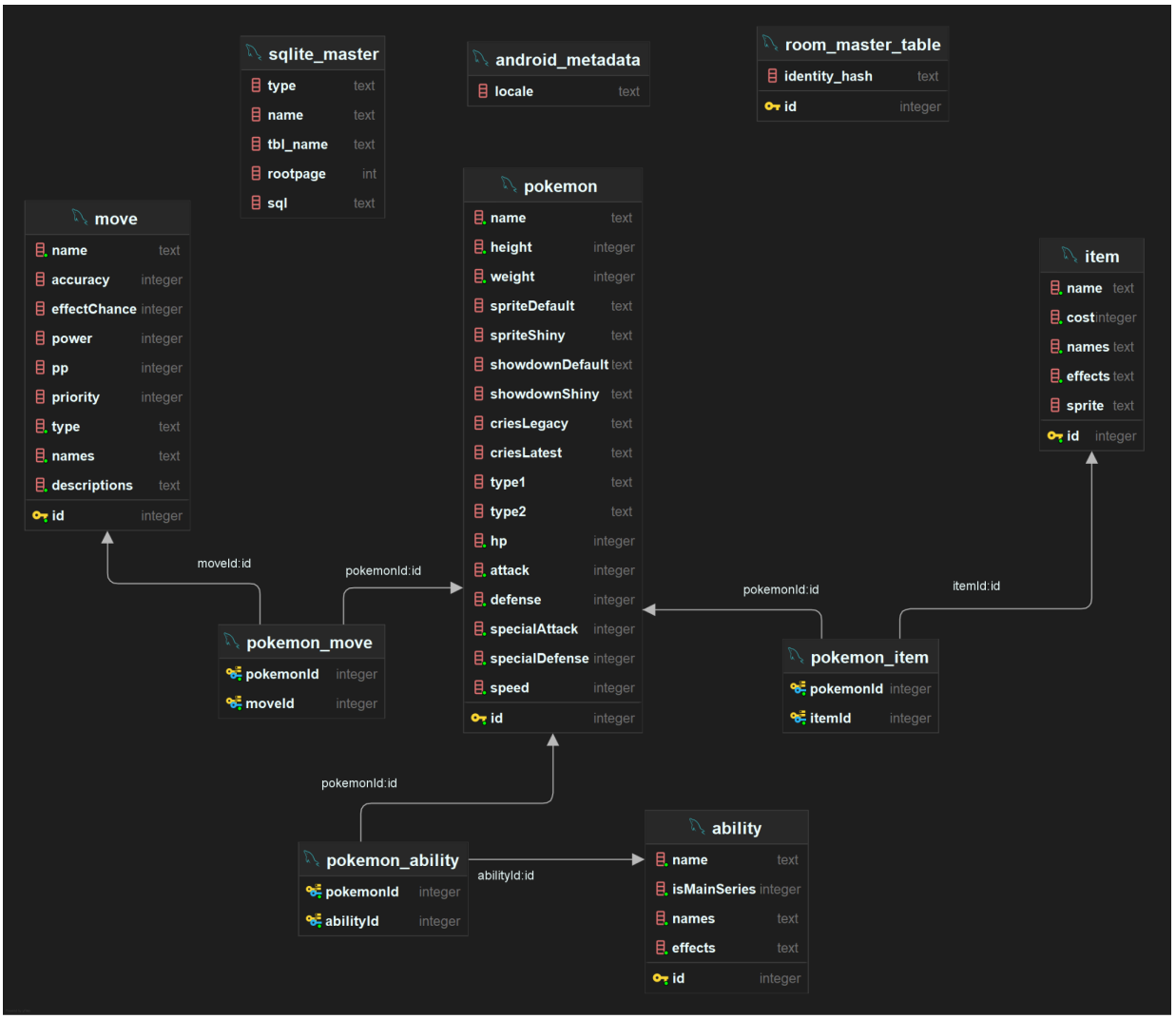
Tuttavia, l'applicazione continua a utilizzare le API per ottenere alcune risorse via internet, nello specifico:

- GIF e immagini rappresentanti i Pokémon.
- Suoni emessi dai Pokémon.

La disponibilità di una connessione internet, pertanto, condiziona la capacità dell'applicazione di visualizzare immagini e riprodurre suoni dei vari Pokémon. Per mitigare questa limitazione, è stata utilizzata la funzione `AsyncImage` della libreria `Coil`, che permette una gestione automatica della cache. In questo modo, le immagini mostrate nell'interfaccia utente e i suoni riprodotti vengono memorizzati nella cache, evitando così la necessità di scaricarli nuovamente. Tale soluzione è stata adottata proprio per garantire un'efficiente gestione della cache.

7. Database

La creazione del database avviene al primo avvio dell'applicazione, poiché una copia di esso, contenente tutti i dati necessari sui Pokémon, viene inclusa come asset all'interno del bundle APK. Al momento del primo avvio, Room si occupa di creare una copia locale del database a partire da quello fornito.



7.1. Database

La classe deputata alla gestione del database è implementata come singleton e contiene i metodi per accedere al DAO relativo, necessari per l'esecuzione delle operazioni sul database.
Nella definizione del database vengono specificate le entità da rappresentare al suo interno, insieme alla versione del database. Per farlo si utilizza l'annotazione `@Database` fornita proprio dalla libreria.

```
/**
 * Classe per gestire il database locale.
 */
@Database(entities = [
        PokemonEntity::class,
        AbilityEntity::class,
        PokemonAbilityCross::class,
        ItemEntity::class,
        PokemonItemCross::class,
        MoveEntity::class,
```

```
        PokemonMoveCross::class
    ],
    version = 1,
    exportSchema = false
)
abstract class PokemonDatabase: RoomDatabase() {

    companion object {

        // Singola istanza del database
        private var db: PokemonDatabase? = null

        /**
         * Ottiene l'istanza del database.
         *
         * @param context Context dell'applicazione.
         * @return L'istanza del database.
         */
        fun getInstance(context: Context): PokemonDatabase {
            if (db == null) {
                Log.d("PokemonDatabase", "Creating new database instance")
                db = Room.databaseBuilder(
                    context.applicationContext,
                    PokemonDatabase::class.java,
                    "pokemon.db"
                )
                    .createFromAsset("pokemon.db")
                    .fallbackToDestructiveMigration()
                    .build()
            }
            return db as PokemonDatabase
        }
    }

    /**
     * Ottiene il DAO per le operazioni di database.
     *
     * @return Il DAO per le operazioni di database.
     */
    abstract fun pokemonDao(): PokemonDao
}
```

7.2. Entità

Ogni tabella all'interno del database corrisponde a una specifica entità. Per consentire la creazione della tabella, ogni classe di un'entità viene annotata con `@Entity` e in essa vengono specificate la chiave primaria e ulteriori dettagli, come il nome della tabella corrispondente.

In particolare, i campi che possono assumere valori nulli sono automaticamente marcati come *nullable* nel database. Le varie colonne vengono definite a partire dagli attributi della classe, con l'eccezione di quelli contrassegnati con l'annotazione `@Ignore`.

```
/**
 * Classe che rappresenta un Pokemon
 */
@Entity(tableName = "pokemon")
data class PokemonEntity(

   /* --- GENERAL --- */

   @PrimaryKey val id: Int,
   val name: String,

   // Altri parametri

) {

   /* --- ABILITIES --- */

   @Ignore
   var abilities: List<AbilityEntity> = emptyList()

   // Altri metodi e attributi della classe

}
```

L'implementazione completa viene mostrata al paragrafo Model/Pokèmon

Per tali attributi, non è prevista una rappresentazione diretta nel database; in questi casi, si ricorre all'uso di tabelle di associazione (*cross table*) per collegare l'entità del Pokémon con le entità relative a oggetti, mosse e abilità. Attraverso l'utilizzo di entità di tipo `Cross`, viene creata una tabella che consente di stabilire una relazione *molti-a-molti* tra i Pokémon e le risorse a essi collegate. Nella definizione di tali tabelle vengono specificati gli indici e le chiavi esterne mediante l'impiego delle annotazioni appropriate.

```
/**
 * Classe che rappresenta una abilità di un Pokemon
 */
@Entity(tableName = "ability")
@TypeConverters(TranslationListConverter::class)
data class AbilityEntity(
   @PrimaryKey val id: Int,
   val name: String,
   val names: List<Translation> = emptyList()

   // Altri parametri

) {
```

```
    // Operazioni della classe

}
```

L'implementazione completa viene mostrata al paragrafo Model/Risorse

```
/**
 * Classe che rappresenta la relazione tra un Pokemon e una abilità
 *
 * @param pokemonId ID del Pokemon
 * @param abilityId ID dell'abilità
 */
@Entity(
    tableName = "pokemon_ability",
    primaryKeys = ["pokemonId", "abilityId"],
    foreignKeys = [
        ForeignKey(
            entity = PokemonEntity::class,
            parentColumns = ["id"],
            childColumns = ["pokemonId"],
            onDelete = ForeignKey.CASCADE,
            onUpdate = ForeignKey.CASCADE
        ),
        ForeignKey(
            entity = AbilityEntity::class,
            parentColumns = ["id"],
            childColumns = ["abilityId"],
            onDelete = ForeignKey.CASCADE,
            onUpdate = ForeignKey.CASCADE
        )
    ],
    indices = [Index(value = ["pokemonId"]), Index(value = ["abilityId"])]
)
data class PokemonAbilityCross(
    val pokemonId: Int,
    val abilityId: Int
)
```

Le risorse contengono testi e descrizioni in diverse lingue. Per garantire una corretta gestione delle traduzioni, si è scelto di rappresentare tali traduzioni nel database attraverso una classe di supporto denominata `Translation`, che consente di memorizzare una stringa di testo per ciascuna lingua specifica, identificata attraverso il relativo codice. La lista di traduzioni per ogni risorsa viene salvata nel database in formato *JSON*, grazie all'uso di una classe di conversione che sfrutta la libreria `Gson`.

```
/**
 * Classe per contenere la traduzione di un testo specifico
 *
```

```
 * @param language La lingua del testo
 * @param text Il testo in quella lingua
 */
data class Translation(val language: String, val text: String)

/**
* Classe per la conversione tra una lista di oggetti Translation e una stringa
JSON.
*/
class TranslationListConverter {

    /**
     * Converte una lista di oggetti Translation in una stringa JSON.
     *
     * @param translations La lista di oggetti Translation da convertire.
     * @return Una stringa JSON che rappresenta la lista di oggetti Translation.
     */
    @TypeConverter
    fun fromTranslationList(translations: List<Translation>): String {
        val gson = Gson()
        return gson.toJson(translations)
    }

    /**
     * Converte una stringa JSON in una lista di oggetti Translation.
     *
     * @param data La stringa JSON da convertire.
     * @return Una lista di oggetti Translation che rappresenta la stringa JSON.
     */
    @TypeConverter
    fun toTranslationList(data: String): List<Translation> {
        val listType = object : TypeToken<List<Translation>>() {}.type
        return Gson().fromJson(data, listType)
    }

}
```

Le altre risorse sono strutturate in modo analogo.

7.3. DAO

Una volta creato il database locale, l'accesso ai dati avviene tramite il DAO (Data Access Object), all'interno del quale sono definite le operazioni da eseguire. Il DAO viene implementato come interfaccia, mentre la generazione della classe concreta che ne realizza l'implementazione avviene durante la fase di compilazione, a cura di `Room`. In questa fase, `Room` specifica il corpo delle operazioni descritte, traducendo le richieste in istruzioni concrete.

Nella definizione del DAO, viene anche fornito il riferimento alla classe responsabile della conversione tra i diversi tipi di dati. In particolare, viene passato il riferimento per la conversione che consente di rappresentare nel database un tipo di Pokémon, associandolo alla relativa enumerazione (enum) che rappresenta tale informazione nell'applicazione.

```
/**
 * Interfaccia per l'accesso al database
 */
@Dao
@TypeConverters(PokemonTypeConverter::class)
interface PokemonDao {
      // Corpo del DAO
}
```

La classe di conversione definisce due metodi principali: il primo metodo è responsabile della serializzazione dell'enumerazione in un formato compatibile con il database, mentre il secondo metodo consente di ottenere un valore dell'enumerazione a partire dalla sua rappresentazione nel database.
Nel dettaglio, nel database viene memorizzata una stringa corrispondente all'attributo type dell'enumerazione, che permette di effettuare la conversione.

```
/**
 * Converte un tipo di Pokemon in una stringa
 */
class PokemonTypeConverter {

   /**
    * Converte un tipo di Pokemon in una stringa
    *
    * @param type Il tipo di Pokemon da convertire
    *
    * @return La stringa corrispondente al tipo di Pokemon
    */
   @TypeConverter
   fun fromPokemonType(type: PokemonType?): String? {
       return type?.type
   }

   /**
    * Converte una stringa in un tipo di Pokemon
    *
    * @param type La stringa da convertire
    *
    * @return Il tipo di Pokemon corrispondente alla stringa
    */
   @TypeConverter
   fun toPokemonType(type: String): PokemonType? {
       return PokemonType.fromType(type)
   }
```

```
}
```

La prima operazione definita nel DAO è una funzione che permette di ottenere la lista completa di tutti i Pokémon presenti nel database. Tale operazione specifica la query SQL da eseguire, e Room si occupa delle conversioni necessarie, avvalendosi eventualmente delle classi di conversione fornite.

```
/**
 * Ottiene la lista di tutti i Pokemon
 *
 * @return Lista di Pokemon
 */
@Query("SELECT * FROM pokemon")
suspend fun retrievePokemonList(): List<PokemonEntity>
```

La seconda operazione utilizza l'ID del Pokémon per ottenere le informazioni relative al singolo Pokémon. In particolare, l'entità del Pokémon viene restituita priva delle informazioni riguardanti mosse, oggetti e abilità.

```
/**
 * Ottiene un Pokemon tramite il suo ID
 *
 * @param pokemonId ID del Pokemon
 * @return Il Pokemon con l'ID specificato
 */
@Query("SELECT * FROM pokemon WHERE id = :pokemonId")
suspend fun retrievePokemon(pokemonId: Int): PokemonEntity?
```

Per reperire queste informazioni aggiuntive, vengono definite operazioni specifiche che, prendendo come argomento l'ID del Pokémon, restituiscono le risorse collegate.

```
/**
 * Ottiene le abilità di un Pokemon tramite il suo ID
 *
 * @param pokemonId ID del Pokemon
 * @return Lista di abilità del Pokemon
 */
@Transaction
@Query(
    """
    SELECT ability.*
    FROM ability
    INNER JOIN pokemon_ability ON ability.id = pokemon_ability.abilityId
    WHERE pokemon_ability.pokemonId= :pokemonId
    """
)
```

```
suspend fun retrieveAbilitiesForPokemon(pokemonId: Int): List<AbilityEntity>

/**
 * Ottiene gli item di un Pokemon tramite il suo ID
 *
 * @param pokemonId ID del Pokemon
 * @return Lista di item del Pokemon
 */
@Transaction
@Query(
    """
    SELECT item.*
    FROM item
    INNER JOIN pokemon_item ON item.id = pokemon_item.itemId
    WHERE pokemon_item.pokemonId= :pokemonId
    """
)
suspend fun retrieveItemsForPokemon(pokemonId: Int): List<ItemEntity>

/**
 * Ottiene le mosse di un Pokemon tramite il suo ID
 *
 * @param pokemonId ID del Pokemon
 * @return Lista di mosse del Pokemon
 */
@Transaction
@Query(
    """
    SELECT move.*
    FROM move
    INNER JOIN pokemon_move ON move.id = pokemon_move.moveId
    WHERE pokemon_move.pokemonId= :pokemonId
    """
)
suspend fun retrieveMovesForPokemon(pokemonId: Int): List<MoveEntity>
```

L'adozione di Room come libreria per la gestione dello strato di persistenza consente di delegare ad essa la responsabilità della gestione dei metodi per l'accesso al database e la serializzazione dei dati. Di conseguenza, i dati ottenuti mediante l'invocazione dei vari metodi vengono automaticamente convertiti nel formato utilizzato dall'applicazione, semplificando così il processo di manipolazione e utilizzo delle informazioni.

8. Repositories

L'accesso ai dati dell'applicazione non avviene direttamente, ma tramite l'utilizzo dei repository. Questo approccio consente di nascondere alle varie classi che richiedono i dati le operazioni necessarie per interagire con lo strato di persistenza. Nello specifico, sono stati implementati tre differenti repository per la gestione dei dati, ciascuno con uno scopo specifico.

8.1. Repository Pokèmon

Il repository dei Pokémon, realizzato come interfaccia attraverso il pattern `Repository`, definisce un contratto ben preciso per tutte le operazioni di lettura dei dati relativi ai Pokémon. Questa astrazione permette di sostituire facilmente l'implementazione sottostante (ad esempio, passando da un database locale a un servizio cloud) senza impattare le altre parti dell'applicazione.
Al momento della creazione, il repository dei Pokémon riceve in input un DAO che si occupa di interagire direttamente con la persistenza utilizzata. Il DAO espone un insieme di metodi di basso livello per eseguire query, mentre il repository si concentra su un'interfaccia più ad alto livello, offrendo metodi come `retrievePokemonList` o `retrievePokemon`, che possono realizzare logiche più complesse.

```
/**
 * Interfaccia per le operazioni di accesso ai dati dei pokemon.
 */
interface PokemonRepository {

   companion object {

       // Flag per specificare quali informazioni devono essere caricate nel
dettaglio del pokemon
       const val LOAD_ABILITIES = 0b001
       const val LOAD_MOVES = 0b010
       const val LOAD_ITEMS = 0b100
       const val LOAD_ALL = 0b111
   }

   /**
    * Ottiene la lista di tutti i pokemon presenti nel database.
    *
    * @return Lista di pokemon.
    */
   suspend fun retrievePokemonList(): List<PokemonEntity>

   /**
    * Ottiene il pokemon con l'ID specificato.
    *
    * @param pokemonId ID del pokemon.
    * @param load Flag per specificare quali informazioni devono essere caricate
nel dettaglio del pokemon. I possibili valori sono:
    * LOAD_ABILITIES, LOAD_MOVES, LOAD_ITEMS, LOAD_ALL (o loro combinazioni
tramite OR).
    *
    * @return Pokemon con l'ID specificato.
    */
   suspend fun retrievePokemon(pokemonId: Int, load: Int = 0): PokemonEntity?

}
```

Il primo metodo permette di ottenere la lista di tutti i Pokémon presenti all'interno del database. Questo metodo, nell'implementazione del progetto, inoltra semplicemente la richiesta al corrispondente metodo del DAO, restituendone il risultato senza ulteriori modifiche.

```
/**
 * Ottiene la lista di tutti i pokemon presenti nel database.
 *
 * @return Lista di pokemon.
 */
override suspend fun retrievePokemonList(): List<PokemonEntity> {
   Log.d("PokemonRepository", "Retrieving pokemon list")
   return dao.retrievePokemonList()
}
```

Il secondo metodo consente di ottenere un Pokémon specifico, cercato a partire dal suo ID. È possibile specificare un parametro che indica quali risorse collegate al Pokémon debbano essere caricate. Di default, nessuna risorsa viene caricata. Questa funzionalità è implementata tramite un meccanismo di flag bitwise, che permette di combinare in modo flessibile le diverse opzioni, utilizzando costanti specifiche definite all'interno dall'interfaccia.

```
/**
 * Ottiene il pokemon con l'ID specificato.
 *
 * @param pokemonId ID del pokemon.
 * @param load Flag per specificare quali informazioni devono essere caricate
nel dettaglio del pokemon. I possibili valori sono:
 * LOAD_ABILITIES, LOAD_MOVES, LOAD_ITEMS, LOAD_ALL (o loro combinazioni tramite
OR).
 *
 * @return Pokemon con l'ID specificato.
 */
override suspend fun retrievePokemon(pokemonId: Int, load: Int): PokemonEntity?
{
   Log.d("PokemonRepository", "Retrieving pokemon with id $pokemonId")
   val p = dao.retrievePokemon(pokemonId)
   if(p != null) {
       if (load and PokemonRepository.LOAD_ABILITIES != 0) p.abilities =
dao.retrieveAbilitiesForPokemon(pokemonId)
       if (load and PokemonRepository.LOAD_MOVES != 0) p.moves =
dao.retrieveMovesForPokemon(pokemonId)
       if (load and PokemonRepository.LOAD_ITEMS != 0) p.items =
dao.retrieveItemsForPokemon(pokemonId)
   }
   return p
}
```

8.2. Repository preferenze dell'utente

L'accesso alle preferenze utente è centralizzato tramite un pattern `Singleton` implementato in Kotlin. Questo design pattern garantisce un unico punto di accesso globale alle impostazioni dell'utente, semplificando la gestione e prevenendo incoerenze.
Il repository delle preferenze sfrutta le `SharedPreferences` di Android per persistere le scelte dell'utente. L'intera interazione con il meccanismo di memorizzazione è incapsulata all'interno di questo componente, che si occupa sia della scrittura che della lettura dei dati.
L'inizializzazione del repository viene effettuata all'avvio dell'applicazione, fornendo al Singleton il contesto dell'applicazione. Il contesto viene utilizzato per ottenere un riferimento al gestore delle `SharedPreferences`, che sarà successivamente impiegato per interagire con lo storage sottostante.

```
/**
 * Classe principale dell'applicazione.
 */
@AndroidEntryPoint
class PokeWorld : ComponentActivity(){

   override fun onCreate(savedInstanceState: Bundle?){
       super.onCreate(savedInstanceState)

       /* --- Inizializzazione --- */

       UserPreferencesRepository.initialize(this)

       // Resto del metodo

}


/**
 * Repository per le preferenze dell'utente.
 */
object UserPreferencesRepository {

   // Variabili per la gestione delle preferenze
   private lateinit var sharedPreferences: SharedPreferences

   /**
    * Inizializza il repository di preferenze.
    */
   fun initialize(context: Context) {
       sharedPreferences = context.getSharedPreferences(PREFERENCES_FILE_NAME,
Context.MODE_PRIVATE)
   }

       // Altre operazioni dell'oggetto
```

```
}
```

Per ciascuna preferenza configurabile, il repository espone due metodi: uno per impostare il valore della preferenza e un altro per recuperarlo.

```
/**
 * Repository per le preferenze dell'utente.
 */
object UserPreferencesRepository {

    // Inizializzazione

    /**
     * Salva la preferenza di tema in SharedPreferences.
     *
     * @param isDarkTheme Indica se il tema è scuro o meno.
     */
    fun saveDarkModePreference(isDarkTheme: Boolean) {
        sharedPreferences.edit().putBoolean(PREFS_THEME_KEY, isDarkTheme).apply()
    }

    /**
     * Ottiene la preferenza di tema da SharedPreferences.
     *
     * @param defValue Valore di default da utilizzare se non viene trovato
nulla.
     * @return True se il tema è scuro, false altrimenti.
     */
    fun getDarkModePreference(defValue: Boolean = false): Boolean {
        return sharedPreferences.getBoolean(PREFS_THEME_KEY, defValue)
    }

    // Altre operazioni dell'oggetto

}
```

8.3. Repository Pokèmon preferito

Per garantire un accesso coerente e indipendente alle informazioni relative al Pokémon preferito da parte di schermate diverse, è stato introdotto un repository intermedio. Questo componente agisce come uno strato di astrazione, evitando un accoppiamento diretto tra le due schermate e centralizzando la gestione dei dati.

Il repository del Pokémon preferito, implementato come `Singleton`, è stato progettato per contenere esclusivamente un oggetto di tipo `MutableStateFlow`. Questo flusso di stato è utilizzato per memorizzare i dati del Pokémon preferito e per notificare automaticamente le eventuali modifiche a tutti gli osservatori, tra cui le schermate interessate.

La schermata dei dettagli del Pokémon è responsabile di caricare le informazioni del Pokémon preferito selezionato nel repository, aggiornando così il `MutableStateFlow`. Contemporaneamente, la schermata principale della lista dei Pokémon osserva lo stesso `StateFlow` per visualizzare in tempo reale il Pokémon preferito attualmente impostato.

```
/**
 * Repository per la gestione del pokemon preferito
 */
object FavoritePokemonSharedRepository {

   // Variabile per memorizzare il pokemon preferito
   private val _sharedFavoritePokemon = MutableStateFlow<PokemonEntity?>(null)
   val sharedFavoritePokemon: StateFlow<PokemonEntity?>
       get() = _sharedFavoritePokemon.asStateFlow()

   /**
    * Aggiorna il pokemon preferito.
    *
    * @param pokemon Il nuovo pokemon preferito
    */
   fun updateFavoritePokemon(pokemon: PokemonEntity?) {
       _sharedFavoritePokemon.value = pokemon
   }

}
```

Per semplicità e per evitare incongruenze, il repository del Pokémon preferito è stato implementato come volatile, ovvero senza alcuna persistenza su disco. I dati vengono mantenuti esclusivamente in memoria RAM. Di conseguenza, all'avvio dell'applicazione, l'`Activity` principale ha il compito di caricare il Pokémon preferito recuperando il suo ID dalle `SharedPreferences` (attraverso il repository delle preferenze) e di aggiornarlo nel repository condiviso andando a recuperare le sue informazioni tramite il repository dei Pokémon.

```
/**
 * Classe principale dell'applicazione.
 */
@AndroidEntryPoint
class PokeWorld : ComponentActivity(){

   override fun onCreate(savedInstanceState: Bundle?){
       super.onCreate(savedInstanceState)

       /* --- Inizializzazione --- */

       UserPreferencesRepository.initialize(this)
```

```
        val pokemonRepository: PokemonRepository =
PokemonRepositoryImpl(PokemonDatabase.getInstance(applicationContext).po
kemonDao())
        loadFavoritePokemon(pokemonRepository)

        // Resto del metodo

    }

    /**
     * Carica il pokemon preferito salvato nelle SharedPreferences
     * nel repository condiviso per essere utilizzato da più ViewModel.
     *
     * @param pokemonRepository Repository per le operazioni di database.
     */
    private fun loadFavoritePokemon(pokemonRepository: PokemonRepository)
{

        // Carica l'id del pokemon preferito
        val favoriteId: Int? =
UserPreferencesRepository.getFavoritePokemonId()

        // Imposta il pokemon preferito nel repository condiviso
        CoroutineScope(Dispatchers.IO).launch {
            val favPokemon: PokemonEntity? =
pokemonRepository.retrievePokemon(favoriteId ?: 0)

FavoritePokemonSharedRepository.updateFavoritePokemon(favPokemon)
        }

    }

}
```

9. Struttura: MVVM

L'architettura adottata nel progetto è quella MVVM (Model-View-ViewModel), una delle architetture più popolari e raccomandate per lo sviluppo di applicazioni Android moderne. MVVM separa la logica di business e l'interfaccia utente, migliorando la chiarezza e la manutenzione del codice.

9.1. Model

Il Model rappresenta la logica di business e i dati dell'applicazione. In questa parte dell'architettura, vengono definiti i dati e le operazioni che coinvolgono il recupero e la gestione delle informazioni.

9.1.1. Pokemon

La classe `PokemonEntity` rappresenta un Pokémon all'interno del modello di dati dell'applicazione. Essa è annotata con `@Entity`, indicando che è mappata a una tabella del database denominata *'pokemon'*.
I campi della classe corrispondono alle proprietà di un Pokémon, come l'ID, il nome, l'altezza, il peso, i tipi, le statistiche e i link agli sprite. I campi `abilities`, `items` e `moves` sono annotati con `@Ignore`, poiché non vengono direttamente persistenziati nel database ma vengono caricati in modo lazy quando necessari.
La classe fornisce due metodi per ottenere l'URL dell'immagine del Pokémon: `getAnimatedImageUrl()` e `getImageUrl()`. Il primo metodo restituisce l'URL dell'immagine animata, se disponibile, mentre il secondo restituisce l'URL dell'immagine statica. Questa distinzione è utile per visualizzare il Pokémon in modo dinamico o statico, a seconda delle esigenze dell'interfaccia utente.

```
/**
 * Classe che rappresenta un Pokemon
 */
@Entity(tableName = "pokemon")
data class PokemonEntity(

    /* --- GENERAL --- */

    @PrimaryKey val id: Int,
    val name: String,
    val height: Int = 0,
    val weight: Int = 0,

    /* --- SPRITES --- */
    // Non contiene direttamente gli sprites ma contiene i link

    val spriteDefault: String? = null,
    val spriteShiny: String? = null,

    val showdownDefault: String? = null,
    val showdownShiny: String? = null,

    val criesLegacy: String? = null,
    val criesLatest: String? = null,

    /* --- TYPES --- */

    val type1: PokemonType? = null,
    val type2: PokemonType? = null,

    /* --- STATS --- */

    val hp: Int = 0,
    val attack: Int = 0,
```

```
    val defense: Int = 0,
    val specialAttack: Int = 0,
    val specialDefense: Int = 0,
    val speed: Int = 0

) {

    /* --- ABILITIES --- */

    @Ignore
    var abilities: List<AbilityEntity> = emptyList()

    /* --- ITEMS --- */

    @Ignore
    var items: List<ItemEntity> = emptyList()

    /* --- MOVES --- */

    @Ignore
    var moves: List<MoveEntity> = emptyList()

    /**
     * Ottiene l'url dell'immagine dinamica da mostrare
     *
     * @return l'url dell'immagine dinamica da mostrare
     */
    fun getAnimatedImageUrl(): String? {
        // Controlla se ci sia un'immagina dinamica
        if(showdownDefault != null) return showdownDefault
        // Se non esiste viene ritornata un'immagina statica
        return getImageUrl()
    }

    /**
     * Ottiene l'url dell'immagine statica da mostrare
     *
     * @return l'url dell'immagine statica da mostrare
     */
    fun getImageUrl(): String? {
        return spriteDefault
    }

}
```

9.1.2. Risorse

Le classi `MoveEntity`, `ItemEntity` e `AbilityEntity` rappresentano rispettivamente le mosse, gli oggetti e le abilità di un Pokémon. Queste entità sono correlate al Pokémon principale attraverso relazioni di molti-a-molti, come visto nella classe `PokemonEntity`.

Tutte e tre le classi sono annotate con `@Entity` per indicare che sono mappate a tabelle del database. Inoltre, utilizzano l'annotatazione `@TypeConverters` per gestire la conversione della lista di Translation in un tipo compatibile con il database.
Le classi contengono campi per i dati principali dell'entità, come l'ID, il nome, le statistiche (nel caso delle mosse) e i link agli sprite (nel caso degli oggetti). Inoltre, includono una lista di `Translation` per i nomi e le descrizioni in diverse lingue.
Per ottenere il nome o la descrizione localizzati di un'entità, le classi forniscono i metodi `getLocaleName()` e `getLocaleEffect()`. Questi metodi utilizzano il repository delle preferenze utente per ottenere la lingua corrente e cercano la traduzione corrispondente nella lista di Translation. Se non viene trovata una traduzione, viene restituito un messaggio di errore.

```
/**
 * Classe che rappresenta una abilità di un Pokemon
 */
@Entity(tableName = "ability")
@TypeConverters(TranslationListConverter::class)
data class AbilityEntity(
   @PrimaryKey val id: Int,
   val name: String,
   val isMainSeries: Boolean,
   val names: List<Translation> = emptyList(),
   val effects: List<Translation> = emptyList()
) {

   /**
    * Ottiene il nome dell'abilità in base alla lingua dell'utente
    *
    * @return Nome dell'abilità in base alla lingua dell'utente
    */
   fun getLocaleName(): String {
       val language = UserPreferencesRepository.getLanguagePreference()
       val name = names.find { it.language == language.code }
       return name?.text ?: "Translation not available"
   }

   /**
    * Ottiene la descrizione dell'abilità in base alla lingua dell'utente
    *
    * @return Descrizione dell'abilità in base alla lingua dell'utente
    */
   fun getLocaleEffect(): String {
       val language = UserPreferencesRepository.getLanguagePreference()
       val effect = effects.find { it.language == language.code }
       return effect?.text ?: "Translation not available"
   }
}
```

L'implementazione delle altre risorse è analoga

9.2. View

La View è responsabile della presentazione dei dati all'utente e della raccolta delle sue interazioni. Nella nostra applicazione, la View è rappresentata dagli elementi dell'interfaccia costruiti interamente con Jetpack Compose. Le View osservano i dati esposti dal ViewModel e si aggiornano automaticamente quando questi dati cambiano, grazie al meccanismo di LiveData o State in Jetpack Compose.

Gli elementi che compongono la parte View della schermata di lista dei Pokémon sono le seguenti:

9.2.1. PokemonListScreen

Serve per costruire la lista dei Pokémon mostrata nella pagina iniziale.
Si recupera la lista di Pokémon dal ViewModel (`pokemonListViewModel`) e la si visualizza utilizzando una `LazyVerticalGrid,` così da gestirla in modo efficiente viste le sue dimensioni. Utilizzando una griglia, in base alla dimensione dello schermo, vengono visualizzati più o meno Pokémon su una stessa riga. La riga di codice `columns = GridCells.Adaptive(PokemonListConstants.cellAdaptiveSize)` viene utilizzata all'interno della `LazyVerticalGrid` per definire il numero e la disposizione delle colonne nella griglia.
Questo codice imposta il numero di colonne nella `LazyVerticalGrid` in modo che si adatti automaticamente. Ciò significa che la griglia regolerà dinamicamente il numero di colonne in base alla dimensione dello schermo e alla larghezza minima specificata in `PokemonListConstants.cellAdaptiveSize`. Se lo schermo è abbastanza ampio, verranno visualizzate più colonne per riempire lo spazio in modo efficiente. Se lo schermo è più stretto, verranno utilizzate meno colonne per evitare uno scorrimento orizzontale eccessivo.
L'IconButton è stata utilizzata per realizzare il bottone Back To Top, che consente di scorrere la lista dei Pokémon e di tornare rapidamente all'inizio. Il pulsante è posizionato nell'angolo inferiore destro della schermata utilizzando `align(Alignment.BottomEnd)`.
Quando l'utente clicca sul pulsante, il codice avvia una `coroutine` per animare lo scorrimento della lista di Pokémon all'elemento iniziale (indice 0).
Il metodo `animateScrollToItem` della `LazyGridState` viene utilizzato per eseguire l'animazione dello scorrimento.

```
//variabile che memorizza lo stato della LazyVerticalGrid
val scope = rememberCoroutineScope()
val pokemonListState = rememberLazyGridState()


Box(
   //
) {
   LazyVerticalGrid(
       columns = GridCells.Adaptive(PokemonListConstants.cellAdaptiveSize),
       state = pokemonListState,
```

```
        content = {
            items(pokemonList.value) { pokemon ->
                PokemonCard(
                    pokemon = pokemon,
                    modifier
                        ),
                    onClick = {
                        onPokemonClicked(pokemon.id)
                    }
                )
            }
        }
    )

    IconButton(
        onClick = {
            scope.launch {
                pokemonListState.animateScrollToItem(index = 0)
            }
        }, modifier = Modifier
            .align(Alignment.BottomEnd)
            .padding(PokemonListConstants.buttonUpArrowPadding)
            .background(
                MaterialTheme.themedColorsPalette.mainYellow,
                RoundedCornerShape(PokemonListConstants.buttonUpArrowRadius)
            )
    ) {
        Icon(
            painter = painterResource(id = R.drawable.uparrow),
            contentDescription = "Up arrow",
            tint = MaterialTheme.themedColorsPalette.mainBlue
        )
    }
```

9.2.2. PokemonCard

Questa schermata definisce una composable chiamata `PokemonCard`, responsabile per la visualizzazione di un singolo Pokémon all'interno di una Card.
La Card mostra il nome del Pokémon, l'immagine animata e i tipi primari e secondari (se presenti). La card è cliccabile e richiama la funzione `onClick` passata come parametro. L'elemento `Canvas` crea lo sfondo di base bianco della card con colore di base bianco.
`AsyncImage` è un composable che esegue una ImageRequest in modo asincrono per un'immagine dato il suo URL e mostra il risultato; è stata utilizzata per recuperare le GIF animate a partire dall'URL fornito dalla PokéAPI.
Per ogni Card, il colore di sfondo è dettato dal tipo primario del Pokémon.
La funzione Composable personalizzata TypeRow è responsabile per la visualizzazione del tipo di un Pokémon all'interno di una etichetta. Si basa sull'enum `PokemonType` che fornisce informazioni sull'icona e sulla stringa da visualizzare.
Per il layout, impiega una riga (`Row`) per disporre orizzontalmente l'icona e il testo.

Il colore di sfondo da applicare viene impostato basandosi sul tipo di Pokémon.

```
/**
 * Composable per la card di un Pokémon.
 *
 * @param pokemon Il Pokémon da visualizzare nella card.
 * @param modifier I modificatori da applicare all'elemento.
 * @param onClick La funzione da eseguire quando l'utente clicca sulla card.
 */
@Composable
fun PokemonCard(
   pokemon: PokemonEntity,
   modifier: Modifier,
   onClick: () -> Unit
) {

   Column(
       modifier = modifier
           .clickable { onClick() },
          ) {

       // ...
       Text(pokemon.name, style =
MaterialTheme.themedTypography.pokemonCardTitle, modifier =
Modifier.padding(PokemonCardConstants.polemonCardPadding))
       Spacer(modifier = Modifier.height(PokemonCardConstants.spacerHeight))
       Box(
           modifier = Modifier
               .width(PokemonCardConstants.cardWidth)
               .height(PokemonCardConstants.cardHeight),
           contentAlignment = Alignment.Center
       ) {

           Canvas(
               modifier = Modifier
                   .width(PokemonCardConstants.cardWidth)
                   .height(PokemonCardConstants.cardHeight)
                   .background(
                       MaterialTheme.themedColorsPalette.pokemonImageBackground,

RoundedCornerShape(PokemonCardConstants.POKEMON_IMAGE_BORDER_RADIUS)
                   )
           ) {}

           AsyncImage(
               model = ImageRequest
                   .Builder(LocalContext.current)
                   .data(pokemon.getAnimatedImageUrl())
                   .decoderFactory(ImageDecoderDecoder.Factory())
                   .build(),
```

```
                contentDescription = "Pokemon GIF",
                modifier = Modifier
                    .height(PokemonCardConstants.pokemonImageWidth)
                    .width(PokemonCardConstants.pokemonImageHeight)
            )
        }
        if (pokemon.type1 !== null) {
            Spacer(modifier = Modifier.height(PokemonCardConstants.spacerHeight))
            TypeRow(type = pokemon.type1)
        }
        if (pokemon.type2 !== null) {
            Spacer(modifier = Modifier.height(PokemonCardConstants.spacerHeight))
            TypeRow(type = pokemon.type2)
        }
    }
}

/**
*  Composable per l'etichetta per mostrare i tipi del Pokemon.
*
*  @param type Il tipo da mostrare
*/
@Composable
fun TypeRow(type: PokemonType) {

    Row (modifier = ...)
       ){
        Image(
            painterResource(id = type.icon), "type_icon",
            modifier = ...)
        Text(stringResource(id = type.string), style =
MaterialTheme.themedTypography.typeRowTextStyle)
    }
}
```

9.2.3. SettingsDrawer

La sezione delle impostazione è costituita da due Composable correlate: `SettingsDrawer` e `DrawerContent`. Insieme, forniscono un drawer di impostazioni per l'applicazione.

In `SettingsDrawer`, per realizzare la sezione delle impostazioni, è stato utilizzato un `ModalDrawer`. In Jetpack Compose, un `ModalDrawer` è un componente dell'interfaccia utente che si sovrappone al contenuto principale dell'applicazione, bloccando l'interazione con esso fino a quando non viene chiuso.

Lo stato di apertura e chiusura del drawer viene gestito tramite una variabile di tipo `DrawerState`, ovvero un oggetto che gestisce lo stato di apertura e chiusura di un drawer.

Il contenuto del drawer viene fornito tramite la composable `DrawerContent`. Questo presenta le opzioni di impostazione per il tema e la lingua.

Per la selezione del tema chiaro/scuro viene utilizzato uno switch; invece, per la selezione della lingua, si utilizza un menù a tendina. `DrawerContent` utilizza lo stato `expandedState` per controllare l'apertura/chiusura del menu a tendina delle lingue.
Vengono utilizzate callback per notificare i cambiamenti di tema (`onThemeToggle`) e lingua (`onLanguageChange`) al resto dell'applicazione.
Inoltre, vengono visualizzate le informazioni sulla versione dell'applicazione e del database.

```
/**
 * Composable per il drawer di impostazioni.
 */
@Composable
fun SettingsDrawer(
   drawerState: DrawerState,
   isDarkTheme: Boolean,
   language: String,
   onThemeToggle: (Boolean) -> Unit,
   onLanguageChange: (selectedLanguage: Language) -> Unit,
   content: @Composable () -> Unit
) {

   // Contenuto del modal
   ModalDrawer (
       drawerState = drawerState,
       drawerContent = {
           DrawerContent(
               isDarkTheme = isDarkTheme,
               language = language,
               onThemeToggle = onThemeToggle,
               onLanguageChange = onLanguageChange
           )
       },
       // Contenuto sotto il modal
       content = content
   )

}

/**
 * Composable per il contenuto del drawer.
 */
@Composable
fun DrawerContent(
   isDarkTheme: Boolean,
   language: String,
   onThemeToggle: (Boolean) -> Unit,
   onLanguageChange: (selectedLanguage: Language) -> Unit
) {
   // Contiene lo stato del dropDown menu
   var expandedState by rememberSaveable { mutableStateOf(false) }
```

```
    ...

    // Menù a tendina per cambiare la lingua
        Row(...) {
            // Label per l'opzione della lingua
            Text(
                stringResource(R.string.language),
                style = MaterialTheme.themedTypography.drawerItemLabel,
                color = MaterialTheme.themedColorsPalette.mainTextColor
            )
            Box {
                // Bottone per aprire il menu a tendina
                Button(
                    onClick = { expandedState = !expandedState },
                    colors = ButtonDefaults.buttonColors(Color.Transparent),
                    modifier = ...) {
                    Text(
                        language,
                        style =
MaterialTheme.themedTypography.drawerOptionsLabel,
                        color =
MaterialTheme.themedColorsPalette.reverseTextColor,
                    )
                }
                // Menu a tendina per le lingue
                DropdownMenu(
                    expanded = expandedState,
                    onDismissRequest = { expandedState = false },
                    modifier = ...) {
                    Language.entries.map { it.text }.forEach { option ->
                        DropdownMenuItem(
                            {
                                Text(
                                    option,
                                    style =
MaterialTheme.themedTypography.drawerDropdownItemLabel
                                )
                            },
                            onClick = {
                                onLanguageChange(Language.fromText(option))
                                expandedState = false
                            }
                        )
                    }
                }
            }
        }
```

9.2.4. SplashScreen

Si tratta della schermata iniziale di caricamento, che viene mostrata all'apertura dell'applicazione, nel mentre che viene caricata la lista dei Pokémon.
Per il colore di sfondo, è stato creato un gradiente che riprende il colore blu dei Pokémon, per creare un effetto visivo gradevole. Per creare questo sfondo sfumato con due colori, impiega `Brush.radialGradient`.

```
/**
 * Composable per la splash screen dell'applicazione.
 */
@Composable
fun SplashScreen() {

    // Colore di sfondo
    val gradient = Brush.radialGradient(
        0.0f to colorResource(id = R.color.gradient_light),
        0.8f to colorResource(id = R.color.gradient_dark),
        radius = 1800.0f,
        tileMode = TileMode.Repeated
    )

    Column(...) {
        // Immagina statica di Pikachu
        Image(
            painter = painterResource(R.drawable.pikachu_static),
            contentDescription = "Pikachu",
            modifier = Modifier
                .size(200.dp)
        )
        Spacer(modifier = Modifier.height(5.dp))
        // Logo dell'applicazione
        Image(
            painterResource(id = R.drawable.logo2),
            contentDescription = "Logo PokeWorld",
            modifier = Modifier
                .size(180.dp)
                .fillMaxHeight())

    }
}
```

9.2.5. Topbar

La sezione della TopBar è composta da un elemento statico, ovvero la barra superiore che è fissa e sempre presente, e da un elemento dinamico, ovvero la barra di ricerca e la sezione dei filtri per tipo di Pokémon che vengono visualizzati solamente se `isSearchBarVisible` è `true`.
Per la sezione del Pokémon preferito, viene mostrata un'icona a stella vuota se non è presente alcun Pokémon preferito, altrimenti visualizza l'immagine del Pokémon preferito e un'icona a stella piena. Quando non ci sono Pokémon preferiti, se l'utente clicca sull'icona a

stella vuota, viene aperto un `AlertDialog` informativo, che visualizza un messaggio di avviso, informandolo che non ci sono Pokémon preferiti.
Per realizzare la barra di ricerca è stato utilizzato un `TextField`, che è un elemento che permette di raccogliere l'input dell'utente.

```
// Variabile per memorizzare la query corrente
val mQuery: String = query ?: ""


TextField(
   value = mQuery,
   onValueChange = { newValue ->
       filterPokemon(newValue, selectedType1, selectedType2)
   },
   placeholder = { Text(stringResource(R.string.search), style=
MaterialTheme.themedTypography.drawerOptionsLabel) } },
   modifier = ...,
   colors = ...),
   shape = RoundedCornerShape(topStart = TopbarConstants.searchBarRadius, topEnd
= TopbarConstants.searchBarRadius)
)
```

Inoltre, è stata creata una funzione Composable personalizzata per il menù per la selezione del tipo.

```
/**
 * Composable per il menu di selezione di tipo.
 *
 * @param type Il tipo selezionato.
 * @param expandedState Lo stato di estensione del menu.
 * @param onOptionSelected La funzione da eseguire quando viene selezionata una
opzione.
 * @param options Le opzioni del menu.
 * @param modifier I modificatori da applicare.
 */
@Composable
fun ChoiceTypeMenu(
   type: PokemonType?,
   expandedState: MutableState<Boolean>,
   onOptionSelected: (String) -> Unit,
   options: List<Pair<PokemonType?, String>>,
   modifier: Modifier
) {

   // Variabili per la gestione del menu
   var selectedOption = if(type != null) stringResource(type.string) else
stringResource(R.string.any)
   var selectedColor = type?.getBackgroundTextColor?.invoke() ?:
MaterialTheme.themedColorsPalette.mainBlue
   var buttonPosition by remember { mutableStateOf(IntOffset(0, 0)) }
```

```
    var buttonSize by remember { mutableStateOf(IntSize.Zero) }

    val configuration = LocalConfiguration.current
    val isLandscape = configuration.orientation ==
Configuration.ORIENTATION_LANDSCAPE

    // Il testo visualizzato viene gestito dal componente genitore
    Box (modifier = modifier){
        Button(
            colors = ButtonDefaults.buttonColors(Color.Transparent),
            onClick = { expandedState.value = !expandedState.value },
            modifier = Modifier
                .background(
                    color = selectedColor,
                    RoundedCornerShape(10.dp)
                )
                .fillMaxWidth()
                .onGloballyPositioned { coordinates ->
                    val position = coordinates.positionInWindow()
                    buttonPosition = IntOffset(0, position.y.toInt())
                    buttonSize = coordinates.size
                }
        ) {
            Text(
                selectedOption,
                style = MaterialTheme.themedTypography.topbarFilterTypeLabel,
                color =
MaterialTheme.themedColorsPalette.topbarFilterTypeTextColor,
            )
        }

        if (expandedState.value) {
            val offsetY = if (isLandscape) {
                buttonPosition.y + buttonSize.height
            } else {
                buttonSize.height
            }
            Popup(
                alignment = Alignment.TopStart,
                offset = IntOffset(0, offsetY),
                properties = PopupProperties(focusable = true)
            ) {
                Surface(
                    color = MaterialTheme.colorScheme.background,
                    shape =
RoundedCornerShape(TopbarConstants.choiceMenuCornerShape),
                    modifier = Modifier
                        .width(
                            maxOf(
                                with(LocalDensity.current) {
buttonSize.width.toDp() },
```

```
                                TopbarConstants.choiceMenuMaxWidth
                            )
                        )
                        .height(TopbarConstants.choiceMenuHeight)
                ) {
                    Column(
                        modifier = Modifier
                            .fillMaxWidth()
                            .verticalScroll(rememberScrollState())
                    ) {
                        options.forEach { (t, text) ->

                            val backgroundColor = t?.getBackgroundColor?.invoke()
?: Color.White
                            val backgroundTextColor =
t?.getBackgroundTextColor?.invoke() ?: Color.White

                            DropdownMenuItem(
                                {
                                    Row(
                                        modifier = Modifier
                                            .fillMaxWidth()

.height(TopbarConstants.choiceMenuButtonHeight)
                                            .background(
                                                backgroundColor,

RoundedCornerShape(TopbarConstants.choiceMenuButtonRadius)
                                            ),
                                        verticalAlignment =
Alignment.CenterVertically,
                                        horizontalArrangement =
Arrangement.Center
                                    ) {
                                        if (t != null) {
                                            Image(
                                                painter = painterResource(id =
t.icon),

                                                contentDescription = "icon",
                                                modifier = Modifier

.height(TopbarConstants.choiceMenuIconOptionsHeight)
                                            )
                                            Spacer(modifier =
Modifier.width(TopbarConstants.spacerWidth))
                                        }
                                        Text(
                                            text = text,
                                            style =
MaterialTheme.themedTypography.topbarFilterTypeOptionsLabel,
                                            color =
```

```
MaterialTheme.themedColorsPalette.topbarFilterTypeTextColor
                                        )
                                    }
                                },
                                onClick = {
                                    selectedOption = text
                                    selectedColor = backgroundTextColor
                                    onOptionSelected(t?.type ?: "any")
                                    expandedState.value = false
                                })
                        }
                    }
                }
            }
        }
    }
}
```

Inoltre, sono stati creati anche i due bottoni Reset e Random.

```
//Pulsante per il Reset

Button(onClick = {
    filterPokemon(null, null, null)
},
    colors = ButtonDefaults.buttonColors(Color.Transparent),
    modifier = Modifier
        .background(
            MaterialTheme.themedColorsPalette.mainYellow,
            RoundedCornerShape(TopbarConstants.BUTTON_RADIUS)
        )
        .width(TopbarConstants.buttonWidth)
        .height(TopbarConstants.buttonHeight)
) {
    Icon(
        painter = painterResource(id = R.drawable.reset),
        contentDescription = "reset",
        tint = MaterialTheme.themedColorsPalette.mainBlue,
        modifier = Modifier.size(TopbarConstants.buttonIconSize)
    )
}

//Pulsante per la scelta casuale
Button(onClick = {
    val (type1, type2) = randomFilters()
    filterPokemon(query, type1, type2)
},
    colors = ButtonDefaults.buttonColors(Color.Transparent),
    modifier = Modifier
        .background(
```

```
            MaterialTheme.themedColorsPalette.mainYellow,
            RoundedCornerShape(TopbarConstants.BUTTON_RADIUS)
        )
        .width(TopbarConstants.buttonWidth)
        .height(TopbarConstants.buttonHeight)
){
    Icon(
        painter = painterResource(id = R.drawable.random),
        contentDescription = "random",
        tint = MaterialTheme.themedColorsPalette.mainBlue,
        modifier = Modifier.size(TopbarConstants.buttonIconSize)
    )
}
```

9.2.6. PokemonDetailsScreen

Questa Composable è stata progettata per visualizzare i dettagli di un Pokémon specifico. La funzione riceve come parametri il `ViewModel` per la gestione dei dati del Pokémon e l'ID del Pokémon da visualizzare. Una volta ottenuti i dati, questi verranno visualizzati all'interno di una `DetailsCard` composable. Se i dati non sono ancora disponibili, viene visualizzata un pagina di caricamento (`Loader`). La funzione implementa un meccanismo di ricaricamento dei dati nel caso in cui l'ID del Pokémon cambi o se i dati non siano ancora stati caricati completamente.

```
/**
 * Composable per la creazione della pagina di dettaglio del Pokemon.
 *
 * @param pokemonDetailViewModel Il ViewModel per la il dettaglio dei Pokemon.
 * @param pokemonId L'identificativo del pokemon di cui mostrare la pagina
dettaglio.
 */
@Composable
fun PokemonDetailsScreen (
    pokemonDetailViewModel: PokemonDetailViewModel,
    pokemonId: Int
)
{

    // Ottiene lo stato del pokemon dal ViewModel
    val pokemon: PokemonEntity? =
pokemonDetailViewModel.pokemon.collectAsStateWithLifecycle().value
    val isDetailsLoaded =
pokemonDetailViewModel.detailsLoaded.collectAsStateWithLifecycle().value

    // Se il pokemon non è stato ancora caricato o se è cambiato, carica il
pokemon
    if(!isDetailsLoaded || pokemon?.id != pokemonId)
pokemonDetailViewModel.loadPokemon(pokemonId)

    if(isDetailsLoaded)
```

```
        DetailsCard(pokemonDetailViewModel, pokemon!!)
    else
        Loader()

}
```

9.2.7. DetailsCard

Questa pagina è responsabile per la visualizzazione dettagliata di un Pokémon specifico. Essa riceve come parametri un'istanza di `PokemonDetailViewModel` e un oggetto `PokemonEntity`.

Accesso ai dati e gestione dello stato:

- Il contesto locale (`LocalContext`) viene recuperato per accedere alle risorse dell'applicazione.
- Il metodo `getFavoritePokemonId` del `pokemonDetailViewModel` viene utilizzato per ottenere l'ID del Pokémon attualmente favorito.
- Una composable `remember` memorizza lo stato dei preferiti basato sul confronto iniziale e lo aggiorna dinamicamente.
- Una variabile di stato `showConfirmationDialog` gestisce la visibilità di un dialogo di conferma quando si sostituiscono i preferiti.

Colore di sfondo e testo:

- Il colore di sfondo e il colore del testo vengono determinati dinamicamente in base al tipo primario del Pokémon utilizzando metodi getter dedicati (`getBackgroundColor` e `getBackgroundTextColor`).

Superficie e colonna scorrevole:

- Una `Surface` definisce lo sfondo con il colore recuperato.
- Una `LazyColumn` consente di scorrere il contenuto all'interno della card, ottimizzando le prestazioni per grandi quantità di dati.

Struttura del contenuto:

- La `LazyColumn` contiene più elementi che rappresentano diverse sezioni dei dettagli del Pokémon:
    - **Intestazione:** Visualizza nome del Pokémon, i tipi (primario e secondario) e la GIF corrispondente, con un pulsante per riprodurre l'audio e un pulsante per i preferiti.
    - **Caratteristiche:** Mostra l'altezza e il peso del Pokémon
    - **Statistiche, abilità, oggetti e mosse:** Ogni sezione viene visualizzata con una composable dedicata (`StatsSection`, `AbilitiesSection`, `ItemSection`, `MoveItem`).

Pulsante dei preferiti:

- L'icona del pulsante dei preferiti cambia in base allo stato `isFavorite` (stella vuota o stella gialla).
- Cliccando sul pulsante, si eseguono azioni diverse a seconda dello stato corrente del preferito:
    - Se il Pokémon è già impostato come preferito, viene rimosso.
    - Se un altro Pokémon è preferito e l'utente tenta di impostare un nuovo Pokémon come preferito, appare un dialogo di conferma.
    - Se nessun Pokémon è favorito e l'utente sta aggiungendo l'attuale Pokémon come preferito, viene impostato come preferito.

Dialogo di conferma:

- Una composable `ConfirmFavoriteChangeDialog` viene visualizzata in modo condizionale basato sullo stato `showConfirmationDialog`.
- Il dialogo fornisce opzioni per confermare o annullare la sostituzione dell'attuale Pokémon preferito.

```
/**
 * Composable per la finestra di dialogo per cambiare il Pokemon preferito.
 *
 * @param onConfirm Callback da eseguire a seguito di conferma da parte
dell'utente.
 * @param onCancel Callback da eseguire a seguito di annullamento.
 */
@Composable
fun ConfirmFavoriteChangeDialog(
   onConfirm: () -> Unit,
   onCancel: () -> Unit,
) {

   val titleText = stringResource(R.string.change_favorite)
   val messageText = stringResource(R.string.change_favorite_disclaimer)
   val confirmButtonText = stringResource(R.string.yes)
   val cancelButtonText = stringResource(R.string.no)

   AlertDialog(
       onDismissRequest = onCancel,
       title = {
           Text(text = titleText,
               style = MaterialTheme.themedTypography.alertDialogTitleStyle)
       },
       text = {
           Text(text = messageText,
               style = MaterialTheme.themedTypography.alertDialogTextStyle)
       },
       confirmButton = {
           Button(
```

```
                onClick = onConfirm,
            ) {
                Text(
                    text = confirmButtonText,
                    style = MaterialTheme.themedTypography.alertDialogTextStyle
                )
            }
        },
        dismissButton = {
            Button(
                onClick = onCancel,
            ) {
                Text(text = cancelButtonText, style =
MaterialTheme.themedTypography.alertDialogTextStyle)
            }
        }
    )
}
```

Pulsante audio:

La funzione chiamata `playAudio` è responsabile della riproduzione del verso di un Pokémon. Essa riceve come parametri:

- `context`: Il contesto dell'applicazione utilizzato per accedere alle risorse e visualizzare messaggi toast.
- `pokemon`: Un oggetto `PokemonEntity` contenente informazioni sul Pokémon, incluso l'URL del suo verso (`pokemon.criesLatest`).

La funzione verifica innanzitutto se è disponibile una connessione di rete utilizzando la funzione `isNetworkAvailable`. Se non viene rilevata alcuna connessione Internet, viene visualizzato un messaggio toast per informare l'utente. Altrimenti, viene creata un'istanza di `MediaPlayer` per la riproduzione audio.

Una coroutine lanciata sul thread `Dispatchers.IO` gestisce il caricamento audio asincrono:

- Viene impostato un timeout di 5 secondi utilizzando `withTimeout`.
- All'interno della finestra di timeout, il `MediaPlayer` imposta l'URL della fonte dati utilizzando l'URL del verso del Pokémon.
- Il metodo `mediaPlayer.prepare()` viene chiamato per preparare l'audio per la riproduzione.
- Utilizzando `withContext(Dispatchers.Main)`, il media player viene avviato sul thread principale per garantire che gli aggiornamenti dell'interfaccia utente (come la visualizzazione di un indicatore di caricamento) siano reattivi.

Un blocco `try-catch` gestisce le potenziali eccezioni. Viene intercettata una `TimeoutCancellationException` se il caricamento audio richiede più di 5 secondi. In questo caso, il media player viene rilasciato e viene visualizzato un messaggio toast sul thread principale per informare l'utente di una connessione Internet instabile.

Un blocco `finally` garantisce che le risorse del media player vengano rilasciate correttamente indipendentemente dal successo della riproduzione o dalle eccezioni. Un listener di completamento viene associato al media player sul thread principale. Quando la riproduzione termina, il listener rilascia le risorse del media player.

```
/**
 * Funzione per riprodurre il verso del Pokemon.
 *
 * @param context Il contesto in cui viene eseguita la funzione.
 * @param pokemon Il Pokemon di cui riprodurre il suono
 */
fun playAudio(context: Context, pokemon: PokemonEntity) {

   if (!isNetworkAvailable(context)) {
       Toast.makeText(context, R.string.no_internet, Toast.LENGTH_SHORT).show()
       return
   }

   val mediaPlayer = MediaPlayer()

   CoroutineScope(Dispatchers.IO).launch {
       try {
           withTimeout(5000L) {
               mediaPlayer.setDataSource(pokemon.criesLatest)
               mediaPlayer.prepare()
               withContext(Dispatchers.Main) {
                   mediaPlayer.start()
               }
           }
       } catch (e: TimeoutCancellationException) {
           withContext(Dispatchers.Main) {
               mediaPlayer.release()
               Toast.makeText(context, R.string.unstable_internet,
Toast.LENGTH_SHORT).show()
           }
       } finally {
           withContext(Dispatchers.Main) {
               mediaPlayer.setOnCompletionListener {
                   mediaPlayer.release()
               }
           }
       }
   }
}
```

9.2.8. Loader

Questa schermata è stata costruita in modo del tutto analogo a quella di SplashScreen e rappresenta la schermata di caricamento della pagina di Dettaglio di un Pokémon.

9.2.9. StatsSection

Questa Composable è responsabile per la visualizzazione delle statistiche dettagliate di un Pokémon.
La composable utilizza una `Column` per disporre verticalmente le statistiche, con ogni riga che contiene due etichette statistiche.
Ogni statistica (HP, Attacco, Difesa, Velocità, Difesa Speciale, Attacco Speciale) viene visualizzata in una riga separata con un colore di sfondo apposito per ogni tipo di statistica.
Viene mantenuta una spaziatura coerente tra le righe e all'interno di ciascuna riga utilizzando componenti `Spacer`.

```
/**
 * Composable per la sezione relativa alle statistiche del Pokemon.
 *
 * @param pokemon Il Pokemon di cui mostrare le statistiche.
 */
@Composable
fun StatsSection(pokemon: PokemonEntity){

   Column(
       modifier = Modifier
           .background(WhiteDetails)
           .fillMaxWidth(),
       horizontalAlignment = Alignment.CenterHorizontally
   ){
       Text(
           text = stringResource(R.string.stats),
           style = MaterialTheme.themedTypography.sectionsTitleTextStyle,
           modifier = Modifier.padding(StatsSectionConstants.sectionPadding)
       )
       Column {
           Row {
               Text(
                   text = stringResource(R.string.hp) + ": ${pokemon.hp}",
                   modifier = Modifier
                       .background(
                           colorResource(id = R.color.hp),

RoundedCornerShape(StatsSectionConstants.statLabelRoundedCornerShape)
                       )
                       .width(StatsSectionConstants.statLabelWidth)
                       .padding(StatsSectionConstants.statLabelPadding),
                   style = MaterialTheme.themedTypography.statisticsTextStyle
               )
               Spacer(modifier =
Modifier.width(StatsSectionConstants.rowSpacerWidth))
```

```
                Text(
                    text = stringResource(R.string.attack) + ":
${pokemon.attack}",
                    modifier = Modifier
                        .background(
                            colorResource(id = R.color.attack),

RoundedCornerShape(StatsSectionConstants.statLabelRoundedCornerShape)
                        )
                        .width(StatsSectionConstants.statLabelWidth)
                        .padding(StatsSectionConstants.statLabelPadding),
                    style = MaterialTheme.themedTypography.statisticsTextStyle
                )
            }
            Spacer(modifier =
Modifier.height(StatsSectionConstants.columnSpacerHeight))
            Row {
                Text(
                    text = stringResource(R.string.defense) + ":
${pokemon.defense}",
                    modifier = Modifier
                        .background(
                            colorResource(id = R.color.defense),

RoundedCornerShape(StatsSectionConstants.statLabelRoundedCornerShape)
                        )
                        .width(StatsSectionConstants.statLabelWidth)
                        .padding(StatsSectionConstants.statLabelPadding),
                    style = MaterialTheme.themedTypography.statisticsTextStyle
                )
                Spacer(modifier =
Modifier.width(StatsSectionConstants.rowSpacerWidth))
                Text(
                    text = stringResource(R.string.speed) + ": ${pokemon.speed}",
                    modifier = Modifier
                        .background(
                            colorResource(id = R.color.speed),

RoundedCornerShape(StatsSectionConstants.statLabelRoundedCornerShape)
                        )
                        .width(StatsSectionConstants.statLabelWidth)
                        .padding(StatsSectionConstants.statLabelPadding),
                    style = MaterialTheme.themedTypography.statisticsTextStyle
                )
            }
            Spacer(modifier =
Modifier.height(StatsSectionConstants.columnSpacerHeight))
            Row {
                Text(
                    text = stringResource(R.string.special_defense) + ":
${pokemon.specialDefense}",modifier = Modifier
```

```
                        .background(
                            colorResource(id = R.color.special_defense),

RoundedCornerShape(StatsSectionConstants.statLabelRoundedCornerShape)
                        )
                        .width(StatsSectionConstants.statLabelWidth)
                        .padding(StatsSectionConstants.statLabelPadding),
                    style = MaterialTheme.themedTypography.statisticsTextStyle
                )
                Spacer(modifier =
Modifier.width(StatsSectionConstants.rowSpacerWidth))
                Text(
                    text = stringResource(R.string.special_attack) + ":
${pokemon.specialAttack}",
                    modifier = Modifier
                        .background(
                            colorResource(id = R.color.special_attack),

RoundedCornerShape(StatsSectionConstants.statLabelRoundedCornerShape)
                        )
                        .width(StatsSectionConstants.statLabelWidth)
                        .padding(StatsSectionConstants.statLabelPadding),
                    style = MaterialTheme.themedTypography.statisticsTextStyle
                )
            }
            Spacer(modifier =
Modifier.height(StatsSectionConstants.columnSpacerHeight))
        }
    }
}
```

9.2.10. AbilitiesSection

Questa sezione è responsabile per la visualizzazione delle abilità di un Pokémon.
La sezione utilizza una `Column` per disporre verticalmente le abilità.
Ogni abilità è rappresentata da una composable `AbilityItem`, che include il nome dell'abilità e un toggle per espandere/chiudere la descrizione dell'abilità. La variabile di stato `isExpanded` controlla se i dettagli dell'abilità sono mostrati o nascosti.
La sezione e i singoli elementi abilità utilizzano colori di sfondo associati al tipo del Pokémon per una coerenza visiva.

```
/**
 * Composable per la sezione relativa alle abilitàdel Pokemon.
 *
 * @param pokemon Il Pokemon di cui mostrare le abilità.
 */
@Composable
fun AbilitiesSection(pokemon: PokemonEntity) {
    Column(
```

```
        modifier = Modifier
            .fillMaxWidth()
            .padding(AbilitySectionConstants.sectionPadding),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = stringResource(R.string.abilities),
            style = MaterialTheme.themedTypography.sectionsTitleTextStyle,
            modifier = Modifier.padding(AbilitySectionConstants.titlePadding)
        )
        pokemon.abilities.forEach { ability ->
            AbilityItem(
                abilityName = ability.getLocaleName(),
                abilityDescription = ability.getLocaleEffect(),
                typeColor = pokemon.type1!!.getBackgroundTextColor(),
                backgroundColor = pokemon.type1.getBackgroundColor()
            )
        }
    }
}

/**
 * Composable per mostrare una abilità.
 *
 * @param abilityName Il nome dell'abilità.
 * @param abilityDescription La descrizione dell'abilità.
 * @param typeColor Il primo colore associato al primo tipo del Pokemon di cui
si sta mostrando l'abilità.
 * @param backgroundColor Il secondo colore associato al primo tipo del Pokemon
di cui si sta mostrando l'abilità.
 */
@Composable
fun AbilityItem(
    abilityName: String,
    abilityDescription: String,
    typeColor: Color,
    backgroundColor: Color
) {
    var isExpanded by remember { mutableStateOf(false) }

    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(AbilityItemConstants.itemPadding)
            .background(typeColor,
RoundedCornerShape(AbilityItemConstants.roundedCornerShape))
            .clickable { isExpanded = !isExpanded }
            .padding(AbilityItemConstants.contentPadding)
    ) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
```

```
            horizontalArrangement = Arrangement.SpaceBetween,
            modifier = Modifier.fillMaxWidth()
        ) {
            Text(
                text = abilityName,
                style = MaterialTheme.themedTypography.itemTextStyle,
                modifier = Modifier.weight(1f)
            )
            Arrow(isExpanded = isExpanded)
        }

        if(isExpanded) {
            Text(
                text = abilityDescription,
                style = MaterialTheme.themedTypography.itemTextStyle,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(top =
AbilityItemConstants.expandedDescriptionPaddingTop)
                    .background(backgroundColor,
RoundedCornerShape(AbilityItemConstants.roundedCornerShape))
                    .padding(AbilityItemConstants.contentPadding)
            )
        }
    }
}
```

9.2.11. ItemsSection

Questa sezione è responsabile per la visualizzazione degli oggetti posseduti da un Pokémon.

La sezione utilizza una `Column` per disporre verticalmente gli oggetti.

Ogni oggetto è rappresentato da una composable `ItemCard`, che include il nome dell'oggetto e un'immagine (se disponibile). Inoltre, cliccando su una ItemCard, si espandono o si collassano i suoi dettagli (costo ed effetto).

```
/**
 * Composable per la sezione relativa agli oggetti del Pokemon.
 *
 * @param pokemon Il Pokemon di cui mostrare gli oggetti.
 */
@Composable
fun ItemSection(pokemon: PokemonEntity) {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(ItemSectionConstants.sectionPadding),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
```

```
        Text(
            text = stringResource(R.string.items),
            style = MaterialTheme.themedTypography.sectionsTitleTextStyle,
            modifier = Modifier.padding(ItemSectionConstants.sectionPadding)
        )
        Row(
            horizontalArrangement =
Arrangement.spacedBy(ItemSectionConstants.cardSpacing),
            modifier = Modifier.fillMaxWidth()
        ) {
            pokemon.items.forEach { item ->
                ItemCard(
                    itemName = item.getLocaleName(),
                    itemCost = item.cost,
                    itemEffect = item.getLocaleEffect(),
                    itemSprite = item.sprite,
                    typeColor = pokemon.type1!!.getBackgroundTextColor()
                )
            }
        }
    }
}

/**
 * Composable per mostrare un oggetto.
 *
 * @param itemName Il nome dell'oggetto.
 * @param itemCost Il costo dell'oggetto.
 * @param itemEffect L'effetto che ha l'oggetto.
 * @param itemSprite L'url dell'immagine dell'oggetto.
 * @param typeColor Il primo colore associato al primo tipo del Pokemon di cui
si sta mostrando l'oggetto.
 */
@Composable
fun ItemCard(
    itemName: String,
    itemCost: Int,
    itemEffect: String,
    itemSprite: String?,
    typeColor: Color) {
    var isExpanded by remember { mutableStateOf(false) }

    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(ItemCardConstants.itemPadding)
            .border(
                width = ItemCardConstants.borderWidth,
                color = typeColor,
                shape = RoundedCornerShape(ItemCardConstants.roundedCornerShape)
            )
```

```
            .background(WhiteDetails,
RoundedCornerShape(ItemCardConstants.roundedCornerShape))
            .clickable { isExpanded = !isExpanded }
            .padding(ItemCardConstants.itemPadding)
    ) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween,
            modifier = Modifier.fillMaxWidth()
        ) {
            itemSprite?.let {
                AsyncImage(
                    model = it,
                    contentDescription = null,
                    modifier = Modifier.size(ItemCardConstants.imageSize)
                )
            }
            Text(
                text = itemName,
                style = MaterialTheme.themedTypography.itemTextStyle,
                modifier = Modifier.padding(top =
ItemCardConstants.itemNamePaddingTop)
            )
            Arrow(isExpanded)
        }

        if (isExpanded) {
            Spacer(modifier =
Modifier.height(ItemCardConstants.expandedSectionSpacerHeight))
            Text(
                text = buildAnnotatedString {
                    withStyle(style = SpanStyle(textDecoration =
TextDecoration.Underline)) {
                        append(stringResource(R.string.cost) + ":")
                    }
                    append(" $itemCost")
                },
                style = MaterialTheme.themedTypography.itemTextStyle,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(top = ItemCardConstants.textPaddingTop)
            )

            Spacer(modifier = Modifier.height(ItemCardConstants.textPaddingTop))
            Text(
                text = buildAnnotatedString {
                    withStyle(style = SpanStyle(textDecoration =
TextDecoration.Underline)) {
                        append(stringResource(R.string.effect) + ":")
                    }
                    append(" $itemEffect")
```

```
                },
                style = MaterialTheme.themedTypography.itemTextStyle,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(top = ItemCardConstants.textPaddingTop))
        }
    }
}
```

9.2.12. MovesSection

Questa sezione è responsabile per la visualizzazione delle informazioni relative a una singola mossa di un Pokémon.
Le informazioni sulla mossa sono strutturate all'interno di una `Column` per una disposizione verticale.
Il nome della mossa viene visualizzato in modo prominente, insieme alla sua priorità indicata in un elemento di testo separato e stilizzato.
Inoltre, vengono presentati ulteriori dettagli come precisione, probabilità di effetto e potenza.
La descrizione della mossa viene visualizzata solo quando l'utente espande l'elemento facendo clic su di esso.

```
/**
 * Composable per mostrare una mossa.
 *
 * @param moveName Il nome della mossa.
 * @param moveDescription La descrizione della mossa.
 * @param accuracy La precisione della mossa.
 * @param effectChance La probabilità di effetto della mossa.
 * @param power La potenza della mossa.
 * @param priority La priorità della mossa.
 * @param type Il tipo del Pokemon di cui stiamo mostrando la mossa.
 */
@Composable
fun MoveItem(
    moveName: String,
    moveDescription: String,
    accuracy: Int?,
    effectChance: Int?,
    power: Int?,
    priority: Int?,
    type: PokemonType
) {
    var isExpanded by remember { mutableStateOf(false) }
    val hasDescription = moveDescription.isNotEmpty()
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(MoveItemConstants.itemOuterPadding)
            .background(
```

```
                type.getBackgroundTextColor().copy(alpha =
MoveItemConstants.ITEM_BACKGROUND_ALPHA),
                RoundedCornerShape(MoveItemConstants.itemRoundedCornerShape)
            )
            .padding(MoveItemConstants.itemPadding)
            .clickable { isExpanded = !isExpanded }
    ) {

        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier.fillMaxWidth()
        ) {
            Text(
                text = moveName,
                style = MaterialTheme.themedTypography.itemTextStyle,
                modifier = Modifier.weight(1f),
                textAlign = TextAlign.Start
            )
            Text(
                text = stringResource(R.string.priority) + ": ${priority ?:
"-"}",
                fontSize = 8.sp, // Esempio di stile non incluso nelle costanti
                modifier = Modifier

.background(Color.White.copy(MoveItemConstants.PRIORITY_BACKGROUND_ALPHA),

RoundedCornerShape(MoveItemConstants.priorityRoundedCornerShape)
                    )
                    .padding(
                        horizontal = MoveItemConstants.priorityPadding,
                        vertical = MoveItemConstants.priorityVerticalPadding
                    ),
                textAlign = TextAlign.End
            )
        }
        Spacer(modifier = Modifier.height(MoveItemConstants.spacerHeight))
        Divider(
            color = Color.Black.copy(alpha = MoveItemConstants.DIVIDER_ALPHA),
            thickness = MoveItemConstants.dividerThickness,
            modifier = Modifier.fillMaxWidth()
        )
        Spacer(modifier = Modifier.height(MoveItemConstants.spacerHeight))
        if (hasDescription) {
            Row(
                modifier = Modifier
                    .fillMaxWidth(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.CenterVertically
            ) {
                Text(
                    text = stringResource(R.string.description),
```

```
                    style = MaterialTheme.themedTypography.itemTextStyle,
                    modifier = Modifier.padding(horizontal =
MoveItemConstants.descriptionPadding)
                )
                Arrow(isExpanded = isExpanded)
            }

        }
        if (hasDescription && isExpanded) {
            Text(
                text = moveDescription,
                fontSize = 9.sp, // Esempio di stile non incluso nelle costanti
                modifier = Modifier
                    .fillMaxWidth()
                    .background(

Color.White.copy(MoveItemConstants.EXPANDED_DESCRIPTION_BACKGROUND_ALPHA),

RoundedCornerShape(MoveItemConstants.expandedDescriptionRoundedCornerShape)
                    )
                    .padding(MoveItemConstants.expandedDescriptionPadding)

            )
        }
        Spacer(modifier = Modifier.height(MoveItemConstants.spacerHeight))
        Row(
            horizontalArrangement = Arrangement.SpaceBetween,
            modifier = Modifier.fillMaxWidth()
        ) {
            ValueMoves(stringResource(R.string.accuracy), accuracy)
            ValueMoves(stringResource(R.string.effect_chance), effectChance)
            ValueMoves(stringResource(R.string.power), power)
        }

    }
}
```

9.3. ViewModel

Il `ViewModel` funge da intermediario tra il Model e la View. Contiene la logica di presentazione, prepara i dati per la View e risponde agli eventi dell'utente. Nel nostro progetto, abbiamo implementato due `ViewModel` principali, entrambi istanziati attraverso una classe figlia di `ViewModelProvider.Factory`.
Dunque, il `ViewModel` isola la logica di business dalla vista, promuovendo una maggiore modularità e facilitando la testabilità del codice.

9.3.1. ViewModelFactory

La classe `ViewModelFactory` è stata implementata per gestire la creazione delle istanze dei `ViewModel` all'interno dell'applicazione. Adottando il pattern `Factory`, questa classe garantisce l'iniezione delle dipendenze necessarie al momento dell'instanziazione di ciascun `ViewModel`.
In particolare, la `ViewModelFactory` fornisce un meccanismo centralizzato per associare i tipi di `ViewModel` (ad esempio, `PokemonListViewModel` e `PokemonDetailViewModel`) alle rispettive dipendenze, come il `PokemonRepository`, assicurando così un corretto isolamento dei componenti e una maggiore manutenibilità del codice.
In linea con il pattern `Factory`, il metodo `create` della `ViewModelFactory` incapsula la logica di creazione delle istanze dei `ViewModel`, nascondendo ai client i dettagli della loro costruzione.
Questo approccio promuove un maggiore accoppiamento e facilita l'introduzione di nuovi tipi di `ViewModel` in futuro.
L'instanziazione dei `ViewModel` non viene effettuata direttamente ma viene delegata alla funzione di `Compose viewModel` che ne gestisce anche il ciclo di vita.

```
/**
 * Factory per la creazione di ViewModel.
 *
 * @param pokemonRepository Il repository per le operazioni di database.
 */
@Suppress("UNCHECKED_CAST")
class ViewModelFactory(
   private val pokemonRepository: PokemonRepository,
): ViewModelProvider.Factory {

   /**
    * Crea una nuova istanza di ViewModel.
    *
    * @param modelClass La classe del ViewModel da creare.
    */
   override fun <T : ViewModel> create(modelClass: Class<T>): T {

       if
(modelClass.isAssignableFrom(PokemonListViewModel::class.java))
           return PokemonListViewModel(pokemonRepository) as T
       else if
(modelClass.isAssignableFrom(PokemonDetailViewModel::class.java))
           return PokemonDetailViewModel(pokemonRepository) as T

       throw IllegalArgumentException("Unknown ViewModel class")
   }
```

```
}
```

9.3.2. PokemonListViewModel

La funzione primaria di questo `ViewModel` consiste nel gestire il ciclo di vita dei dati relativi alla lista dei Pokémon nella relativa `View`. Al momento dell'inizializzazione, il `ViewModel` procede al recupero dell'intera lista di Pokémon dal repository sottostante, popolando così il proprio stato interno. Successivamente, il `ViewModel` espone all'interfaccia utente una vista filtrata e aggiornata in tempo reale della lista completa, consentendo all'utente di eseguire ricerche e raffinare i risultati in base a criteri specifici come nome e tipo.

Il `ViewModel`, inoltre, mantiene una referenza al Pokémon preferito e la espone all'interfaccia utente, consentendo alla vista di visualizzare e aggiornare dinamicamente questa informazione.
Il meccanismo fondamentale su cui questo `ViewModel` si base è il sistema di notifica `StateFlow`. Grazie a questo strumento, ogni volta che la lista dei Pokémon filtrati subisce modifiche, l'interfaccia utente viene automaticamente aggiornata, garantendo una sincronizzazione costante tra i dati sottostanti e la rappresentazione visuale.

```
/**
 * Classe ViewModel per la schermata PokemonListScreen.
 *
 * @param pokemonRepository Repository per le operazioni di database.
 */
class PokemonListViewModel(
   private val pokemonRepository: PokemonRepository,
): ViewModel() {

   // Lista dei pokemon da visualizzare sulla schermata principale
   private var _pokemonList: List<PokemonEntity> = emptyList()
   private val _filteredPokemonList: MutableStateFlow<List<PokemonEntity>> =
MutableStateFlow(emptyList())

   // Pokemon preferito selezionato dall'utente
   val favoritePokemon: StateFlow<PokemonEntity?> =
FavoritePokemonSharedRepository.sharedFavoritePokemon
   val pokemonList: StateFlow<List<PokemonEntity>>
       get() = _filteredPokemonList.asStateFlow()

   // Inizializzazione della lista dei pokemon
   init {
       viewModelScope.launch(Dispatchers.IO) {
           // Carica la lista dei pokemon dal db
           _pokemonList = pokemonRepository.retrievePokemonList()
           _filteredPokemonList.value = _pokemonList
       }
   }
```

```
    /**
     * Filtra la lista dei pokemon in base al nome e ai due tipi selezionati.
     * Se viene passato come parametro null la funzione non considera quel
filtro.
     *
     * @param name Nome del pokemon da cercare.
     * @param type1 Tipo 1 del pokemon da cercare.
     * @param type2 Tipo 2 del pokemon da cercare.
     */
    fun filterPokemon(name: String?, type1: PokemonType?, type2: PokemonType?) {
        // Contiene la lista da filtrare
        var filteredList = _pokemonList

        // Filtra per nome
        if (name?.isNotBlank() == true) filteredList = filteredList.filter {
it.name.contains(name, ignoreCase = true) }
        // Filtra per il tipo 1
        if (type1 != null) filteredList = filteredList.filter { it.type1 == type1
}
        // Filtra per il tipo 2
        if (type2 != null) filteredList = filteredList.filter { it.type2 == type2
}

        // Risultato del filtro
        _filteredPokemonList.value = filteredList
    }

    /**
     * Genera casualmente due tipi di pokemon diversi.
     * La funzione controlla che i filtri non producano in uscita una lista
vuota.
     *
     * @return Una coppia di tipi di pokemon casuali.
     */
    fun randomFilters(): Pair<PokemonType, PokemonType> {

        // Variabili per memorizzare i tipi di pokemon casuali
        var randomPokemonType1: PokemonType
        var randomPokemonType2: PokemonType

        // Lista dei pokemon da filtrare per controllare non sia vuota
        var filteredList: List<PokemonEntity>

        // Controllo che i filtri non producano una lista vuota di pokemon
        do {
            randomPokemonType1 = PokemonType.getRandomPokemonType()
            randomPokemonType2 = PokemonType.getRandomPokemonType()
            filteredList = _pokemonList.filter { it.type1 == randomPokemonType1
&& it.type2 == randomPokemonType2 }
        } while (filteredList.isEmpty())
```

```
        return Pair(randomPokemonType1, randomPokemonType2)
    }

    /**
     * Aggiorna il tema selezionato dall'utente.
     *
     * @param isDarkTheme Valore booleano che indica se il tema è scuro.
     */
    fun saveDarkModePreference(isDarkTheme: Boolean) {
        UserPreferencesRepository.saveDarkModePreference(isDarkTheme)
    }

    /**
     * Aggiorna la lingua selezionata dall'utente.
     *
     * @param language Lingua selezionata dall'utente.
     */
    fun saveLanguagePreference(language: Language) {
        UserPreferencesRepository.saveLanguagePreference(language)
    }

}
```

9.3.3. PokemonDetailViewModel

Il `PokemonDetailViewModel` è specializzato nella gestione dei dati relativi a un singolo Pokémon, forniti alla relativa `View`. La sua funzione primaria è quella di caricare i dettagli completi di un Pokémon specifico quando viene selezionato dall'utente. Una volta caricati i dati, il `ViewModel` espone queste informazioni all'interfaccia utente attraverso un StateFlow, permettendo alla vista di aggiornarsi in modo reattivo. Inoltre, il ViewModel gestisce le operazioni relative ai preferiti dell'utente, consentendo di salvare il Pokémon corrente come preferito e di rimuovere l'ID del Pokémon preferito precedentemente salvato, aggiornando di conseguenza lo stato sia a livello locale che nel repository delle preferenze dell'utente.

```
/**
 * ViewModel per la schermata dei dettagli di un pokemon.
 *
 *  @param pokemonRepository Repository per le operazioni di database.
 */
class PokemonDetailViewModel(
    private val pokemonRepository: PokemonRepository,
): ViewModel() {

    // Pokemon da visualizzare
    private val _pokemon: MutableStateFlow<PokemonEntity?> =
MutableStateFlow(null)
    private val _detailsLoaded: MutableStateFlow<Boolean> =
MutableStateFlow(false)
```

```
    val pokemon: StateFlow<PokemonEntity?>
        get() = _pokemon.asStateFlow()

    val detailsLoaded: StateFlow<Boolean>
        get() = _detailsLoaded.asStateFlow()

    /**
     * Carica i dettagli di un pokemon.
     *
     * @param pokemonId Id del pokemon da caricare.
     */
    fun loadPokemon(pokemonId: Int) {
        // Specifica che i valori non sono ancora stati caricati
        _detailsLoaded.value = false
        viewModelScope.launch(Dispatchers.IO) {
            // Carica i dettagli del pokemon dal DB
            _pokemon.value = pokemonRepository.retrievePokemon(pokemonId,
PokemonRepository.LOAD_ALL)
            // Specifica che i valori sono ancora stati caricati
            _detailsLoaded.value = true
        }
    }

    /**
     * Salva il pokemon preferito dell'utente.
     * Il pokemonId viene preso dal pokemon correntemente visualizzato.
     */
    fun saveFavoritePokemon() {
        UserPreferencesRepository.saveFavoritePokemonId(pokemon.value!!.id)
        FavoritePokemonSharedRepository.updateFavoritePokemon(pokemon.value)
    }

    /**
     * Ritorna l'id del pokemon preferito dell'utente.
     *
     * @return Id del pokemon preferito dell'utente.
     */
    fun getFavoritePokemonId(): Int? {
        return UserPreferencesRepository.getFavoritePokemonId()
    }

    /**
     * Rimuove il pokemon preferito dell'utente.
     */
    fun clearFavoritePokemon() {
        UserPreferencesRepository.clearFavoritePokemonId()
        FavoritePokemonSharedRepository.updateFavoritePokemon(null)
    }

}
```

10. Tema e localizzazione

L'applicazione offre all'utente la possibilità di personalizzare l'interfaccia selezionando un tema (chiaro o scuro) e una lingua tra quelle supportate. Queste preferenze vengono memorizzate in modo persistente nelle `SharedPreferences` tramite l'apposito repository.
All'avvio dell'applicazione, la composizione dell'interfaccia utente viene avvolta da una funzione dedicata alla gestione del tema e della localizzazione. Questa funzione riceve in input le preferenze utente (tema e lingua) e il contesto dell'applicazione. In base al tema selezionato, viene applicata la corrispondente palette di colori; in assenza di una preferenza esplicita, viene utilizzato il tema di sistema. La lingua scelta, invece, viene impostata nel contesto dell'applicazione.
Successivamente, la composizione dell'interfaccia viene ulteriormente avvolta dalla funzione `MaterialTheme` di `Compose UI`.
In esso, il componente `CompositionLocalProvider` viene utilizzato per sostituire il contesto corrente con quello personalizzato, garantendo che tutte le funzioni `Composable` figlie ereditino le impostazioni del tema e della lingua.

```
/**
 * Composable per il tema e la lingua della UI.
 *
 * @param darkTheme Indica se il tema è scuro.
 * @param languageCode Il codice della lingua da utilizzare.
 * @param context Il contesto corrente dell'applicazione.
 * @param dynamicColor Indica se utilizzare i colori dinamici.
 * @param content Il contenuto della UI.
 */
@Composable
fun PokeWorldTheme(
   darkTheme: Boolean = isSystemInDarkTheme(),
   languageCode: String = Locale.getDefault().language,
   context: Context = LocalContext.current,
   // Dynamic color is available on Android 12+
   dynamicColor: Boolean = false,
   content: @Composable () -> Unit
) {

   // Ottiene il contesto con la lingua specificata
   val localizedContext = context.getLocalizedContext(languageCode)

   // Ottiene i colori personalizzati e il tipografia da utilizzare
   val themedColorsPalette = if(darkTheme) darkThemedColorsPalette else
lightThemedColorsPalette
   val themedTypography = mainThemedTypography

   // Imposta lo schema di colori di default per il tema
   val colorScheme = when {
       dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S ->
           if (darkTheme) dynamicDarkColorScheme(localizedContext) else
```

```
dynamicLightColorScheme(localizedContext)

        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }
    val view = LocalView.current
    if (!view.isInEditMode) {
        SideEffect {
            val window = (view.context as Activity).window
            window.statusBarColor = colorScheme.primary.toArgb()
            WindowCompat.getInsetsController(window,
view).isAppearanceLightStatusBars = darkTheme
        }
    }

    // Fornisce lo schema di colori personale e la tipografia ai componenti
dell'applicazione
    // Il contesto corrente dell'applicazione viene sovrascritto con quello
modificato
    CompositionLocalProvider(
        LocalContext provides localizedContext,
        LocalThemedColorsPalette provides themedColorsPalette,
        LocalThemedTypography provides themedTypography
    ) {
        MaterialTheme(
            colorScheme = colorScheme,
            typography = Typography,
            content = content
        )
    }
}

/**
* Ottiene un contesto con la lingua specificata.
*
* @param languageCode Il codice della lingua da utilizzare.
* @return Il contesto con la lingua specificata.
*/
fun Context.getLocalizedContext(languageCode: String): Context {
    val locale = Locale(languageCode)
    Locale.setDefault(locale)

    val config = Configuration(resources.configuration)
    config.setLocale(locale)
    config.setLayoutDirection(locale)

    return createConfigurationContext(config)
}
```

I valori del tema e della lingua sono memorizzati nell'Activity principale sotto forma di MutableStateFlow. Ogni volta che l'utente modifica una di queste preferenze, il valore corrispondente viene aggiornato nel MutableStateFlow, innescando così una ricostruzione dell'interfaccia utente grazie al meccanismo di reattività di Compose. Questo approccio assicura che le modifiche apportate dalle preferenze dell'utente si riflettano immediatamente nell'aspetto dell'applicazione.

```
/**
 * Classe principale dell'applicazione.
 */
@AndroidEntryPoint
class PokeWorld : ComponentActivity(){

   override fun onCreate(savedInstanceState: Bundle?){
       super.onCreate(savedInstanceState)

       // Inizializzazione

       setContent {

           val isSystemInDarkTheme = isSystemInDarkTheme()

           var isDarkTheme by rememberSaveable {
mutableStateOf(UserPreferencesRepository.getDarkModePreference(isSystemInDarkThe
me)) }
           var language by rememberSaveable {
mutableStateOf(UserPreferencesRepository.getLanguagePreference()) }

           PokeWorldTheme(
               darkTheme = isDarkTheme,
               languageCode = language.code,
               context = applicationContext
           ) {
             // UI
           }

   }
}
```

11. Aggiornamenti e future features

Il mondo dei Pokémon è in continua evoluzione e l'espansione del Pokédex è un fenomeno costante. L'applicazione è stata progettata con l'obiettivo di adattarsi agilmente a queste novità, garantendo un'integrazione rapida e fluida di nuovi Pokémon all'interno del database.

La struttura modulare e flessibile del sistema favorisce l'introduzione di nuove specie, senza richiedere modifiche sostanziali al codice esistente.
Un'ulteriore direzione di sviluppo prevede l'integrazione di funzionalità di realtà aumentata, consentendo agli utenti di visualizzare i Pokémon in un ambiente tridimensionale e di confrontarne le dimensioni con il mondo reale attraverso la fotocamera del dispositivo. Questa feature arricchirebbe significativamente l'esperienza utente, offrendo un'interazione più immersiva e coinvolgente con i Pokémon.
Infine, si prevede di implementare animazioni dinamiche e interattive per rappresentare le evoluzioni dei Pokémon, ispirandosi ai celebri giochi della saga. Questa funzionalità consentirebbe di visualizzare le diverse fasi evolutive di ogni specie, animando i cambiamenti morfologici e aggiungendo un elemento di gioco all'applicazione.