Relazione del progetto MeteoApp
realizzato da Giulia Vanni (giulia.vanni6@studio.unibo.it)

L'obiettivo di questo progetto è la realizzazione di un'applicazione che permetta all'utente di visualizzare in maniera semplice e intuitiva le previsioni del tempo della città scelta sia in tempo reale che nei successivi cinque giorni dal momento in cui viene effettuata la ricerca. Inoltre le città cercate vengono salvate ed è possibile taggare quelle preferite dall'utente che rimarranno sempre visibili in cima alla lista della cronologia nella schermata principale. Per ogni città salvata è anche possibile visualizzare la posizione sulla mappa.

Ciò che mi ha spinto a realizzare questa applicazione è la sua indubbia utilità, tanto che oggi le previsioni meteo sono integrate di default nei nostri dispositivi, senza più bisogno di installare un'app dedicata.

Di seguito vengono illustrati i componenti all'interno di ogni cartella e sottocartella con una breve spiegazione in modo da illustrare la struttura dei sorgenti.

App:

CustomApplication: questa classe viene utilizzata per configurare e inizializzare globalmente i componenti fondamentali dell'applicazione, in particolare l'istanza del database locale (Room), il *RepositoryProviderImpl* e il *UseCaseProvider.*

Data:

di:

RepositoryProviderImpl: è il punto centralizzato per fornire le istanze dei repository utilizzati nell'applicazione implementando il pattern Service Locator in maniera manuale. I repository inizializzati in questa classe sono: *CityRepository*, *WeatherRepository*, *ForecastRepository* e *SettingsRepository*.

local:

AppDatabase: permette di salvare i dati localmente tramite la libreria Room. Contiene la definizione della tabella *CityEntity* ed espone *CityDao.* Viene inizializzata una volta sola (pattern Singleton) per evitare problemi di prestazioni o accessi concorrenti.

CityDao: (Data Access Object) è l'interfaccia responsabile della comunicazione tra il database e il resto dell'applicazione, nel nostro caso questa classe manipola tramite query la tabella "cities". Viene istanziato da *AppDatabase* e utilizzato nel *CityRepositoryImpl* per permettere alla parte di domain di accedere ai dati delle citta.

CityEntity: è una classe annotata che rappresenta la tabella "cities" nel database SQLite. È parte della layer locale e viene mappata (tramite mapper) in una classe *City* del domain layer.

mapper:

CityMapper: serve a convertire i dati tra i vari livelli dell'architettura. Il livello Data usa oggetti *CityEntity* legati al database, mentre il livello Domain usa oggetti *City*.

Remote:

model:

NominatimResult: rappresenta il risultato di una ricerca geografica ottenuta dall'API Nominatim, contenente il nome completo del luogo (*display_name*) e le coordinate di latitudine (*lat*) e longitudine (*lon*) in formato stringa.

repository:

CityRepositoryImpl: è l'implementazione concreta dell'interfaccia *CityRepository*. Serve a gestire tutte le operazioni sui dati relativi alle città, come il salvataggio, l'eliminazione, l'aggiornamento e la lettura. Si collega a *CityDao* per accedere al database Room, fungendo da ponte tra il livello Domain e il database, grazie anche all'utilizzo dei mapper per la conversione dei dati.

ForecastRepositoryImpl: fornisce un'implementazione dell'interfaccia *ForecastRepository*. Utilizza l'API di OpenWeather per recuperare i dati meteo in tempo reale. Si occupa di inviare le richieste HTTP, gestire le risposte e trasformare i dati ricevuti in oggetti del dominio.

SettingsRepositoryImpl: è l'implementazione dell'interfaccia *SettingRepository.* Gestisce la memorizzazione e il recupero delle impostazioni dell'utente, come l'unità di misura della temperatura, la lingua e il tema. Utilizza SharedPreferences per garantire che le preferenze vengano mantenute tra una sessione e l'altra.

WeatherRepositoryImpl: implementa l'interfaccia *WeatherRepositoryImpl* definita nel modulo Domain e si occupa della gestione dei dati meteo correnti. Utilizza l'API di OpenWeather per ottenere informazioni aggiornate sul tempo atmosferico, i dati vengono trasformati in modelli di dominio, rendendoli utilizzabili dal resto dell'applicazione.

Domain:

di:

RepositroryProvider: si occupa di creare e fornire le istanze dei repository usati nell'applicazione. Si occupa della creazione e dell'iniezione delle implementazioni concrete come *WeatherRepositoryImpl* e *ForecastRepositoryImpl,* garantendo che i componenti dell'applicazione ricevano le dipendenze corrette.

UseCasePovider: fornisce le istanze dei casi d'uso dell'applicazione. Si occupa di creare gli oggetti che contengono la logica per eseguire operazioni specifiche, come ottenere il meteo o salvare le impostazioni.

model:

City: rappresenta un modello di dominio per una città. Contiene le informazioni principali come il nome, la latitudine, la longitudine e un flag *isFavorite*. Questa classe viene utilizzata per gestire e visualizzare i dati relativi alle città all'interno dell'applicazione.

DailyForecast: rappresenta le previsioni meteo giornaliere per una determinata data. Contiene la data, la temperatura meteo del giorno, una breve descrizione delle condizioni meteorologiche e l'URL dell'icona corrispondente.

ForecastItem: rappresenta un singolo elemento di previsione meteo, contenente informazioni come la data, la temperatura, la descrizione del tempo, l'icona, l'umidità, la pressione e la velocità del vento. Implementa l'interfaccia Parcelable, permettendo di essere facilmente passata tra attività, in questo caso *ForecastActivity* e *DetailedForecastActivity.*

WeatherInfo: contiene le informazioni principali sul meteo di una città e include anche le coordinate geografiche.

repository:

CityRepository: definisce le operazioni principali per gestire le città all'interno dell'applicazione. Include metodi per inserire, recuperare, aggiornare e cancellare città.

ForecastRepository: definisce il metodo per ottenere le previsioni meteo per una città specifica. Il metodo getForecast prende in input il nome della città, la lingua e la chiave API, e restituisce un risultato contenente una lista di oggetti *ForecastItem*.

SettingsRepository: definisce i metodi per gestire le impostazioni dell'applicazione, come lingua, il tema, e l'unità di temperatura.

WeatherRepository: definisce i metodi per ottenere le informazioni meteo aggiornate. Permette di recuperare i dati sia tramite il nome della città che attraverso le coordinate geografiche.

usecase:

DeleteCityUseCase: gestisce la rimozione di una città salvata. Utilizza *CityRepository* per eliminare la città selezionata dell'utente.

FetchForecastUseCase: recupera le previsioni meteo per una città specifica usando *ForecastRepository*. Restituisce una lista di previsioni giornaliere.

FetchWeatherUseCase: ottiene le informazioni meteo correnti in base al nome della città tramite il *WeatherRepository.*

FetchWeatherByCoordinatesUseCase: recupera i dati meteo correnti usando le coordinate geografiche (latitudine e longitudine), utile per ottenere il meteo basato sulla posizione dell'utente.

GetAppLanguageUseCase: restituisce la lingua attualmente impostata nell'applicazione tramite il *SettingsRepository.*

GetSavedCitiesUseCase: recupera tutte le città salvate dall'utente tramite il *CityRepository,* ad esempio per mostrarle in una lista.

GetTempUnitUseCase: restituisce l'unità di temperatura selezionata (Celsius o Fahrenheit) tramite il *SettingsRepository.*

SaveCityUseCase: gestisce il salvataggio di una nuova città nella memoria locale, utilizzando il *CityRepository*.

SetTempUnitUseCase: aggiorna l'unità di misura della temperatura preferita dall'utente tramite il *SettingsRepository.*

ToggleFavoriteCityUseCase: modifica lo stato di una città (aggiunta o rimozione dai preferiti) agendo sul campo *isFavorite* dell'oggetto *City*.

ui:

detailedforecast:

DetailedForecastActivity: mostra i dettagli meteo di una singola giornata, compresi temperatura, umidità, pressione e velocità del vento. Riceve i dati tramite Intent dalla *ForecastActivity* e li presenta all'utente. Comunica con *DetailedForecastViewModel* per recuperare e osservare i dati.

DetailedForecastViewModel: gestisce la logica di presentazione per la schermata che mostra i dettagli di una giornata specifica nelle previsioni meteo. Il suo compito principale è formattare la data ricevuta in un formato più leggibile e fornire alla view la lista dei dati orari (oggetti *ForecastItem*) corrispondenti.

ThreeHourForecastAdapter: è un adapter per RecyclerView utilizzato nella *DetailedForecastActivity* per visualizzare le previsioni meteo suddivise in intervalli di tre ore. Prende in input una lista di oggetti *ForecastItem,* ognuno dei quali rappresenta una previsione per una specifica ora del giorno.

mainactivity:

Cityadapter: è un componente della UI che gestisce la visualizzazione e l'interazione con l'elenco delle città salvate dall'utente. Estende *RecyclerView.Adapter* e lavora con un dataset composto da oggetti *CityEntity*, che vengono convertiti in oggetti City tramite il metodo *toDomain()* nel metodo onBindViewHolder.

MainActivity: è l'attività principale dell'app e funge da dashboard per visualizzare le città salvate, cercarne di nuove tramite l'API Nominatim *(searchCities(...))* e ottenere il meteo in tempo reale tramite geolocalizzazione. La UI è gestita tramite *ActivityMainBinding,* con un RecyclerView popolato da *CityAdapter*. I risultati della ricerca vengono mostrati con un ArrayAdapter, e la posizione dell'utente è ottenuta tramite *FusedLocationProviderClient*, con gestione dei permessi e recupero automatico della posizione *(checkLocationPermissionAndFetchWeather(), getLastKnownLocation())*.

MainViewModel: rappresenta il collegamento tra la MainActivity e il dominio dell'applicazione, occupandosi della gestione dei dati relativi alle città salvate e alle informazioni meteo. Viene inizializzato con diversi use case che incapsulano la logica di accesso e manipolazione dei dati. I risultati di queste operazioni sono esposti tramite oggetti LiveData, osservabili dalla UI, come cities, weather ed error. Tutte le chiamate vengono eseguite in coroutine all'interno del *viewModelScope*, garantendo un'esecuzione asincrona e sicura rispetto al ciclo di vita dell'applicazione.

MainViewModelFactory: è responsabile della creazione istanze di *MainViewModel*, fornendo tutti i parametri necessari tramite il costruttore.

forecast:

DailyForecastAdapter: è un adapter per un RecyclerView che mostra una lista di previsioni meteo giornaliere. Riceve una lista di oggetti DailyForecast e una funzione callback da chiamare al click su un elemento. Il metodo updateData permette di aggiornare i dati e notificare l'adapter per rifare il rendering.

ForecastActivity: è responsabile di mostrare le previsioni meteo giornaliere per una città specifica, passata tramite intent. All'avvio, inizializza la UI con un RecyclerView e una TextView per il nome della città. Imposta il RecyclerView con un *DailyForecastAdapter* che gestisce il click su ogni elemento per aprire un'activity di dettaglio con le previsioni più dettagliate del giorno selezionato. Il *ForecastViewMode*l viene creato con gli use case necessari e osserva i dati delle previsioni (*dailyForecastList)* e degli errori, aggiornando l'adapter o mostrando toast in caso di problemi.

ForecastViewModel: carica le previsioni meteo per una città usando f*etchForecastUseCase* e la lingua impostata. Espone due LiveData: una lista completa di previsioni e una sintesi giornaliera con temperatura media e condizione principale. I dati sono elaborati in modo asincrono con *viewModelScope*. In caso di errore, aggiorna un LiveData con il messaggio di errore.

ForecastViewModelFactory: è una factory che crea istanze di *ForecastViewModel,* fornendo i necessari use case per il caricamento delle previsioni meteo e la gestione della lingua dell'app.

map:

MapActivity: mostra una mappa utilizzando la libreria osmdroid. Al momento della creazione, carica la configurazione di osmdroid, riceve latitudine, longitudine e nome della città tramite intent, e imposta la vista della mappa centrata su queste coordinate con uno zoom predefinito

settings:

SettingsActivity: gestisce l'interfaccia per modificare le impostazioni dell'app, come lingua, tema e unità di temperatura.

SettingsViewModel: gestisce il caricamento e il salvataggio delle impostazioni dell'app tramite il repository. Espone le impostazioni correnti (lingua, tema e unità di temperatura) come LiveData osservabili dalla UI, permettendo di aggiornare l'interfaccia in modo reattivo.

SettingsViewModelFactory: crea istanze di *SettingsViewModel* fornendogli l'oggetto repository, che è responsabile di gestire i dati delle impostazioni.

utils:

TemperatureUtils: contiene una funzione che converte una temperatura espressa in gradi Celsius nell'unità di misura preferita dall'utente, memorizzata nelle SharedPreferences.

BaseActivity: personalizza il contesto e il tema di tutte le attività che la estendono. Nel metodo *attachBaseContext* imposta la lingua dell'app leggendo la preferenza salvata e aggiornando la configurazione locale di Android di conseguenza. Nel metodo *onCreate* invece applica il tema scelto dall'utente (chiaro, scuro o predefinito di sistema) prima di chiamare il metodo *super.onCreate,* così che tutte le attività ereditate abbiano lingua e tema corretti fin dall'avvio.

SplashActivity: mostra un'animazione Lottie all'avvio dell'app, impostandola in loop. Dopo un ritardo di 2 secondi gestito con una coroutine, avvia *MainActivity* e termina se stessa. Questo assicura una transizione fluida senza bloccare il thread principale.

Punti di forza:

L'applicazione è stata progettata seguendo i principi della Clean Architecture. Questo approccio migliora la manutenibilità, la testabilità e la scalabilità del progetto. Per la parte di presentazione è stato adottato il pattern MVVM che garantisce una separazione chiara tra l'interfaccia utente e la logica di presentazione. I ViewModel utilizzano LiveData per esporre lo stato osservabile alla UI, mentre le operazioni asincrone vengono gestite con le coroutine Kotlin tramite *viewModelScope,* garantendo un'esecuzione non bloccante e sicura rispetto al ciclo di vita. L'iniezione delle dipendenze avviene tramite factory personalizzate per i ViewModel, evitando accoppiamenti stretti e migliorando la riusabilità dei componenti. Questo insieme di scelte architetturali consente di ottenere un'app robusta, modulare e facilmente migliorabile con aggiornamenti.

Possibili Migliorie:

- Caching locale dei dati meteo: salvare le previsioni in locale per offrire accesso offline e ridurre le chiamate API
- Aggiunta di test automatizzati per aumentare la robustezza dell'applicazione
- Implementare mappa utilizzando l'API di Google Maps (API inizialmente implementata , ma si è preferito optare per delle mappe opensource con OSM per via dei limiti di quota nel piano gratuito di Google Console)
- Migliorare la UX con animazioni fluide e aggiunta di feedback visivo