Presentazione DietiDeals24

# Scelte implementative

## Interfaccia grafica

Il logo dell’applicazione è un’immagine che richiama i mondi dell’informatica e del mercato delle aste: un monitor simboleggia la natura informatizzata dei servizi offerti, mentre il martelletto che compare sullo schermo rimanda al mondo del mercato delle aste.

I colori principali dominanti nell’interfaccia sono l’arancione e il blu:

* L’arancione è spesso associato alla creatività, al cambiamento (dovuto al suo rimando ai colori autunnali), ma anche a salubrità e vitalità. Essendo un colore caldo, ricorda passione, felicità, entusiasmo, energia. La sua scelta risiede nella volontà di associare la nostra applicazione a un’opera creativa e vivace, capace di catturare l’utente.
* Il blu, complementare dell’arancione, è spesso utilizzato per trasmettere calma e responsabilità. In particolare, i toni più scuri sono spesso scelti dai brand premium per conferire un tocco di forza ed affidabilità allo stesso. Allo stesso modo, abbiamo scelto il blu scuro per far percepire la nostra applicazione come affidabile e professionale.
* Lo sfondo impiega un colore grigio tendente al blu, risultando in una sfumatura molto chiara di quest’ultimo. Questa scelta permette sia di ridurre il fastidio visivo, sia di donare un senso di calma.
* Altri colori impiegati sono il rosso, spesso associato al pericolo e l’importanza, utilizzato principalmente per richiamare l’attenzione su elementi dell’interfaccia che comportano azioni importanti o irreversibili, ma anche per sottolineare messaggi di errore dell’applicazione.
* Il blu chiaro sottolineato è stato utilizzato per le parole (o frasi) cliccabili, essendo questa una convenzione ben radicata nell’ambito delle interfacce grafiche.
* Il grigio ed il bianco mantengono una funzione conforme ai normali standard di realizzazione dell’interfaccia:
  + Il grigio è utilizzato per quei campi con i quali non si può interagire.
  + Il bianco viene sfruttato per evidenziare campi con i quali è possibile interagire.

È stato cercato di rispettare il più possibile le euristiche di Nielsen. Ad esempio:

* L'applicazione permette agli utenti di effettuare delle operazioni con un numero estremamente basso di tap e re-direzioni.
* Ha un'interfaccia minimale senza troppi ingombri visivi che possono sovraccaricare la memoria a breve termine dell'utente.
* È stata fornita una sezione di aiuto interna che permette all'utente di capire come effettuare passo per passo ogni operazione più importante.
* Impedisce all'utente di effettuare delle operazioni importanti senza prima richiedere una conferma, così da consentire di cambiare idea oppure prevenire errori di tap.
* Backward recovery, ossia impedire a monte che l’utente effettui un’operazione non valida così da evitare gli errori, accompagnato da messaggi che spiegano perché l’operazione non possa essere effettuata.

## Architettura

Si è optato, in questo sistema, per un'architettura Client-Server a due livelli con modello client leggero e una base di dati relazionale per la persistenza dei dati. Questo perché:

* Ogni modulo adempie ad una funzione ben precisa. Ovvero:
  + I client si occupa dell’interazione con l’utente e invia le richieste al server (questo è possibile perché i client conoscono le interfacce del server).
  + Il server si occupa di gestire le richieste pervenute dal Client, far rispettare i vincoli esposti dalla business logic, interagire con la base di dati relazionale e inviare le risposte al client.
  + Il client elabora la risposta ricevuta dal server.
* In questo modo è stato possibile sviluppare client e server parallelamente.
* Il client si occupa solamente dell'interfaccia e l'interazione con l'utente, e quindi può essere eseguito su gran parte dei dispositivi, anche con modeste prestazioni. Lo svantaggio di tale scelta è che pone maggior carico di elaborazione sul server.

Il client è un'applicazione per sistema operativo Android, scritta in linguaggio Kotlin.

Il server si articola in:

* Un'applicazione Java che adotta lo Spring Framework, salvata in un'immagine ed eseguita in un container Docker, installato su una macchina virtuale fornita da AWS EC2.
* Una base di dati relazionale PostgreSQL, al cui interno sono conservati tutti i dati dell'applicazione, eseguita anch'essa come immagine Docker sulla stessa macchina virtuale AWS EC2.

I due container comunicano grazie all'utilizzo di Docker Compose.

* Abbiamo scelto Android perché quasi tutti ormai posseggono uno smartphone. Ed essendo un sistema operativo così diffuso (conta circa il 69.94% del mercato dei sistemi operativi per dispositivi mobili) Android è la scelta giusta per rendere l'applicazione disponibile quasi ovunque ed accessibile a chiunque, attraverso Chromebook, smartphone, automobili, smartwatch, smart TV.
* Abbiamo scelto Kotlin perché riduce la quantità di codice con conseguente maggiore leggibilità, supporta lo sviluppo multipiattaforma, poiché le librerie Kotlin possono essere utilizzabili anche per lo sviluppo di app iOS, desktop e web ed è molto semplice da apprendere ed utilizzare, nonché presenta una serie di agevolazioni rispetto a Java per facilitare la gestione della nullità dei riferimenti e delle operazioni asincrone. Infine è anche pienamente compatibile con le librerie ed il codice Java essendo eseguiti entrambi su una JVM.
* Abbiamo scelto AWS in quanto è senza eguali in termini di maturità, affidabilità e sicurezza. In AWS le funzionalità offerte sono più ricche rispetto alla concorrenza. L'utilizzo di AWS permette di tagliare i costi, infatti il Free Tier è un'ottima opportunità per piccole aziende che richiedono modesti strumenti per il rilascio della propria applicazione. A causa della sua ampia personalizzazione e grazie alla possibilità di creare, lanciare, e fermare le istanze delle macchine virtuali sul server su richiesta, pagando solo il tempo di computazione sfruttato, l'elasticità del servizio è indubbia. AWS Cognito semplifica il processo di autenticazione, delegando a un authorization server tutta la gestione del sign-in, sign-up e autenticazione.
* Abbiamo scelto Java perché, essendo un linguaggio eseguito su macchina virtuale, assicura isolamento dalla macchina sottostante, impedendo l'esecuzione di codice malevolo e aumentando la sicurezza e la portabilità, poiché il codice viene trasformato in un codice universalmente comprensibile da tutte le implementazioni della JVM sui vari dispositivi. Il linguaggio è anche ormai un caposaldo nel mondo dello sviluppo software grazie alla sua stabilità, la ricca libreria standard nonché supporto di terze parti.
* Abbiamo scelto Docker perché permette di impacchettare all'interno di un'immagine Docker l'applicazione e tutto l'ambiente necessario alla sua esecuzione. L'immagine può essere caricata sulla repository centrale di Docker affinché possa essere distribuita. Ogni container è isolato e contiene tutto ciò che è necessario, e non c'è bisogno di installare applicazioni oltre al Docker Engine.
* Abbiamo scelto PostgreSQL perché ha una forte reputazione di RDBMS open source robusto, affidabile, estensibile. Esso è eseguibile su tutti i principali sistemi operativi, garantisce le proprietà ACID (atomicità, consistenza, isolamento, durabilità) per dati e transazioni, e il suo linguaggio procedurale, PL\pgSQL, è pienamente compatibile con quello delle altre basi di dati, facilitando una eventuale migrazione futura.
* Abbiamo adottato i principi REST perché permette di spingere al massimo l'indipendenza tra i vari servizi, non rinunciando però ad una semplicità di fondo nella comunicazione, essendo il protocollo HTTP un protocollo maturo, quindi testato e funzionante.
* Abbiamo scelto Spring Framework, con l’aiuto di Spring Boot, perché permette di avere un’alta modularità e basso livello di dipendenze grazie al potente meccanismo di dependency injection, che promuove un basso accoppiamento, facilitando manutenzione e riusabilità e flessibilità nella scelta di specifici componenti e moduli in base alle necessità del progetto. Sping Boot agevola il processo di configurazione dello Spring Framework attraverso una serie di convenzioni che promuovono le best practices.

## Design

### Front-end

La scelta dell’architettura per il front-end viene qui discussa poiché meno canonica.

Nel complesso, è un ibrido tra il pattern MVVM (Model-View-ViewModel) e MVC (Model-View-Controller).

La motivazione è una maggiore suddivisione tra le responsabilità di elaborazione e quelle di recupero ed aggiornamento dei dati.

Distinguiamo due strati:

* Lo strato della UI, che comprende View, StateHolder (che mantengono i dati sullo stato della View, come i ViewModel) e Controller e si occupa di gestire l’interfaccia e l’interazione con l’utente.
  + Le View rappresentano ciò che l’utente vede a schermo.
  + I ViewModel mantengono i dati in caso di cambiamenti di stato (rotazione o spegnimento schermo), recuperano i dati paginati e convalidano l’input dell’utente.
  + I Controller creano e mostrano messaggi di errore, gestiscono funzionalità come autenticazione e navigazione, osservano i cambiamenti nel ViewModel e aggiornano le View.
* Lo strato dei dati è rappresentato dal Model e il suo ruolo è comunicare con le fonti di dati, siano esse esterne (come basi di dati) che locali (come cache o basi di dati interne). Il Model, a sua volta, è e organizzato secondo il pattern Repository, con tre sottostrati: Repository, Service e ServiceImpl.
  + Le classi Repository rappresentano una enumerazione delle operazioni che la logica dell’applicazione può effettuare sui dati.
  + Le classi Service rappresentano una interfaccia che astrae queste operazioni sui dati. In questo caso, esse rappresentano una interfaccia di metodi che verranno utilizzati per la creazione di richieste HTTP.
  + Le classi ServiceImpl rappresentano dei metodi concreti- implementati dalla libreria Retrofit- che, sulla base delle annotazioni e i parametri inseriti nelle interfacce dei metodi di Service, creeranno la giusta richiesta HTTP.

### Back-end

L’architettura adottata per il back-end è un’architettura di tipo esagonale. L’obiettivo di tale architettura è di definire una separazione netta tra la business logic del dominio (anche detto domain layer) e tutto ciò che sono i fattori esterni, ovvero il layer di interfaccia con l’utente (anche detto application layer) e il layer di persistenza (anche detto infrastracture layer), attraverso una serie di interfacce ben definite. Il design pattern adottato per il domain layer è di tipo Domain Driven. Questo ci permette di avere un’alta coesione e ridurre al minimo l’accoppiamento tra le classi della business logic, così che il loro unico motivo per cambiare sia un cambiamento nell’entità del dominio corrispondente.

* L’application layer contiene le classi responsabili di esporre le REST API che verranno chiamate dall’utente. È dipendente solo e unicamente da classi DTO, che si occupano di raccogliere le informazioni ricevute sottoforma di HTTP Request e saranno tradotte in HTTP Response.
* Il domain layer contiene le classi responsabili della business logic, che dipende solo e unicamente dalle classi del dominio, ovvero le entità. Al suo interno troviamo anche classi responsabili di tradurre i DTO in entità del dominio. Inoltre, grazie all’interfaccia offerta dalla Java Persistance API, le classi necessarie per la comunicazione con il persistance layer sono gestite da Spring Boot attraverso una Spring dependency.
* Il persistance layer contiene i driver responsabili della comunicazione con il database.