



TRUCCO PARRUCCO

# PROGETTO INGEGNERIA DEL SOFTWARE

Marco Vinciguerra, David Guzmán Piedrahita

2021-2022

# CONTENUTI

## Documentazione e sviluppo

01	02	03	04	05	06
Life cycle	Requirements e quality	Architecture (global design)	Design (detailed design)	Testing	Maintenance

## TECH STACK

# TECNOLOGIE E LINGUAGGI UTILIZZATI

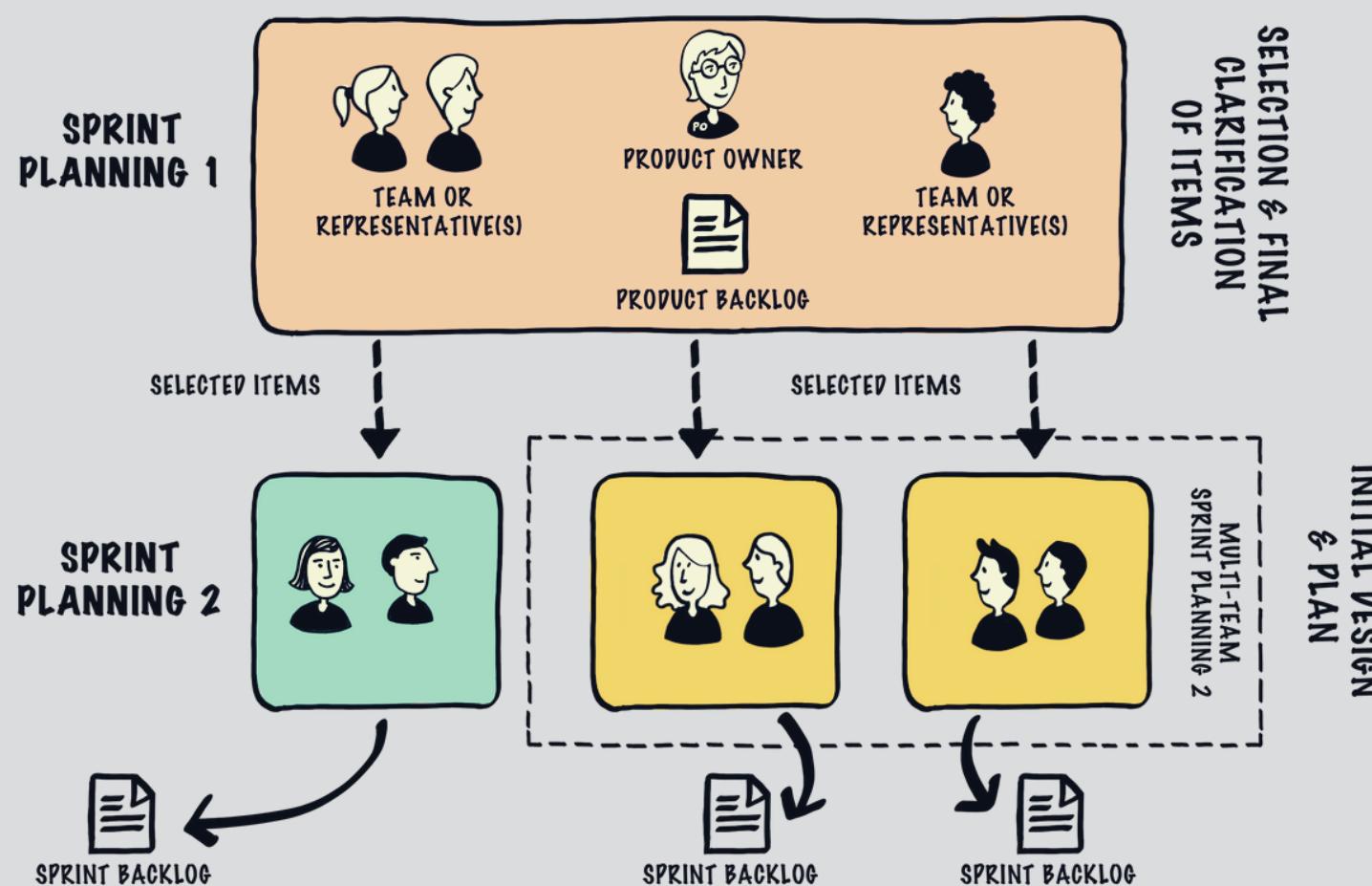
**che costituiscono la base sulla quale si costruisce l'app**

- Flutter & Dart per la scrittura del codice
- StarUML per i diagrammi UML
- Github per la gestione dei file
- Vim e Visual studio code come IDE per la programmazione



# SOFTWARE LIFE CYCLE

Tecnica di sviluppo utilizzata: SCRUM.



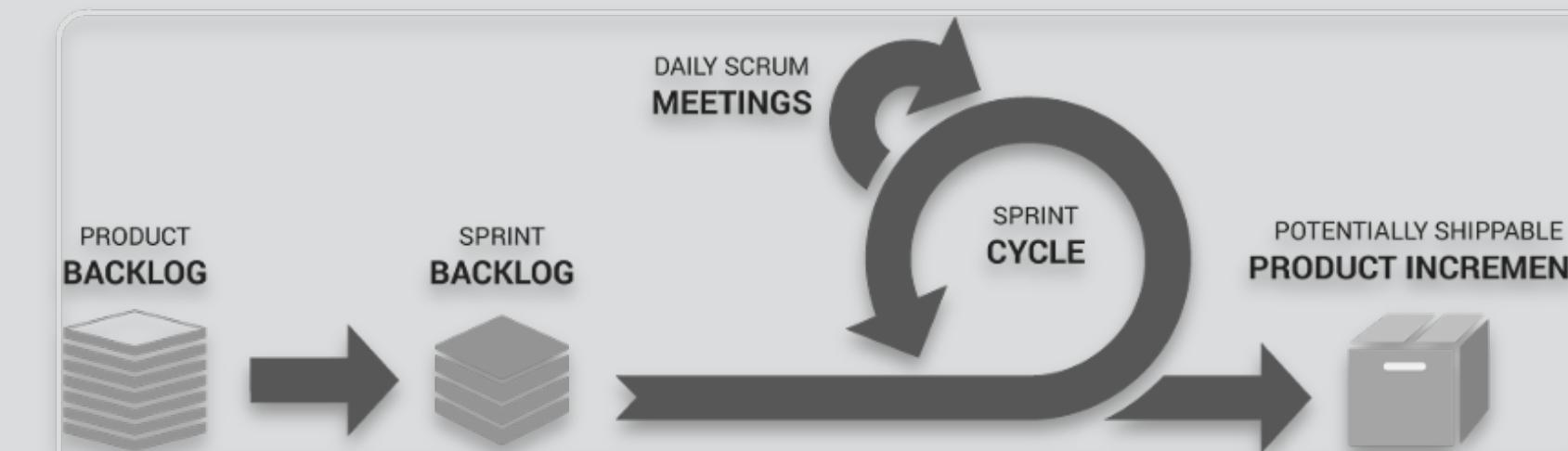
## Scrum life cycle: ruoli, backlog, sprint

**Ruoli:** i tre ruoli o gruppi fondamentali di scrum, al posto di essere fissi, sono stati alternati tra i team member, dovuto alla sua dimensione.

- **Product owner**
- **Development team**
- **Scrum master**

**Backlog:** il backlog globale è stato completato durante 4 sprint, ognuno con il suo corrispettivo sprint Backlog.

**Daily scrum e altre riunioni:** sono state fatte ogni giorno, con un breve report per gli incontri più importanti.



# PEOPLE MANAGEMENT AND TEAM ORGANIZATION

**La configurazione organizzativa del team condivide aspetti Mintzberg di:**

- **Simple structure**, direct supervision come coordination mechanism.
  - **Adhocracy**, mutual adjustment come coordination mechanism.
- Usando scrum, esistono alcuni pochi ruoli gerchici che comporterebbero un elemento di direct supervision, ma la ridotta dimensione del team e l'alternanza dei ruoli fa sì che, nella pratica, prevalga la mutual adjustment.



# Funzionali

categorizzati a seconda la priorità Kano e Moscow.

(Kano, Moscow)

- Possibilità di scegliere i diversi tipi di acconciature e in base al tipo di acconciatura scegliere la durata della prenotazione (**Attactive, Must have**)
- Possibilità di registrarsi per la prima volta al sito da parte di un cliente. (**Must-be, should have**)
- Utilizzo di un database per la gestione dei clienti. In alternativa si potrebbero utilizzare variabili per gestire le prenotazioni. (**Must-be, should have**)
- Creazione di un profilo base utente per le prenotazioni. (**Must-be, should have**)
- Che il sistema delle prenotazioni non funzioni correttamente e che si accavallino sulla stessa fascia oraria. (**One-Dimensional, should have**)
- Possibilità di mettere recensioni da parte dell'utente. (**Questionable, could have**).

# REQUIREMENTS E QUALITY



# REQUIREMENTS NON FUNZIONALI

## Punti fondamentali McCall

- **Usability:** Il prodotto finale non deve essere difficile da utilizzare e la user experience deve essere il più semplice possibile per l'utente in quanto non deve essere necessariamente esperto di coding. Ad esempio i bottoni sono studiati per essere il più semplici e intuitivi possibile. Dal punto di vista delle prestazioni deve essere il più reattivo possibile e facile da capire.
- **Efficiency:** si vuole creare il programma più efficiente possibile che utilizzi il minimo delle risorse del dispositivo da cui si utilizza l'applicazione. Per migliorare l'efficienza si `e usato per esempio la variabile const nella UI.
- **Maintanability:** Il codice deve essere scritto nel modo più conforme alle regole di standard di programmazione per favorire la leggibilità e la successiva manutenzione.

# CRITERI DI QUALITÀ

Sono stati utilizzati i dart code metrics, in particolare:

- **Miglioramento delle performance del codice:** dart analyze
- **Calcolo complessità:** flutter pub run dart code metrics:metrics analyze lib
- **File non utilizzati:** flutter pub run dart code metrics:metrics check-unused-files lib

```
dart_code_metrics:  
anti-patterns:  
- long-method  
- long-parameter-list  
metrics:  
cyclomatic-complexity: 20  
maximum-nesting-level: 5  
number-of-parameters: 4  
source-lines-of-code: 50  
metrics-exclude:  
- test/**
```

# ARCHITECTURE (GLOBAL DESING)

## Dsicussion points

- Architecture design methods
- diagramma dei componenti
- stile architetturale
- IEEE 1471



# INTERAZIONE CON SERVIZI ESTERNI

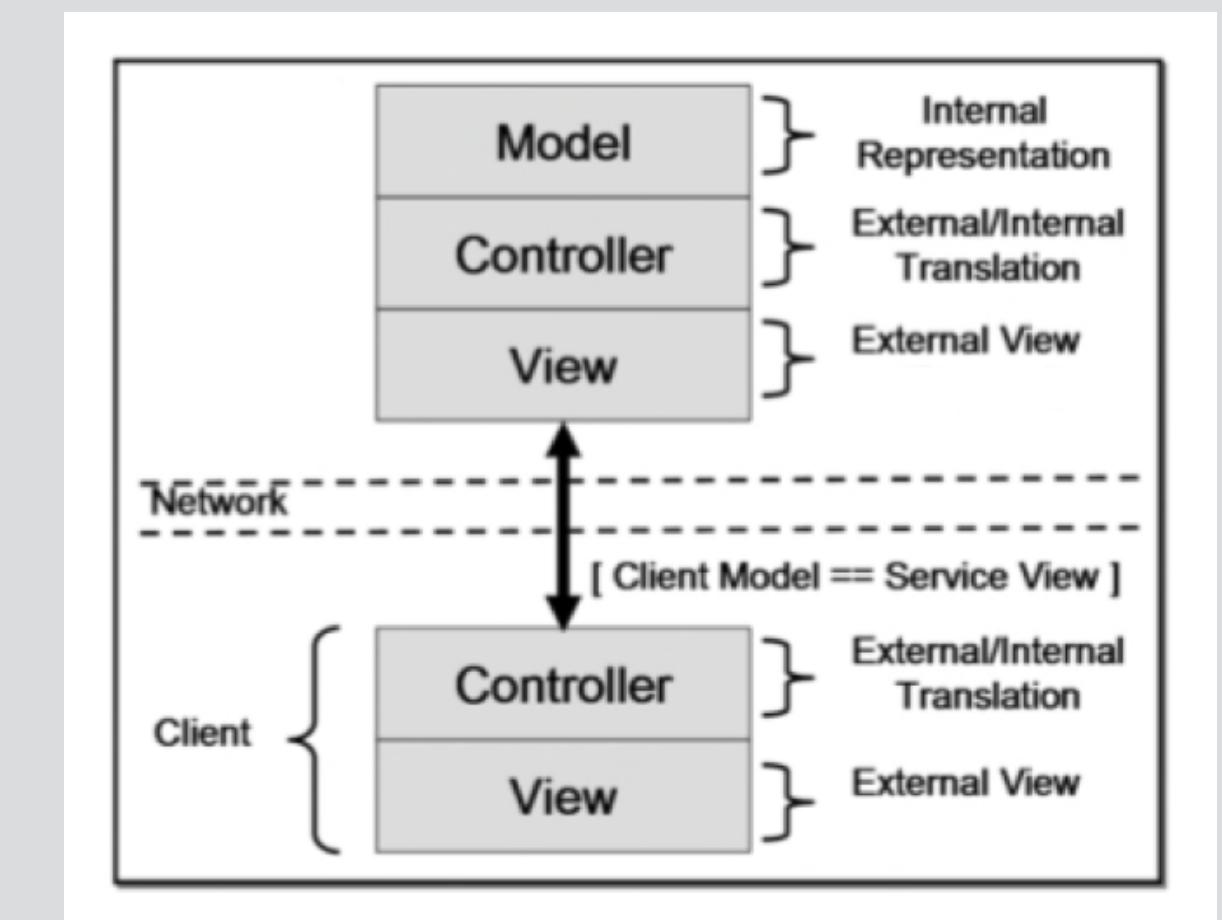
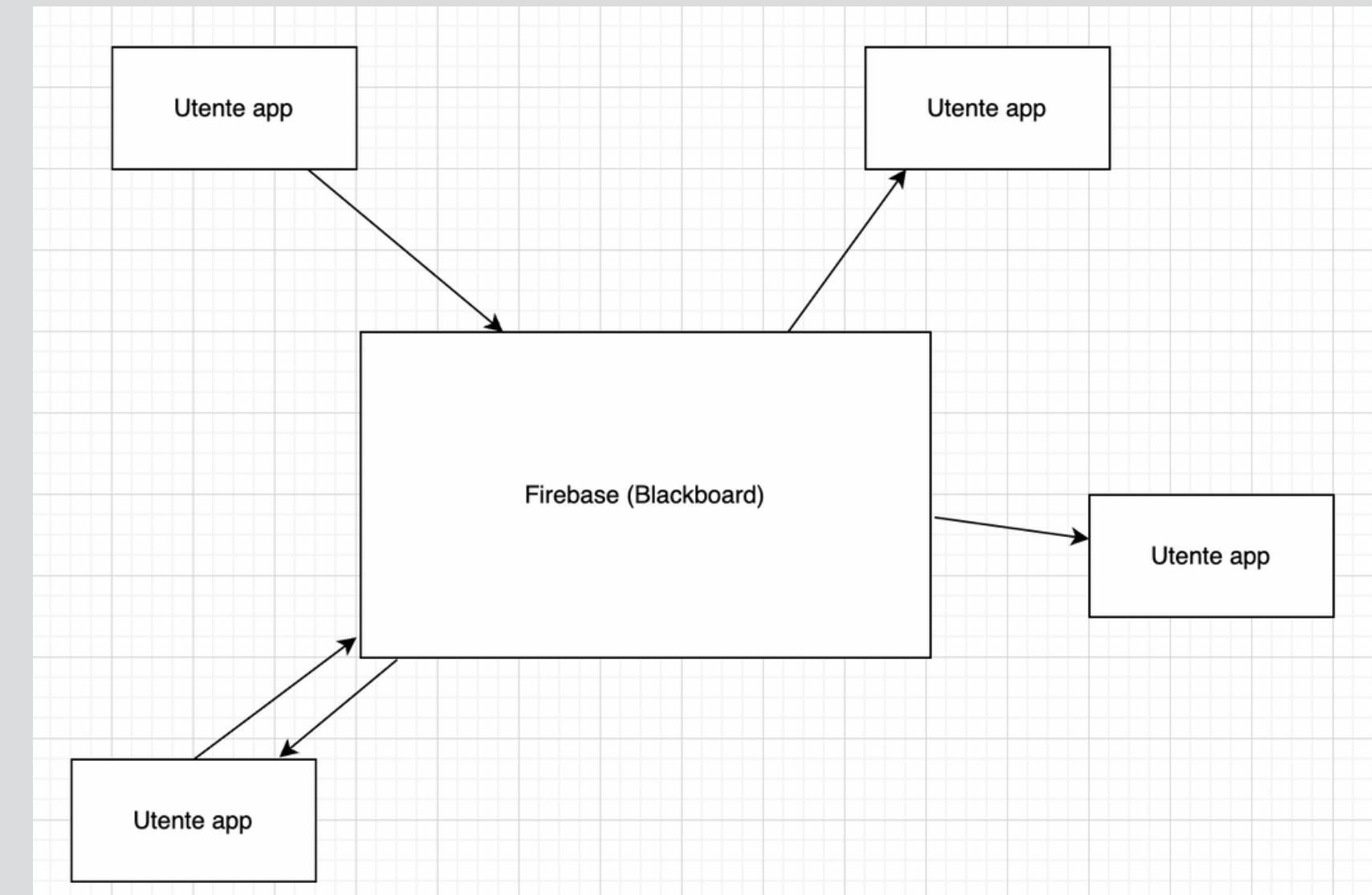
**Firebase Realtime Database, Firebase authentication service/**

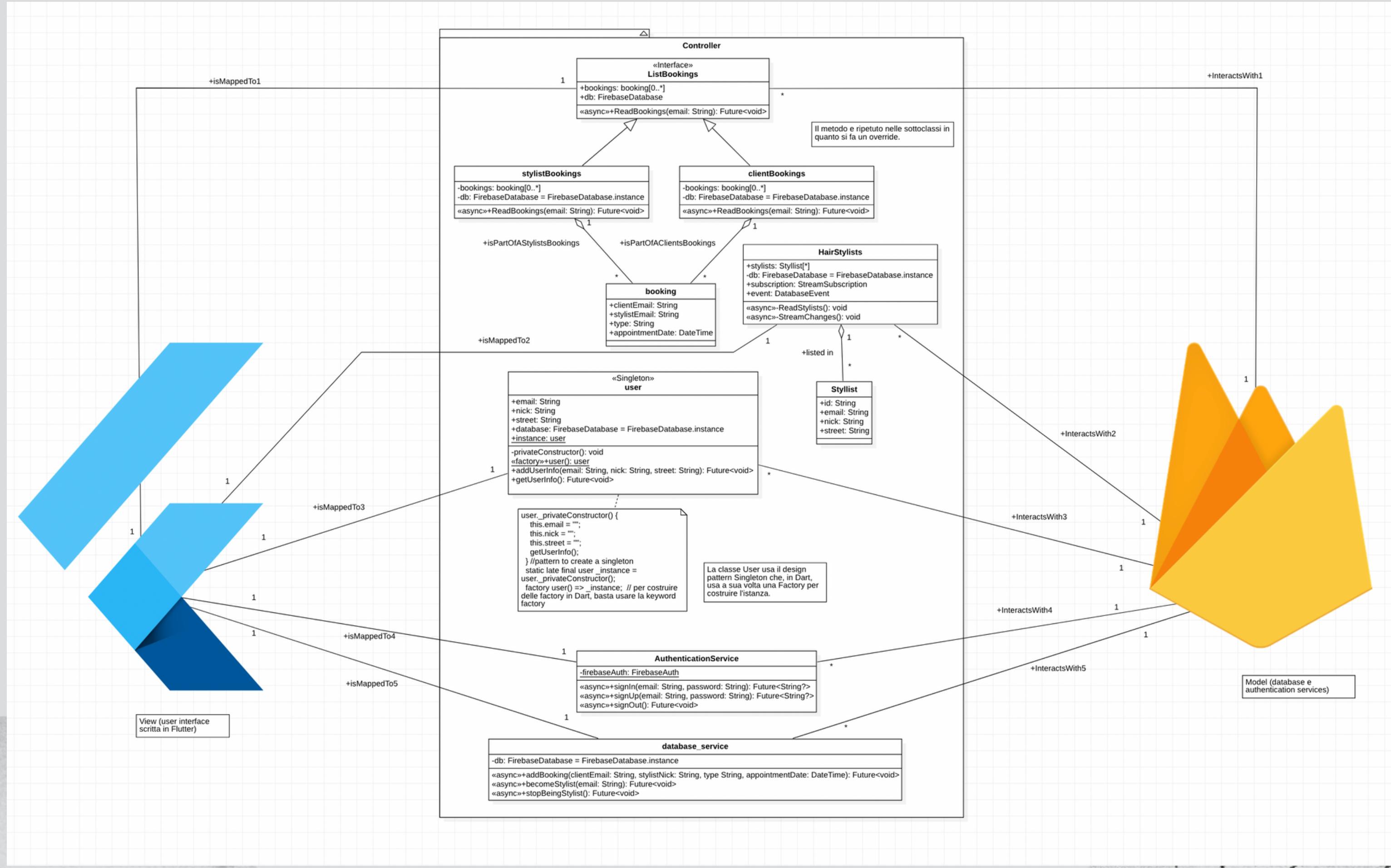
- **Repository style**
- **Controller + view + remote model:**

**Controller:** il controller, a differenza del model, è implementato come parte del client vero e proprio, ed è stato costruito riflettendo, per quanto possibile, le diverse entità del mondo reale.

**View:** interamente costruita utilizzando i servizi offerti dal framework flutter, dove tutti i diversi tasti e pagine sono rappresentate da cosiddetti widget, che a sua volta sono delle classi dart.

**Model:** si appoggia a servizi esterni e standardizzati per modellizzare i dati del mondo reale.





# DESIGN (DETAILED DESING)

## Dsicussion points

- Funzionamento delle classi Dart
- diagramma dei componenti
- Strategie standard del linguaggio Dart e il Framework Flutter
- Design pattern
- Design methods



# DESIGN PATTERN

## Specifici di Dart/Flutter e generici

- **Provider**
- **Futures**
- **Singleton e factory**: per gestire le informazioni dell'utente che, come nel caso di altre informazioni, pu'o essere disponibile sia in locale che nei server, si usa una classe singleton di nome User che pertanto pu'o essere istanziata una sola volta.
- **Abstraction occurrence**: questo design pattern è utilizzato nel caso dei singoli parrucchieri che vengono raggruppati nella classe HairStylists o nel caso delle prenotazioni e le classi per raggruppare le prenotazioni (StylistsBookings e ClientBookings).
- **Façade**: La classe AuthenticationService e la classe DatabaseService sono delle façade in quanto hanno il compito di offrire tutta una serie funzionalità (per il login e per gestione del database rispettivamente) usando i metodi di un'unica classe.
- **Observer**: la dinamica Observer–Observable è utilizzata con il meccanismo dei provider, che offrono dei metodi per rendere alcuni classi Observable.

**HairStylists**

```
+stylists: Stylist[*]
-db: FirebaseDatabase = FirebaseDatabase.instance
+subscription: StreamSubscription
+event: DatabaseEvent

«async»-ReadStylists(): void
«async»-StreamChanges(): void
```

**«Singleton» user**

```
+email: String
+nick: String
+street: String
+database: FirebaseDatabase = FirebaseDatabase.instance
+instance: user

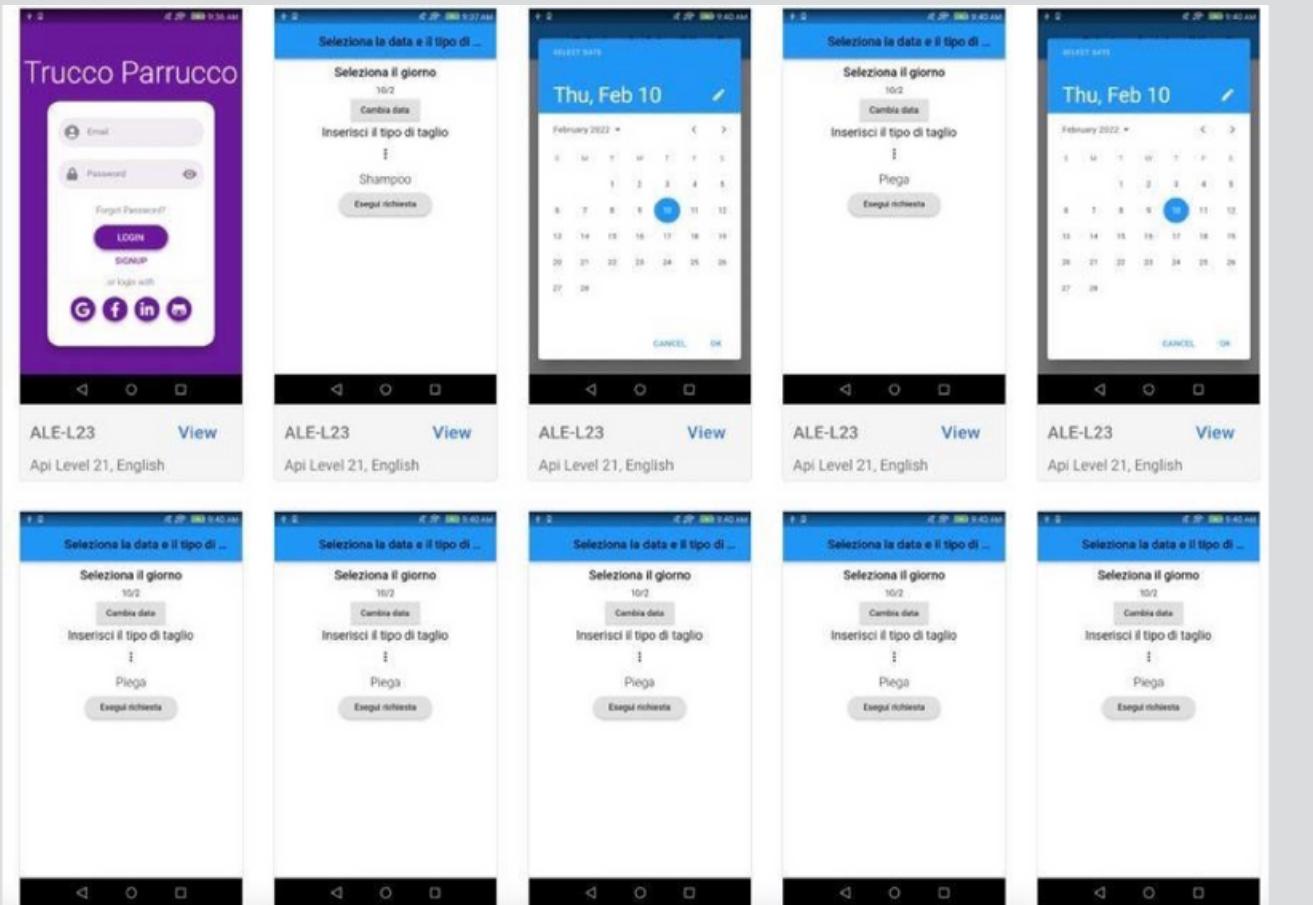
-privateConstructor(): void
«factory»+user(): user
+addUserInfo(email: String, nick: String, street: String): Future<void>
+getUserInfo(): Future<void>
```

```
user._privateConstructor() {
    this.email = "";
    this.nick = "";
    this.street = "";
    getUserInfo();
} //pattern to create a singleton
static late final user _instance =
user._privateConstructor();
factory user() => _instance; // per costruire
delle factory in Dart, basta usare la keyword
factory
```

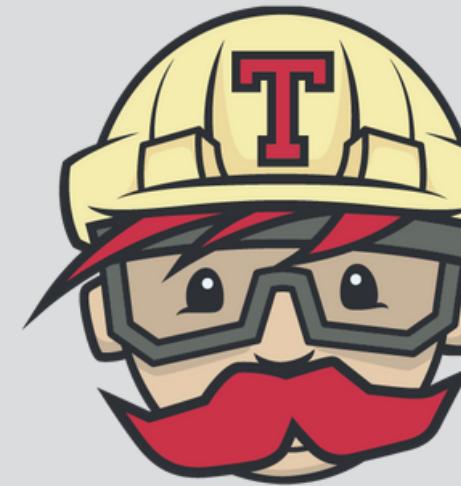
La classe User usa il design pattern Singleton che, in Dart, usa a sua volta una Factory per costruire l'istanza.

Per il testing sono stati seguiti diversi approcci per verificare il funzionamento di diverse aspetti

- **Unit testing:** test dei singoli metodi delle classi scritte in dart
- **Widget testing:** test che serve per verificare che le scritte si vedano correttamente e che i tasti siano effettivamente cliccabili
- **Continuous integration:** travis
- **Firebase testing:** Test automatico fatto da un bot simulato su diversi dispositivi mobile per vedere che l'applicazione funzioni correttamente

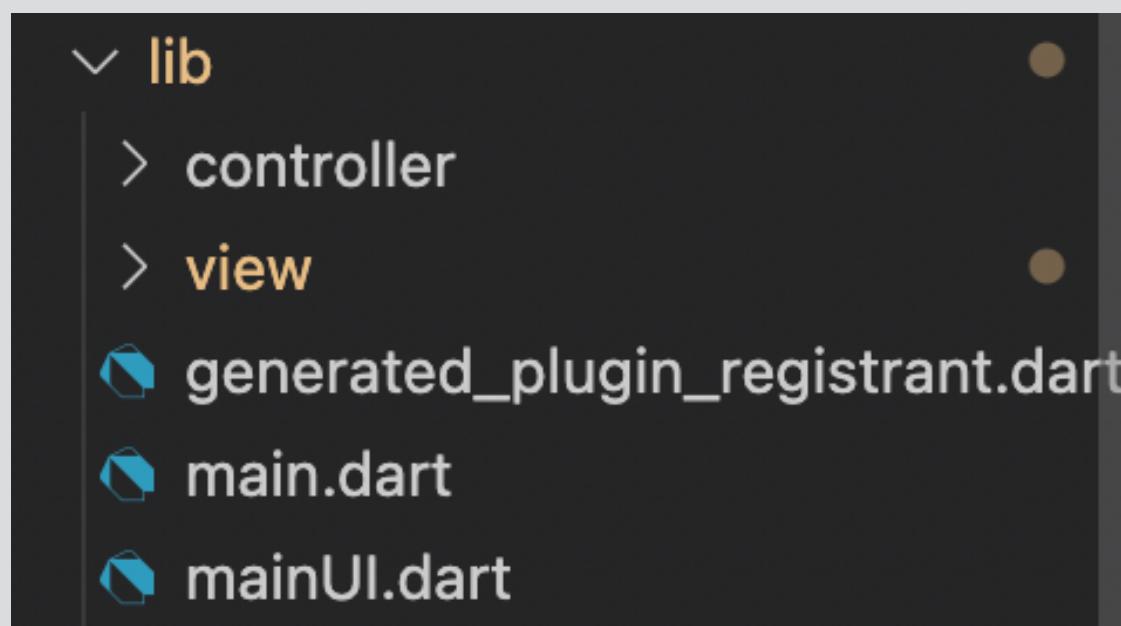


# STRATEGIE DI TESTING



# MAINTENANCE

Roundtrip engineering e refactoring



## strumenti utilizzati

- Strumenti di refactoring granulare offerti da VSCode, quali widget wrapping, creazione di funzioni, aggiornamento di import dopo spostamenti di file...
- Ristrutturazione delle cartelle per riflettere la software architecture proposta.
- Reverse engineering dei diagrammi UML, partendo dal codice che, per via di nuove scelte, differisce dalla modellazione originale.
- Forward engineering partendo dai diagrammi UML aggiornati, che a sua volta introducono nuovi elementi per migliorare la chiarezza e struttura del codice.