

**MMR Stock Manager**  
**Tecnologie Web**  
**03/06/2024**

**Francesco Mecatti**  
(Mat. 165716)

# Introduzione

## Abstract

Piattaforma per la gestione di un **magazzino** in-house di **componenti elettronici**, rivolto a organizzazioni di media grandezza (in particolare, progettato sulle esigenze di un team di **Formula Student**), con segmentazione dei **permessi** e meccanismo di **protezione** dei componenti scarsi/pregiati, esposto mediante **dashboard real-time** per approvazione e rifiuto delle richieste di utilizzo.

Mantenimento dei **log** relativi agli utilizzi e alle richieste di utilizzo dei componenti.

Presenza di un **recommendation system** di tipo user-based collaborative filtering, basato sulla preferenza binaria degli elementi (hearts); conseguente possibilità di mantenere una lista di elementi ”**preferiti**” dall’utente.

## Requisiti

Nel dettaglio, i requisiti funzionali si articolano in:

- *Core* che permetta operazioni CRUD sul DB, relativamente a componenti e profili utente.
- Possibilità di bloccare l’utilizzo di alcuni componenti, che diventano prelevabili previa richiesta ai Division Leaders.
- Sistema di notifica ai DLs delle richieste di utilizzo e notifica del relativo esito (accettazione o rifiuto) ai membri.
- Dashboard di amministrazione rivolta ai DLs per il monitoraggio dei componenti prelevati da ciascun membro fino a quel momento, in congiunzione alla gestione delle richieste di utilizzo.
- Recommendation system user-based; in quanto la similarità tra utenti è calcolabile con meno sforzo rispetto all’estrazione di features dai componenti, condizione necessaria per i sistemi di raccomandazione content-based.
- Salvataggio degli elementi ”preferiti” dall’utente, con lo scopo secondario di estrarre statistiche di preferenza, funzionali al recommendation system.
- Form di ricerca dei componenti, con filtri su quantità disponibili (eg. utile per filtrare i componenti in via di esaurimento); form di ricerca dei membri, accessibile dai DLs.

- Pagine di gestione del profilo utente: cambio password, modifica username/propic/email, login e logout.

Requisiti non funzionali:

- Diritto di accesso differenziato (Tabella 1):
  - Utente anonimo: visualizzazione di una "vetrina" di componenti, con cui non può interagire se non per la visualizzazione (riassumibile con: permessi esclusivamente gli *HTTP safe methods* sui componenti). Necessaria l'autenticazione per qualsiasi altra schermata, inclusa la visualizzazione dei propri componenti, siccome non sarebbe possibile determinare di quale utente si tratta.
  - Membro: utente base che può interagire con la lista dei componenti aggiungendoli ai propri preferiti, prelevandoli se ad accesso libero o richiedendone l'utilizzo se ad accesso protetto. I membri hanno a disposizione una *mailbox* in cui vengono notificate le richieste processate, sia con esito positivo che negativo.
  - Division Leader: superuser in grado di approvare o rifiutare le richieste dei membri, oltre ad avere i permessi in scrittura (creazione, modifica, eliminazione) sui componenti.
  - Staff user: amministratore del sito. Colui che controlla il pannello amministratore; si occupa di creare, modificare ed eliminare nuove utenze. Può non coincidere con un Division Leader (staff ma non superuser).
  - Admin user: utente con i massimi privilegi (staff e superuser). Ha l'accesso alle viste DL e alla gestione delle utenze.
- Unit testing esemplificativo sul comportamento esibito dalle viste e da parte della logica applicativa.
- Comandi di utilità per la gestione della base di dati: importazione automatizzata dei componenti già esistenti, creazione di mock users per ragioni di testing.

## Autorizzazioni

La Tabella 1 riporta in modo schematico e puntuale le autorizzazioni concesse ad ogni classe di utenti. Le *relazioni tra le entità* definite nell'intestazione della tabella, vengono definite formalmente nella Figura 1. Mentre la Tabella 2 esplicita le condizioni necessarie per appartenere ad una classe di utenti.

User	Component	Use/Request	Star	Profile
Anonymous	View	$\emptyset$	$\emptyset$	$\emptyset$
Member	View	View, Create	Toggle	View, Update of its profile
Division Leader	View, Create, Update, Delete	View, Create, Update	Toggle	View, Update of its profile
Staff	View	View, Create	Toggle	View, Create, Update, Delete of others' profile
Admin	View, Create, Update, Delete	View, Create, Update	Toggle	View, Create, Update, Delete of others' profile

Tabella 1: Autorizzazioni vs Classe di utenti

User	Condition
Anonymous	is_anonymous
Member	is_authenticated
Division Leader	is_superuser
Staff	is_staff + some permissions
Admin	is_staff + is_superuser

Tabella 2: Classi di utenti vs Condizioni di appartenenza

# Database

## Entity-Relationship Diagram

Lo schema ER mostrato in Figura 1 consta di 3 entità – *Componente*, *Profilo* e *Utente* – e 3 relazioni – *Utilizzo*, *Richiesta* e *Preferenza*.

Un *Componente* è modellato con gli attributi mostrati in figura, di cui è necessario evidenziare la tripla  $(r, c, d)$ , che rappresenta la coordinata del componente nella cassettera, in cui riga e colonna identificano un cassetto, mentre la profondità identifica in quale "slot" del cassetto si trova il componente; la condizione di unicità previene i conflitti di posizione.

Un *Profilo* non è altro che una entità che aggiunge un'immagine di profilo – *profile picture* – a *User*, la classe che modella un'utenza, fornita da Django.

La relazione (multi-a-molti) di *Uso* di un componente, modella l'evento di prelievo di un certo numero *Quantity* di componenti, in una certa data. Lo stesso vale per la relazione (multi-a-molti) di *Richiesta*, alla quale sono tuttavia aggiunti due valori booleani:

- *Approved* un valore 3-state {NULL, True, False} che memorizza se la richiesta è, rispettivamente, nello stato *pending*, *approved* o *rejected*.
- *Viewed* un flag booleano (infatti, questo attributo non ammette NULL) che è marcato a vero quando la notifica relativa alla richiesta viene visualizzata dall'utente (apertura della pagina *mailbox*).

## Dettagli

Il DMBS scelto è SQLite, principalmente per ragioni di convenienza. Inoltre, è stato scelto in quanto non sono state riscontrate severe limitazioni allo sviluppo del progetto. La sola query per cui ha rivelato carenze, è la query di full-text search sul nome dei componenti; infatti, a differenza, ad esempio, del più completo PostgreSQL, non supporta nativamente operazioni di full-text search e creazione di term vectors. Questa limitazione è stata tuttavia agevolmente aggirata mediante la costruzione di una query che metta in AND ogni termine cercato nella searchbar della pagina Home:

```
query = self.request.GET.get('query', '')
splitted_query = query.split()
conditions_gen = (Q(name__icontains=word) for word in splitted_query)
ft_query = reduce(operator.and_, conditions_gen, Q())
queryset = queryset.filter(code__icontains=query) | queryset.filter(ft_query)
```

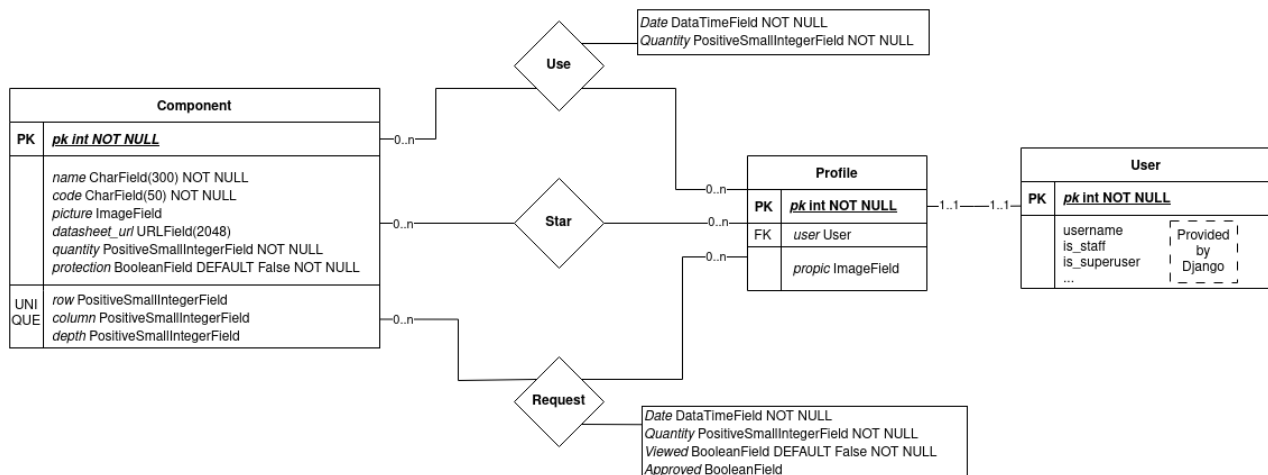


Figura 1: Entity-Relationship Diagram

Lo snippet sopra riportato esegue l'operazione appena descritta; lancia poi due query distinte – ricerca dei termini nel campo *code* e ricerca full-text sul campo *name* – per poi "mergiarne" i risultati.

## Implementazione

L'implementazione dello schema di Figura 1 differisce parzialmente da esso. La principale differenza risiede nel modo in cui sono state gestite le relazioni molti-a-molti.

Sia *Use* che *Request* sono state implementate definendo una entità che contenga tutti gli attributi presenti nel diagramma ER, con due campi aggiuntivi di tipo *ForeignKey* che referenziano, rispettivamente, utente e componente. Queste due classi sono quindi state "linkate" alla classi padre mediante l'attributo *through* del campo *ManyToManyField*:

```
class Request(models.Model):
    ...
    profile = models.ForeignKey(Profile, on_delete=models.CASCADE)
    component = models.ForeignKey(Component, on_delete=models.CASCADE)
    date = models.DateTimeField()
    quantity = models.PositiveSmallIntegerField()
    approved = models.BooleanField(null=True, blank=True)
    viewed = models.BooleanField(default=False)

class Profile(models.Model):
    ...
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

```
propic = models.ImageField(...)
stars = models.ManyToManyField(
    Component, blank=True,
    related_name='stars'
)
requests = models.ManyToManyField(
    Component, through='Request',
    blank=True, related_name='requests'
)
uses = models.ManyToManyField(
    Component, through='Use',
    blank=True, related_name='uses'
)
...
```

Nella classe *Component* non è invece presente nessuna traccia di questa relazione; la gestione viene automaticamente gestita dall'ORM di Django.

Infine, come si può notare, la relazione *stars* richiede una configurazione molto più semplice. Questo perchè non sono presenti attributi nella relazione.

# Progetto

## Tecnologie

### Frontend

Le tecnologie utilizzate per lo sviluppo del frontend comprendono i classici JS, HTML e CSS, integrati dai componenti offerti da Bootstrap 5.3 e la libreria di utilità jQuery, ampiamente utilizzata per navigare il DOM o cercare nodi al suo interno. I due controller JS – *home\_controller.js* e *dash\_controller.js* – sono stati scritti entrambi a partire da principi primi in Javascript. Essi gestiscono la logica di business delle pagine riconducibili a *Home* – ie. *Home* stessa e *Favorites* – e *Dashboard* – ie. *Dashboard* stessa e *DashboardDetail*.

Il controller della *Home* si occupa di:

- Inviare richieste HTTP, con AJAX, alle API RESTful per la gestione dei componenti: quando viene cliccata la card di un componente, viene inviata una richiesta GET per fetchare il componente, quando si richiede un componente si invia una richiesta PATCH all'endpoint interessato e, analogamente, per eliminazione, modifica e creazione.
- Inviare richieste AJAX al RESTful endpoint per il toggling dei preferiti.
- Infine, si premura di propagare le modifiche effettuate ad un componente a tutte le card dello stesso componente presenti nella pagina, per ragioni di coerenza della UI.

Mentre il controller della *Dashboard* si occupa di:

- Instaurare una connessione WebSocket e registrare una funzione di callback, invocata all'arrivo di nuovi dati.
- Aggiornare lo stato dei componenti – card delle richieste – nella pagina all'approvazione/rifiuto di una richiesta, sia da parte propria che di un utente terzo. Infatti, anche se l'approvazione dovesse venire da un utente terzo, l'aggiornamento dello stato della card viene notificato mediante WS, quindi aggiornato dal controller.
- Aggiungere i componenti appena approvati alla colonna del log di utilizzo.

### Backend

In maniera complementare, il backend espone 3 tipi di endpoint:

- *Django Views* Viste elaborate server-side tramite Django Template Engine.
- *RESTful APIs* API la cui interazione avviene per mezzo di oggetti JSON.



- *WebSockets* Un unico endpoint WS è utilizzato per ricevere aggiornamenti sullo stato delle richieste nella dashboard.

Lato backend le tecnologie utilizzate sono Django Framework, Daphne ASGI Server e Django in-memory channels. Di quest'ultimo sono state usate sia la funzionalità di raggruppamento dei canali, che come meccanismo di IPC tra più istanze del server.

Questa caratteristica di Django channels, è utilizzata per propagare l'informazione di aggiornamento di una richiesta, inviata all'endpoint REST `/requests/ID/`. In particolare, dalla logica dell'endpoint RESTful, viene recuperato il channel-layer, quindi notificato un evento di tipo *request\_XXXX* (con XXXX uguale a *add*, *approve*, *reject*), sul gruppo relativo all'utente che ha effettuato la richiesta, con payload pari al numero di richiesta appena modificata. Su questi 3 tipi di evento sono in ascolto tutti i *Consumer WS*, che si differenziano però dal **gruppo/i** sul quale ascoltano; ad esempio, se la dashboard è aperta sulla pagina dell'utente 1, il rispettivo *Consumer* resterà in ascolto degli eventi *request\_add*, *request\_approve* e *request\_reject* nel gruppo *user\_1*. Alla ricezione di un evento, dopo le opportune trasformazioni, lo inoltrerà all'altro capo del socket. Il client riceverà quindi l'evento e agirà di conseguenza.

## Organizzazione del Codice

Il progetto presentato prende il nome di *mmr\_stock* ed è suddiviso in 3 applicazioni Django:

- *core* contiene la definizione di tutti i modelli e implementa le viste e i template delle schermate: *Home*, *Favorites*, *ProfileDetail*, *PasswordChange*, *Login* e *Logout*.
- *dash* gestisce la dashboard, sia dalla prospettiva del membro che del division leader; sotto la sua gestione troviamo *Mailbox*, *Dashboard* e *DashboardDetail*
- *analytics* si occupa di esporre l'API per il toggling dei preferiti e di implementare il recommendation system.

Nella top-level directory *common\_templates* si possono trovare i due template "scheletro" sui quali sono stati costruiti i restanti template del progetto: in *base.html* si trovano le inclusioni di risorse JS e CSS esterne, l'inclusione della favicon e la definizione della navbar; mentre *base\_card.html* estende *base.html* aggiungendo una flat card al centro dell'area dei contenuti.

## Data Flow

La quasi totalità delle interazioni tra client e server avviene mediante richiesta-risposta HTTP, l'unico caso che si distingue è l'interazione della dashboard con il backend, illustrata in figura 2 e descritta poco sopra.

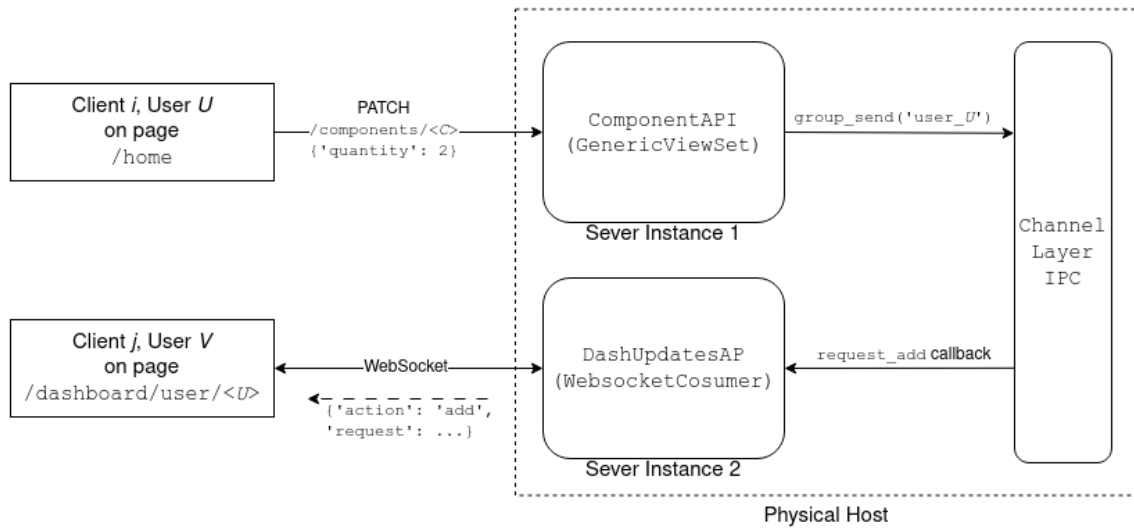


Figura 2: Dashboard Update Data Flow

Per riassumere, sia che il client  $i$  sia un utente  $U$  che sta interagendo con la propria schermata home, sia che esso sia un division leader (un terzo utente  $W$  non rappresentato in figura, concettualmente sostituibile a  $U$ ) che sta interagendo con la dashboard dell'utente  $U$ , l'aggiornamento e la creazione delle richieste di utilizzo avvengono mediante endpoint REST. Come si può notare, nel secondo caso – sia  $i$  che  $j$  connessi alla dashboard – entrambi i client avrebbero avuto a disposizione i WS per comunicare al server l'aggiornamento di stato, ma non vengono usati nella direzione client to server, per mantenere omogeneità architetturale. L'aggiornamento deve essere inviato per mezzo di una richiesta PUT a `/requests/ID`; sarà il backend ad occuparsi della propagazione dell'evento alle altre istanze del server in ascolto, mediante il meccanismo di IPC offerto da Django Channels.

## Architettura

Tra le decisioni architetturali degne di nota possiamo trovare il sopra menzionato meccanismo di creazione di una richiesta di utilizzo, che fa uso del metodo PATCH (partial update) **sul componente** di cui si sta facendo richiesta, per creare la relazione tra l'utente loggato e il componente.

Inoltre, è degno di nota anche il meccanismo di starring degli elementi. Infatti, per evitare al client il mantenimento dello stato dei preferiti, al click di un cuore corrisponde non un set/unset del preferito, bensì un toggle effettuato con richiesta POST all'unico endpoint di gestione dei preferiti; nella risposta si trova quindi lo stato del preferito dopo il toggle, così che possa essere impostato graficamente. Questo meccanismo evita spiacevoli situazioni di inconsistenza, generate da due tab aperte sulla schermata home dello stesso utente (e non aggiornate in

quanto home non fa uso di WS). Lo scenario peggiore è quello in cui una pagina, che mantiene il vecchio stato di un cuore, viene usata dall'utente per togglare il preferito; il comportamento esibito con questo meccanismo è semplicemente una non-risposta alla prima azione dell'utente che, determinato nel suo scopo, riproverà a togglare il preferito, riuscendoci al secondo tentativo.

## Unit Testing

Le funzionalità coperte da **unit testing** sono: l'endpoint per il **toggling dei preferiti** (*/favorites/ID*) e il metodo che racchiude la logica per determinare se mostrare il **badge delle notifiche** all'utente (*core.models.Profile.has\_notification()*). I test si trovano, rispettivamente, in *analytics/tests.py* e *core/tests.py*.

# Recommendation System

## Assunzioni

Il recommendation system proposto si regge su diverse assunzioni, tra le quali troviamo:

- Difficoltà nell'estrarre le features degli elementi, contrapposta alla facilità di definire una metrica di distanza tra gli utenti: distanza euclidea in uno spazio vettoriale N-dimensionale, con N numero di elementi.
- Presenza di una informazione (relazione *Star*) sulla preferenza di un elemento da parte di un utente.
- Ipotesi che due utenti sono simili se condividono un alto numero di preferiti.
- Ipotesi che due utenti simili condividono gli stessi "interessi" rispetto ai componenti

Basandomi su queste assunzioni, ho scelto un approccio di tipo **user-based collaborative filtering** che fa uso del **vettore dei preferiti** per caratterizzare un utente. La vicinanza tra due utenti può essere quindi calcolata come **distanza euclidea n-dimensionale**.

## Implementazione

Il recommender è stato implementato come:

- Estrazione dei vettori binari di preferenze da ogni utente.
- Una ricerca K-Nearest-Neighbors, per ricavare l'intorno di utenti simili, con K parametrizzato in *analytics/recommendation.py*.
- Media semplice del rating di ogni elemento ( $r_i = \frac{\sum_{u=0}^M star_{u,i}}{M} \forall i \in [0, N]$ ). Possono essere usate strategie più raffinate come media pesata sulla correlazione/similarità, ma questo approccio si è rivelato sufficientemente efficace per piccole istanze di test.
- Filtraggio degli elementi: thresholding  $r_i > T$  + slicing (N-top items by ranking). Entrambi questi parametri sono stati parametrizzati nello stesso modulo di cui sopra e, per istanze di test, assumono il valore di 0.5 e 10.

# Interfaccia

## Wireframe

Le pagine del sito sono state pensate per mantenere una **navbar** sempre visibile, nella porzione **superiore** dello schermo. La navbar è a sua volta organizzata in **3 sezioni**, da sinistra verso destra:

- **Titolo** del progetto, cliccabile, rimanda alla schermata home
- Nome della **pagina corrente**
- Pulsanti di **navigazione**. In ordine: **home**, **preferiti**, **mailbox**, **dashboard**, **profilo utente**

Sottostante alla navbar si trova l'area dei **contenuti**. Quest'area ha un'altezza fissa e, per tanto, non sarà scrollabile; infatti, in caso di lunghe liste di elementi, saranno le **card** ad occuparsi a gestire l'**overflow** e la "**scrollabilità**". Nella pagina home, ho prestato particolare attenzione alla **paginazione** degli elementi; la paginazione è gestita dal paginatore di Django, che ritorna 12 elementi per pagina, generati in modo **lazy**.






## Design System

Sebbene non sia stato utilizzato un design-system "preconfezionato", i componenti grafici e le interazioni con l'utente sono ispirati al design system di Atlassian, che utilizza estensivamente il componente della *card* per organizzare il contenuto su schermo e per racchiudere informazioni di una stessa entità.

## Risultato

MMR Stock Manager

Home



francesco

New +

200 results found | Page 1

MOSFET  
STMicroelectronics,  
canale N, 5 mΩ, 120 A,  
TO-220, Su ...

Code: FQP33N10  
Quantity: 22

Condensatore  
ceramico multistrato  
MLCC, 0805 (2012M),  
47nF, ...

Code: X7R 50V 47nF  
Quantity: 50

MOSFET  
STMicroelectronics,  
canale N, 18 mΩ, 50 A,  
D2PAK (TO- ...

Code: STB55NF06T4  
Quantity: 2

Comparatore  
analogico

Code: LM393ADR  
Quantity: 0

Comparatore Texas  
Instruments, Su foro  
alimentazione singola  
...

Code: LM239N  
Quantity: 24

Contatore CD4017BM,  
stadi 5, 4000, SOIC,  
16-Pin 1

Code: CD4017BM  
Quantity: 0

Regolatore di tensione  
L7805CDT-TR, 1.5A,  
3-Pin, DPAK

Code: L7805CDT-TR  
Quantity: 10

Gate logico Quad OR  
Texas Instruments, 3 V  
→ 18 V, 14 Pin, P ...

Code: CD4071BE  
Quantity: 49

Resistore Vishay, 5mΩ  
±1% 3W, Serie LVR

Code:  
LVR035L000FE12  
Quantity: 12

MOSFET Infineon,  
canale P, 110 MO, 18 A,  
DPAK (TO-252), SMD

Code: IFR5505TRPBF  
Quantity: 40

U may want to start from ↗

Cold start

Figura 3: Home

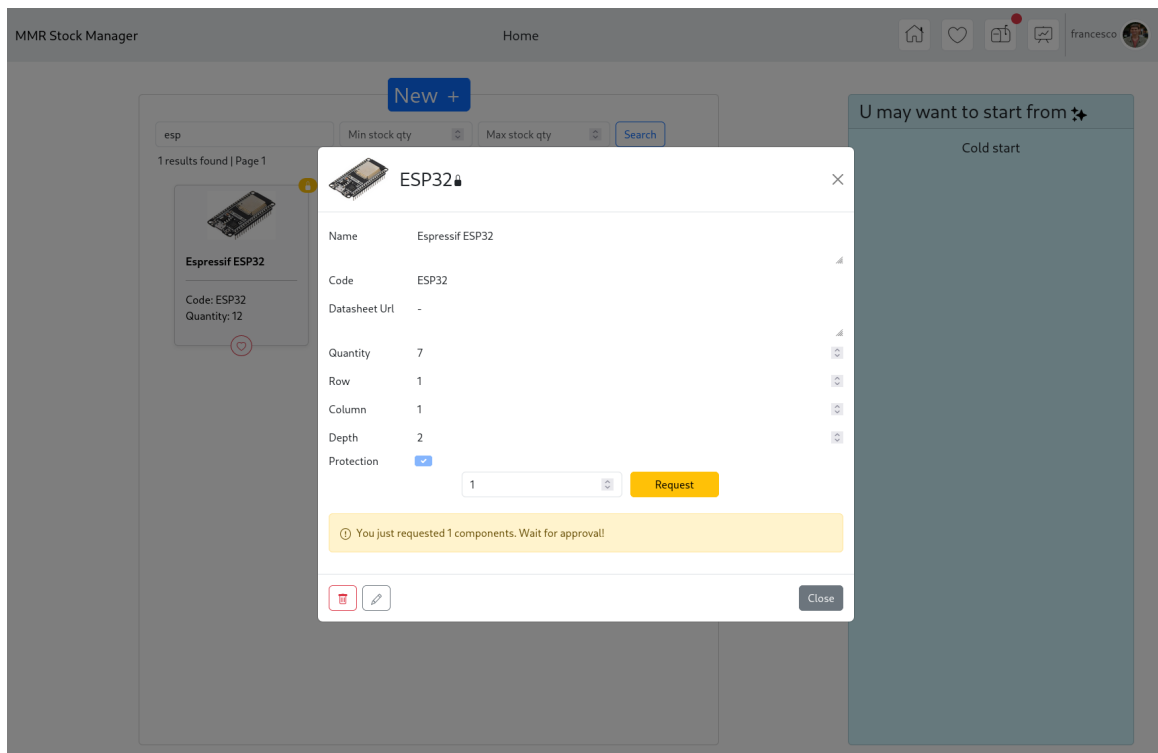


Figura 4: Component

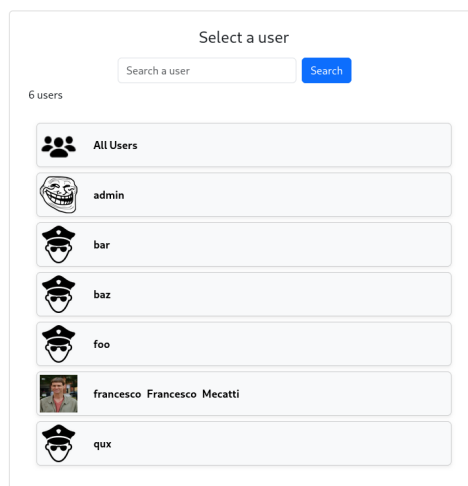
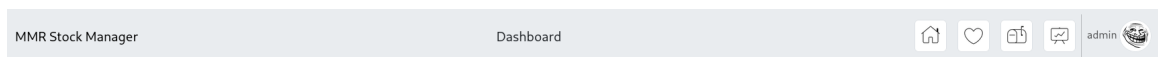


Figura 5: Dashboard

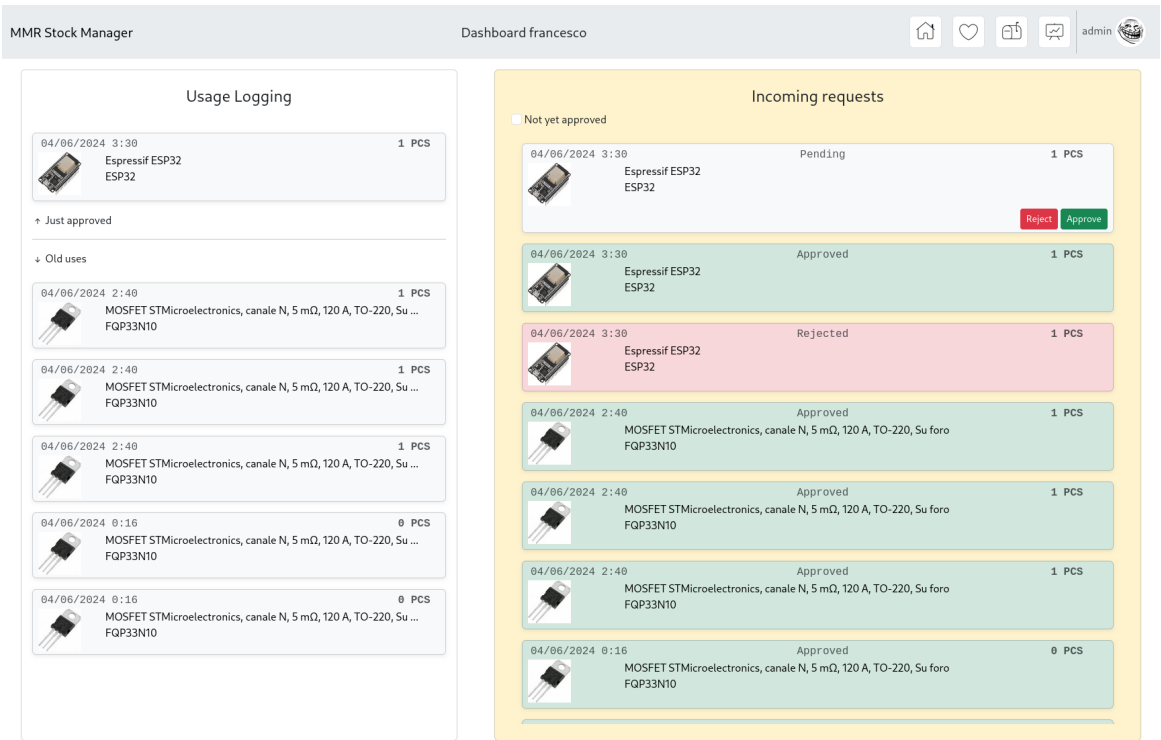


Figura 6: Dashboard User "Francesco"