

OUT OF BOUNDS

SPECIFICA TECNICA
PROGETTO CAPTCHA
Versione 1.0.0

Responsabile	Michele Cazzaro
Redattori	Alberto Matterazzo Valentina Caputo Edoardo Retis Jacopo Angeli Michele Cazzaro
Verificatori	Michele Cazzaro Valentina Caputo Simone Bisortole Alberto Matterazzo
Uso	Esterno
Destinatari	Out of Bounds prof. Tullio Vardanega Prof. Riccardo Cardin Zucchetti S.p.A.

CONTATTI
sweoutofbounds@gmail.com
REPOSITORIES
[orgs/SWE-OutOfBounds/repositories](https://github.com/SWE-OutOfBounds/repositories)

Registro delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
1.0.0	2023/05/28	Michele Cazzaro	Responsabile	Approvazione e rilascio del documento.
0.1.0	2023/05/27	Alberto Matterazzo	Verificatore	Revisione complessiva del documento.
0.0.9	2023/05/19	Jacopo Angeli, Alberto Matterazzo	Progettista, Verificatore	Modifiche alla struttura del documento e alle sezioni §2 e §3 e verifica.
0.0.8	2023/05/09	Michele Cazzaro, Simone Bisortole	Progettista, Verificatore	Stesura §3.3.2 e verifica.
0.0.7	2023/05/06	Michele Cazzaro, Simone Bisortole	Progettista, Verificatore	Incremento §3, inizio stesura §4 e verifica.
0.0.6	2023/05/02	Alberto Matterazzo, Valentina Caputo	Progettista, Verificatore	Fix diagrammi e verifica.
0.0.5	2023/04/22	Jacopo Angeli, Valentina Caputo	Progettista, Verificatore	Ampliamento scheletro, inizio stesura §3, incremento §2 e verifica.
0.0.4	2023/04/05	Edoardo Retis, Michele Cazzaro	Progettista, Verificatore	Incremento sezione §2 e verifica.
0.0.3	2023/04/02	Valentina Caputo, Michele Cazzaro	Progettista, Verificatore	Incremento sezione §2 e verifica.
0.0.2	2023/03/30	Alberto Matterazzo, Michele Cazzaro	Amministratore, Verificatore	Stesura §1 e verifica.
0.0.1	2023/03/29	Alberto Matterazzo	Amministratore	Creazione Scheletro del documento.



Tabella 1: Registro delle modifiche

Indice

1	Introduzione	5
1.1	Scopo del documento	5
1.2	Scopo del capitolato	5
1.3	Glossario	5
1.4	Riferimenti	5
1.4.1	Normativi	5
1.4.2	Informativi	5
2	Tecnologie utilizzate	7
3	Architettura dei prodotti	10
3.1	Architettura generale	10
3.1.1	Divisione logica dei prodotti	10
3.1.2	Interazioni tra i componenti	11
3.2	Architettura di dettaglio	13
3.2.1	Web App	13
3.2.1.1	Applicazione di back-end	13
3.2.1.1.1	Modello architetturale	13
3.2.1.1.2	Application Program Interface	13
3.2.1.2	Applicazione di front-end	13
3.2.1.2.3	Modello architetturale	14
3.2.1.2.4	Diagramma delle classi	14
3.2.1.2.5	Design pattern in Angular	16
3.2.2	clock-CAPTCHA	18
3.2.2.1	Presentation tier	18
3.2.2.1.1	Diagramma delle classi	18
3.2.2.1.2	Interfaccia della classe	19
3.2.2.2	Application Tier	20
3.2.2.2.3	Diagramma delle classi	21
3.2.2.3	Design patterns utilizzati	21
3.2.2.3.4	Dependency injection pattern	21
3.2.2.3.5	Strategy pattern	22
3.2.2.3.6	Decorator pattern	22
3.2.3	Database	24
3.2.3.1	Diagramma ER	24
3.2.3.2	Diagramma UML	25

Elenco delle figure

1	Visione d'insieme delle relazioni tra i vari componenti	10
2	<i>Three tier architecture</i> applicata ai prodotti sviluppati.	11
3	Diagramma delle classi dell'applicazione di front-end.	15
4	Modulo HTML rappresentato da ClockCAPTCHAView	18
5	Diagramma della classe ClockCAPTCHAView.	19
6	Diagramma delle classi della parte back-end della libreria clockCAPTCHA.	21

7	Diagramma ER del database utilizzato dall'applicazione.	24
8	Diagramma UML del database utilizzato dall'applicazione.	25

Elenco delle tabelle

1	Registro delle modifiche	2
2	Tabella delle tecnologie utilizzate	9
3	Interfaccia fornita dal back-end.	13
4	Interfaccia della classe clockCAPTCHAView.	20

1 Introduzione

1.1 Scopo del documento

Lo scopo di questo documento è quello di descrivere e motivare le scelte architettureali che il gruppo *Out Of Bounds* ha deciso di prendere nella fase di progettazione e codifica del prodotto. In questo documento vengono riportati i diagrammi delle classi per descrivere l'architettura e le funzionalità del prodotto, insieme alle tecnologie che il gruppo ha scelto di adottare per la realizzazione del progetto.

1.2 Scopo del capitolato

La richiesta dell'azienda Zucchetti è un servizio *web CAPTCHA_G*, ovvero un prodotto che discrimini esseri umani da *bot_G* artificiali. Il tipo di tecnologia e la modalità di sviluppo non sono soggetti ad alcun vincolo.

1.3 Glossario

I termini che possono generare dubbi riguardo al loro significato vengono contrassegnati con una lettera G al pedice, a indicare che il termine si può trovare nel documento *Glossario v2.0*.

1.4 Riferimenti

1.4.1 Normativi

- [Regolamento del progetto didattico](#) [Ultima consultazione: 2023/05/27],
- [Documentazione Capitolato presentato dall'azienda](#) [Ultima consultazione: 2023/05/27],
- [Norme di progetto v2.0](#) [Ultima consultazione: 2023/05/27],
- [Analisi dei requisiti v3.0](#) [Ultima consultazione: 2023/05/27].

1.4.2 Informativi

- [Progettazione e programmazione: Diagrammi delle classi \(UML\) — Slide fornite del corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],
- [Progettazione: I pattern architettureali — Slide fornite dal corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],
- [Progettazione: Il pattern Dependency Injection — Slide fornite dal corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],
- [Progettazione: Il pattern Model-View-Controller e derivati — Slide fornite dal corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],
- [Progettazione: I pattern crezionali — Slide fornite dal corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],
- [Progettazione: I pattern strutturali — Slide fornite dal corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],



- [Progettazione: I pattern di comportamento — Slide fornite dal corso "Ingegneria del Software" \(A.A. 2022/2023\)](#) [Ultima consultazione: 2023/05/27],
- [Guides — Node.js](#) [Ultima consultazione: 2023/05/27],
- [Representational state transfer — Wikipedia](#) [Ultima consultazione: 2023/05/27],
- [Angular developer guides — Angular](#) [Ultima consultazione: 2023/05/27].

2 Tecnologie utilizzate

Di seguito sono riportate le tecnologie scelte per lo sviluppo rispettivamente di *Web App_G* e servizio *CAPTCHA_G* del progetto sviluppato dal gruppo *Out of Bounds*.

Tecnologia	Descrizione	Versione
Linguaggi		
Javascript	Utilizzato per l'implementazione dei servizi lato <i>server</i> per la comunicazione tra <i>Web App_G</i> e <i>database</i> .	V8 10.7
HTML	Utilizzato assieme ad <i>Angular</i> per l'implementazione della struttura statica delle pagine <i>web</i> della <i>Web App_G</i> .	5
CSS	Utilizzato per definire la formattazione dei documenti <i>HTML</i> e lo stile.	3
Typescript	Utilizzato assieme ad <i>Angular</i> per definire il comportamento della <i>Web App_G</i> .	4.8.2
YAML	Utilizzato per creare dei file di configurazione per <i>Swagger</i> .	0.3
JSON	Utilizzato per rappresentare i dati scambiati tra <i>client</i> e <i>server</i> .	Standard RFC 4627
Strumenti		
MySQL	Utilizzato per lo stoccaggio e l'archiviazione dei <i>token</i> attivi o deprecati utilizzati dal servizio <i>CAPTCHA_G</i> , e, più in generale, per la creazione e la gestione del <i>database</i> della <i>Web App_G</i> .	2.18.1
NodeJS	Ambiente di esecuzione che permette di eseguire codice <i>Javascript</i> come un qualsiasi linguaggio di programmazione usato per supportare il lato <i>server</i> del prodotto	19.0.0
npm	Gestore di pacchetti predefinito di NodeJS, utilizzato per l'installazione e per la gestione delle dipendenze tra i <i>packages</i> utilizzati nell'implementazione del prodotto.	8.19.2
Librerie e Framework _G		

Angular	Framework utilizzato per la creazione di <i>Web App_G</i> dinamiche per dispositivi <i>desktop</i> e <i>mobile</i> .	15.0.0
Angular Material	Libreria di componenti Angular utilizzata per sviluppare la <i>Web App_G</i> lato <i>client</i> .	15.0.0
RxJS	<i>Reactive Extensions Library for JavaScript</i> , è la libreria utilizzata in <i>Angular</i> per la gestione degli eventi asincroni, nel progetto è stata utilizzata per gestire la comunicazione tra servizi e componenti e per la gestione dei dati da visualizzare in funzione dello stato di sessione.	7.5.0
Express	<i>Framework</i> per applicazioni <i>web</i> utilizzato assieme a <i>NodeJS</i> per sviluppare l'applicazione <i>Web</i> lato <i>server</i> .	4.18.2
Bcrypt	Libreria per <i>NodeJS</i> utilizzata per l' <i>hashing</i> delle <i>password</i> .	5.1.0
Crypto-js	Libreria <i>Javascript</i> utilizzata per criptare il <i>token</i> del <i>clock-CAPTCHA_G</i> .	4.1.1
JsonWebToken	<i>Token</i> di accesso standardizzato utilizzato per consentire lo scambio sicuro di dati tra la parte <i>client</i> e la parte <i>server</i> .	9.0.0
node-canvas	Canvas utilizzato con <i>NodeJS</i> per permettere alla libreria <i>CAPTCHA_G</i> di disegnare le figure geometriche necessarie per generare l'immagine dell'orologio.	2.11.2
Analisi statica		
<i>Prettier</i>	Estensione di <i>Visual Studio Code_G</i> per la formattazione automatica del codice. Utilizzato per delegare l'azione di formattare il codice ed avere quindi una struttura dei <i>file</i> omogenea.	9.13.0
Analisi dinamica		
<i>Jest</i>	<i>Jest</i> è un <i>framework</i> di <i>testing</i> utilizzato per verificare il funzionamento del codice <i>Javascript</i>	29.5.0

<i>SuperTest</i>	<i>SuperTest</i> è una libreria di test dei nodi per le chiamate <i>HTTP</i> . Estende la libreria di test dei superagenti e consente di effettuare richieste come <i>GET</i> , <i>POST</i> , <i>PUT</i> e <i>DELETE</i>	6.3.3
<i>Jasmine</i>	<i>Jasmine</i> è un <i>framework</i> di <i>testing</i> per <i>Javascript</i> , utilizzato per la produzione del codice di test, come indicato nelle linee guida di <i>Angular</i>	4.5.0
<i>Cypress</i>	<i>Cypress</i> è un <i>framework</i> di <i>testing</i> e2e <i>Javascript</i> , utilizzato per l'analisi <i>end-to-end</i> del <i>front-end</i> , come indicato nelle linee guida di <i>Angular</i>	12.12.0
Produzione della documentazione		
<i>Swagger</i>	<i>API_G</i> utilizzata per la documentazione e la descrizione delle <i>API RESTful_G</i> implementate nel progetto.	2.23.1
<i>CompoDoc</i>	<i>Tool</i> utilizzato per generare in maniera automatica la documentazione dell'applicazione implementata con <i>Angular</i>	1.1.19

Tabella 2: Tabella delle tecnologie utilizzate

3 Architettura dei prodotti

Il proponente richiede la consegna di due prodotti distinti: un servizio di $CAPTCHA_G$ e un'applicazione web che ne dimostri il funzionamento. Questa sezione ha lo scopo quindi di presentare tutti i prodotti creati, utilizzando una metodologia $Top-Down_G$.

3.1 Architettura generale

Il servizio di $CAPTCHA_G$ è stato sviluppato sotto forma di libreria *javascript*. La libreria si divide in due parti:

- **Front-end:** fornisce un oggetto, contenente degli elementi *HTML*, che consente di mostrare a schermo i dati generati dalla parte *back-end* e di raccogliere l'input fornito dall'utente,
- **Back-end:** fornisce due funzioni:
 - **Generazione dei dati:** vengono generati dei dati criptati e la loro rappresentazione grafica,
 - **Verifica dei dati:** valida l'input utente in relazione ad un set di dati precedentemente generato.

È stata sviluppata in seguito un'applicazione che utilizza appieno le funzionalità proposte dalla libreria, utilizzando *Angular* per la parte *front-end*, *Node.js* con *framework Express* per la parte *back-end*.

Quest'ultimo fornisce un'interfaccia di $APIs_G$ di tipo *RESTful_G* per la completa gestione dei dati contenuti nel *database*.

Il salvataggio e la gestione dei dati in esso viene infine gestita con *MySQL*.

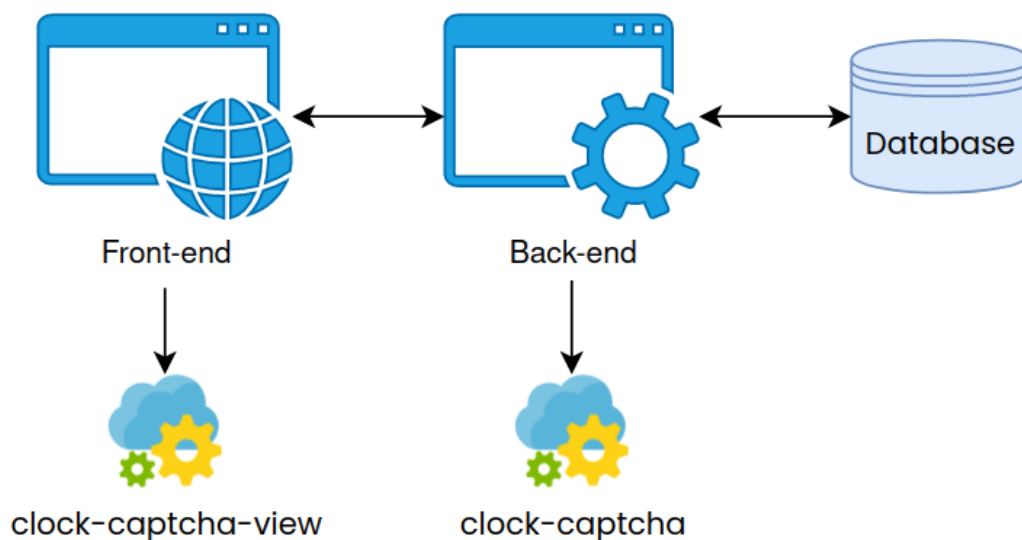


Figura 1: Visione d'insieme delle relazioni tra i vari componenti

3.1.1 Divisione logica dei prodotti

È stata rispettata durante la progettazione la *three-tier architecture_G*:

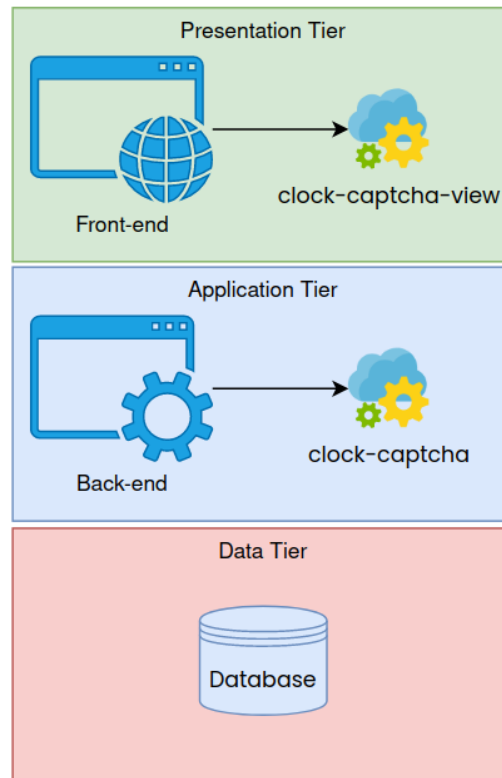


Figura 2: *Three tier architecture* applicata ai prodotti sviluppati.

- **Presentation tier:** costituito dal *front-end* della *Web App_G*, rappresenta la parte visibile all'utente occupandosi anche della raccolta dei dati e del loro adattamento al *back-end*,
- **Logic tier:** costituito dalla parte *back-end* dell'applicazione, e si occupa dell'elaborazione completa dei dati rappresentando di fatto il comportamento effettivo dell'applicazione,
- **Data tier:** rappresenta la parte di dati del progetto, viene gestito mediante *MySQL*.

3.1.2 Interazioni tra i componenti

Utilizzo di *clock-captcha-view*

Ogni volta che un utente si collega al *link* della pagina di accesso o di registrazione, il corrispondente componente *Angular* viene creato. Durante il processo di creazione il *front-end* istanzia un oggetto di tipo *clock-captcha-view*, lo inserisce all'interno della vista del componente e in seguito richiede al *back-end* i dati da inserire all'interno dell'oggetto *clock-captcha-view*. Prima di ricevere la risposta, il modulo rappresenta un'animazione di caricamento comunicando all'utente lo stato del processo.

Quando l'applicazione deve effettuare la registrazione o l'accesso e necessita di inviare i dati può semplicemente recuperarli utilizzando l'interfaccia fornita dal modulo.

Utilizzo di *clock-captcha*

Il *back-end* fornisce un *end-point* dedicato per la richiesta di generazione dei dati che genera utilizzando interfaccia e oggetti forniti dalla libreria *clock-captcha* e una *password* privata.

Per quanto riguarda la validazione dei dati invece, il *back-end* utilizza una funzione



dedicata che controlla il contenuto della richiesta, e valida i dati del *CAPTCHA* in essa contenuti ritornando un messaggio di errore se necessario o proseguendo con la funzionalità principale altrimenti.

Registrazione

Creazione di un *account* per generare delle credenziali con cui è possibile in seguito effettuare l'accesso nella *web-app*.

Presentation tier

Dalla pagina *signup* l'utente può compilare la *form_G* apposita con i dati che più preferisce, l'applicazione notifica in tempo reale all'utente eventuali errori in fase di compilazione, evitando anche l'invio di dati inconsistenti. L'invio dei dati non è consentito senza la compilazione anche del modulo *clock-captcha-view*. Dopo l'invio l'utente viene informato sullo stato della richiesta e se la richiesta ha avuto esito positivo viene reindirizzato alla pagina di accesso.

Application tier

Alla ricezione della richiesta l'applicazione di *back-end* controlla nell'ordine, rispondendo con un messaggio di errore se necessario:

- Consistenza dei dati inviati,
- Validità del *CAPTCHA* utilizzando *clock-captcha*,
- Assenza dell'*email* nel *database*.

Se tutti i controlli risultano positivi, allora si applica la funzione di *hash* alla *password* e il risultato viene inserito all'interno del *database*, insieme agli altri dati ricevuti.

Autenticazione

Processo di apertura di sessione, necessaria per poter accedere a funzionalità nascoste all'utente generico.

Presentation tier

Dalla pagina *login* l'utente può tentare l'accesso al sistema con conseguente apertura di sessione. Viene fornita una *form_G* per l'inserimento dei dati con controllo di validità solo per il campo *email*. Per effettuare l'invio è richiesto anche l'inserimento di un valore all'interno del modulo *clock-captcha-view*. Il risultato dell'operazione viene notificato all'utente solo se negativo altrimenti viene reindirizzato alla pagina principale.

Una volta aperta la sessione, questa viene mantenuta attiva tramite salvataggio dei dati all'interno dei *cookies*. Questi dati saranno cancellati non appena l'utente effettuerà il *log out* dall'applicazione o allo scadere della sessione. Ogni sessione, infatti, avrà un tempo massimo di apertura, allo scadere del quale verrà invocata la funzione di *log out* implementata nel *front-end* dell'applicazione.

Application tier

Ricevuta la richiesta di accesso, l'applicazione di *back-end* controlla nell'ordine, rispondendo con messaggi di errore se necessario:

- Consistenza dei dati inviati,
- Validità del *CAPTCHA* utilizzando *clock-captcha*,
- Corrispondenza nel *database* della coppia *email*, *password*.

Se tutti i controlli risultano positivi, allora viene creato un *token* di sessione, viene firmato utilizzando *JsonWebToken* e viene incluso nella risposta.

3.2 Architettura di dettaglio

La sezione descrive a basso livello tutti i dettagli implementativi dell'Applicazione Web e della libreria sviluppate.

3.2.1 Web App

Viene fornita un'applicazione che rappresenta un *Forum* generico. Essa non contiene, infatti, alcun dato consistente, ma fornisce tutte le funzionalità richieste dal proponente.

In particolare, l'applicazione è composta da tre pagine: una di accesso, una di registrazione e una principale che ha lo scopo di dimostrare l'effettiva apertura e chiusura della sessione.

3.2.1.1 Applicazione di back-end

È la porzione che implementa l'effettivo funzionamento dell'applicazione e che gestisce i dati salvati nel *database*.

3.2.1.1.1 Modello architetturale

Vista la semplicità dell'applicazione è stato sviluppato un *back-end* basilare, per il quale non è stata utilizzata un'architettura specifica, ma al quale sono stati comunque applicati aspetti di *MVC*. Infatti, alla ricezione di una richiesta i *routers* delegano la sua gestione al controllore e, se necessario prima ai *middleware*, dedicata che elaborano i dati contenuti in essa e generano la risposta in formato *JSON*.

3.2.1.1.2 Application Program Interface

Path	Metodo HTTP	Funzione
/users	POST	Registrazione di un utente al servizio con controllo <i>CAPTCHA</i> .
/session	GET	Recupero dei dati di sessione
/session	POST	Apertura di una nuova sessione
/clock-captcha	GET	Richiesta dei dati per inizializzare un oggetto <i>clock-captcha-view</i> .

Tabella 3: Interfaccia fornita dal back-end.

3.2.1.2 Applicazione di front-end

È la porzione che rappresenta la parte visibile dell'applicazione e ha la funzione di intermediario tra utente e *back-end*. Essa, infatti, raccoglie i dati inseriti dall'utente e li rende compatibili

con l'interfaccia fornita dal *back-end*.

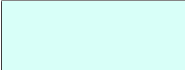


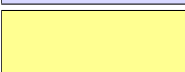
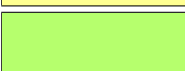

3.2.1.2.3 Modello architetturale

Utilizzando il *framework Angular* il modello architetturale adottato è quello implementato da quest'ultimo. *Angular* utilizza il *pattern MVVM (Model, View, View Model)* ed è basato sui componenti.

- **View:** corrispondente in *Angular* del *template*, rappresenta la parte visibile dell'applicazione, con la quale l'utente può direttamente interagire;
- **View Model:** l'insieme dei componenti di *Angular*, ognuno dei quali fornisce dei metodi per la manipolazione dei dati e gestisce il *binding* tra dati e vista,
- **Model:** costituito dalle classi e dai servizi di *business logic*, nel progetto vengono utilizzati per organizzare i dati e gestire l'interazione con il *back-end*

Le viste in *Angular* sono scritte in *HTML* con direttive proprie da utilizzare per effettuare il *binding* tra i dati. Il contenuto del componente è scritto invece in *typescript* ed espone dati e metodi alla vista, il suo contenuto è tipicamente il meno complesso possibile. La *business logic* più pesante è sviluppata nei servizi, i quali vengono usati per sviluppare il *core* di *Angular*, nel progetto sono usati per interagire con il *back-end*, per impostare delle regole di *routing*, o per gestire la sessione.

3.2.1.2.4 Diagramma delle classi

Legenda	
	Componenti.
	Built-in.
	Servizi.
	Configurazione.
	Libreria clockCAPTCHA.
	Modello.

Il diagramma si trova nella pagina successiva.

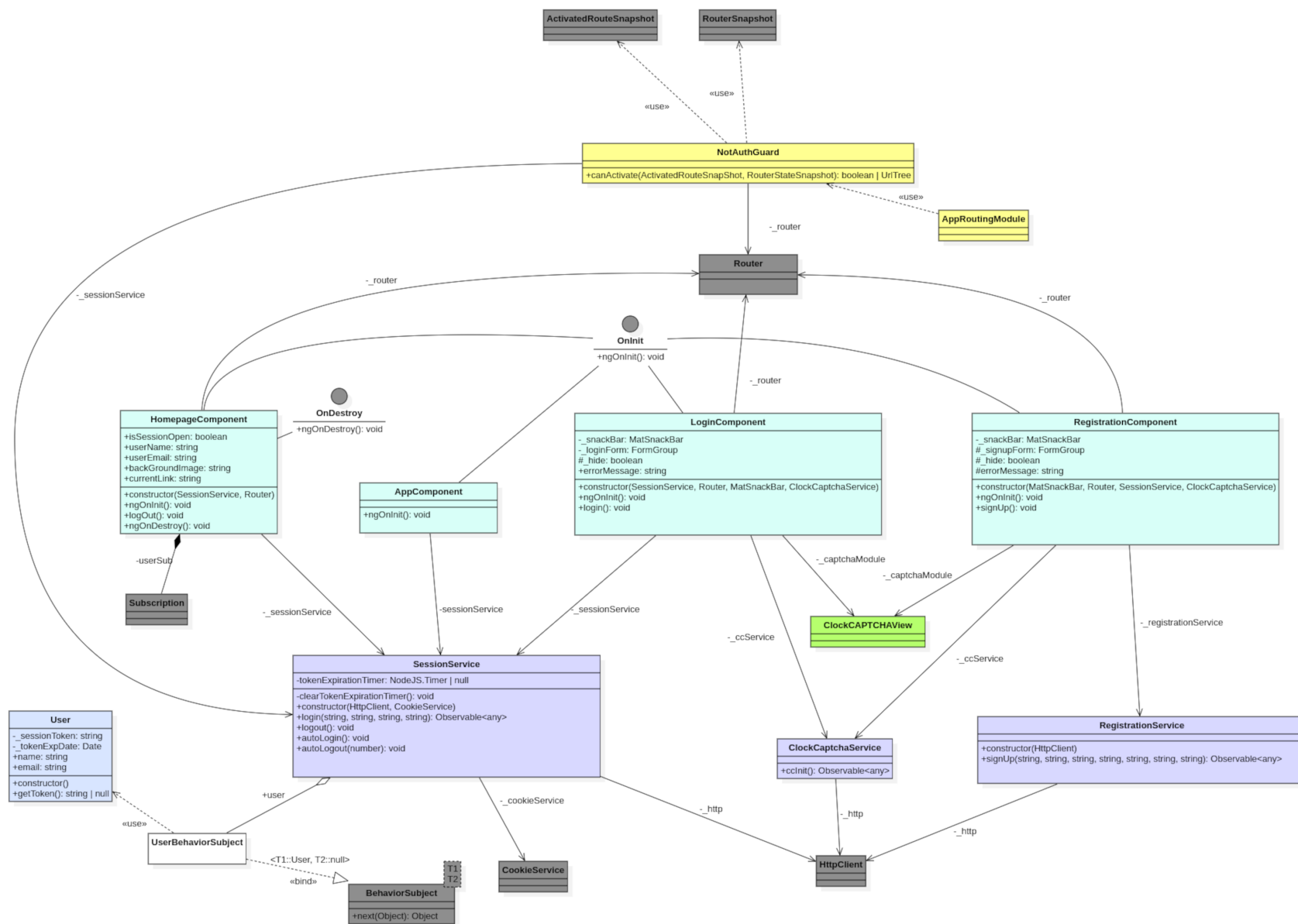


Figura 3: Diagramma delle classi dell'applicazione di front-end.

3.2.1.2.5 Design pattern in Angular

Pattern Observer

Nel progetto vengono utilizzati gli *Observable*, tra cui il *BehaviorSubject*. Questi sono implementati nella libreria RxJS, inclusa in *Angular*, utilizzando il *pattern Observer*.

Il *BehaviorSubject* è una variante del tipo *Subject*, il quale è a sua volta un tipo di *Observable*. L'oggetto definito con *BehaviorSubject* può essere considerato come un "soggetto osservabile", in cui i componenti o i servizi possono registrarsi come osservatori per ricevere gli aggiornamenti del valore corrente del soggetto. Quando un nuovo valore viene emesso, tramite chiamata alla funzione *next*, tutti gli osservatori registrati ricevono automaticamente l'aggiornamento, tramite chiamata alla funzione *subscribe*. Il motivo per cui è stato scelto di utilizzare *Behavior Subject*, anziché *Subject*, è dovuto alla possibilità, data dal primo, di memorizzare il dato emesso.

Nel progetto il *pattern* viene utilizzato tramite:

- *Observable* per gestire la comunicazione dei dati tra i servizi e i relativi componenti,
- *BehaviorSubject* per garantire all'utente la corretta visualizzazione dei dati sulla base dello stato della sessione.

Ad esempio, in quest'ultimo caso, ogni volta che il servizio *SessionService* crea o trova i dati dell'utente che ha effettuato l'accesso al sistema, emette:

```
this.user.next(loggedUser);
```

Mentre, ogni volta che la sessione termina, per chiusura volontaria o per scadenza della durata, emette:

```
this.user.next(null);
```

In questo modo, il componente *HomepageComponent*, che gestisce i dati dell'utente autenticato, consentendone la corretta visualizzazione, è in grado di osservare tutti i cambiamenti avvenuti nella sessione e mostrare sempre i dati aggiornati:

```
this.userSub = this._sessionService.user.subscribe((user) => {  
    this.isSessionOpen = !user ? false : true;  
    this.userName = !user ? 'Ospite' : user.name;  
    this.userEmail = !user ? '' : user.email;  
});
```

Singleton

Il *pattern Singleton* è un *design pattern* creazionale che garantisce l'esistenza di una sola istanza di una classe all'interno di un'applicazione. Questo viene raggiunto limitando l'accesso al costruttore della classe e fornendo un metodo statico per ottenere l'istanza unica. L'istanza viene creata la prima volta che il metodo di accesso viene chiamato e successivamente viene restituita in tutte le chiamate successive.

Il *pattern* viene utilizzato nel progetto in tutti i servizi, ogni file **.service.ts* presenta infatti la direttiva *injectable*. Se in un servizio d'esempio viene inserita come segue:

```
@injectable({  
    providedIn: 'root',  
})  
export class MyService {
```



```
//...  
}
```

essa indica ad *Angular* di fornire un'unica istanza di *MyService* in tutto il modulo radice dell'applicazione.

Dependency injection

Il *Dependency Injection (DI)* in *Angular* è un meccanismo per gestire le dipendenze tra le classi. Le classi dichiarano le loro dipendenze nel costruttore e *Angular* si occupa di fornirle automaticamente attraverso i provider configurati. Questo permette una maggiore modularità, separazione e facilità di testabilità. Il *DI* risolve le dipendenze cercando i *provider* associati e creando le istanze necessarie, che vengono poi passate alle classi che le richiedono. Questo permette un codice più pulito, riusabile e manutenibile.

Nel progetto viene fatto largo uso del *pattern*, ovviamente in linea con il funzionamento del *framework*, in particolare:

- **Registration component:** il cui costruttore ha la seguente firma:

```
constructor(  
    private _snackBar: MatSnackBar,  
    private _router: Router,  
    private _registrationService: RegistrationService,  
    private _ccService: ClockCaptchaService  
)
```

- **Login component:** il cui costruttore ha la seguente firma:

```
constructor(  
    private _snackBar: MatSnackBar,  
    private _router: Router,  
    private _sessionService: SessionService,  
    private _ccService: ClockCaptchaService  
)
```

- **Homepage component:** il cui costruttore ha la seguente firma:

```
constructor(  
    private _sessionService: SessionService,  
    private _router: Router  
)
```

- **Registration service:** il cui costruttore ha la seguente firma:

```
constructor(  
    private _http: HttpClient  
)
```

- **Session service:** il cui costruttore ha la seguente firma:

```
constructor(  
    private _http: HttpClient,  
    private _cookieService: CookieService  
)
```

)

3.2.2 clock-CAPTCHA

Il "sistema in grado di distinguere un utente umano da un robot", così come richiesto, viene fornito sotto forma di libreria *javascript*, il cui sviluppo è stato però in *typescript*.

La libreria è composta da una parte che permette di mostrare e raccogliere dati dall'utente, quindi utilizzabile nel *front-end* e da una seconda parte che permette la creazione e la validazione dei dati utilizzati dalla prima.

Viene di conseguenza divisa l'analisi dell'architettura di dettaglio in *Presentation* e *Application tiers*.

3.2.2.1 Presentation tier

La parte di *front-end* è composta solamente dalla classe *ClockCAPTCHAView*, che rappresenta il seguente modulo *HTML*:



Figura 4: Modulo HTML rappresentato da *ClockCAPTCHAView*

È composto principalmente da una *HTMLCanvas* dove è contenuta l'immagine, un *HTMLInputElement* che raccoglie i dati dall'utente e infine da un *HTMLElement* (*paragraph*) che viene utilizzato dall'applicazione per comunicare con quest'ultimo.

3.2.2.1.1 Diagramma delle classi

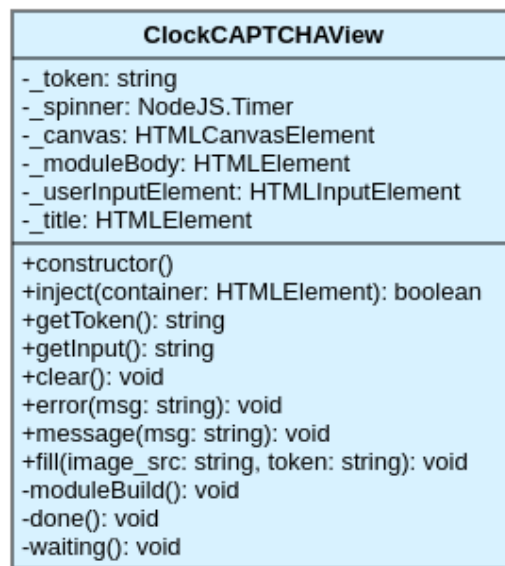


Figura 5: Diagramma della classe ClockCAPTCHAView.

3.2.2.1.2 Interfaccia della classe

Dove non indicato il tipo di ritorno è *void*.

Firma	Descrizione
Interfaccia Pubblica	
<code>constructor()</code>	Costruisce il modulo HTML che rappresenta i dati di <i>clockCAPTCHA</i> , il modulo è composto da elementi HTML tra i quali i principali sono la <i>canvas</i> che contiene l'immagine, il campo di <i>input</i> che raccoglie i dati dall'utente, e il paragrafo che viene usato per comunicare con quest'ultimo. Alla costruzione dell'oggetto all'interno della <i>canvas</i> viene mostrata un'animazione di caricamento.
<code>inject(container: HTMLElement)</code>	Inserisce il modulo HTML all'interno del contenitore indicato come parametro.
<code>fill(src: string, token: string)</code>	Termina l'animazione di caricamento, inserisce all'interno della <i>canvas</i> l'immagine rappresentata da <code>src</code> e inizializza il campo dati <code>_token</code> .
<code>getToken(): string</code>	Ritorna il contenuto del campo dati <code>_token</code> e lo pulisce.

<code>getInput(): string</code>	Ritorna il contenuto del campo di input.
<code>error(msg: string)</code>	Utilizza il paragrafo contenuto nel modulo per comunicare un messaggio di errore di colore rosso all'utente.
<code>message(msg: string)</code>	Utilizza il paragrafo contenuto nel modulo per comunicare un messaggio all'utente.
<code>clear()</code>	Pulisce il contenuto della <i>canvas</i> e riattiva l'animazione di caricamento, pulisce inoltre il campo dati <code>_token</code> e il campo <i>input</i> .
Interfaccia privata	
<code>moduleBuild()</code>	Costruisce il modulo <i>HTML</i> sotto tutti gli aspetti: strutturale, visivo e comportamentale.
<code>waiting()</code>	Avvia l'animazione di caricamento all'interno della <i>canvas</i> .
<code>done()</code>	Termina l'animazione di caricamento all'interno della <i>canvas</i> .

Tabella 4: Interfaccia della classe `clockCAPTCHAView`.

3.2.2.2 Application Tier

La parte di *back-end* della libreria sviluppata ha due funzioni:

- generare dati che possono essere rappresentati visivamente e restituire la coppia

`{informazione, sua_rappresentazione_visiva}` ,

- confermare la congruenza della seguente coppia

`{informazione, informazione_criptata}` .

. La libreria tenta di generare una rappresentazione visiva quanto più riconoscibile dall'essere umano e quanto meno leggibile da *software*, utilizzando a tal proposito un orologio analogico, strumento tanto sensato per l'essere umano quanto senza senso per un programma.

3.2.2.2.3 Diagramma delle classi

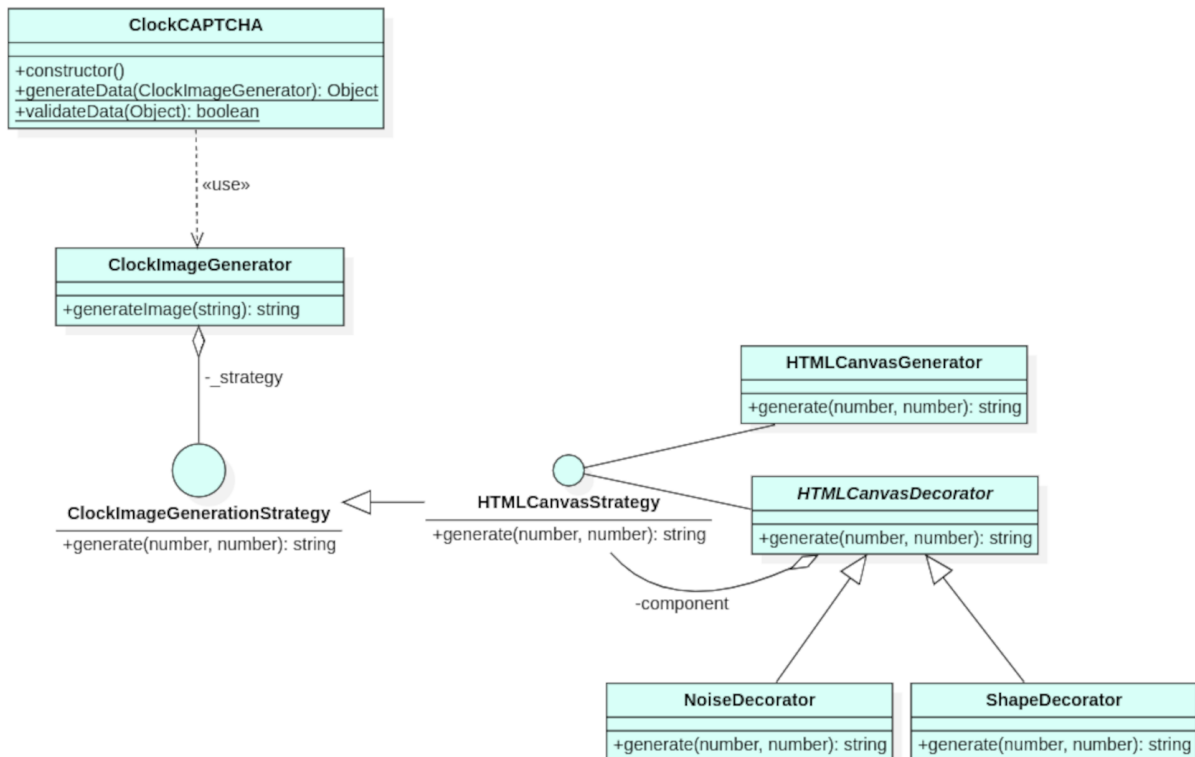
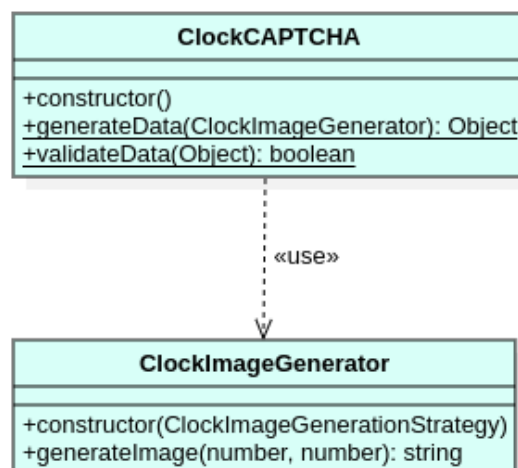


Figura 6: Diagramma delle classi della parte back-end della libreria clockCAPTCHA.

La classe `ClockCAPTCHA` è composta da due funzioni statiche che corrispondono alle due funzioni sopra descritte. La funzione `generateData()` utilizza un oggetto di tipo `ClockImageGenerator` per generare l'immagine, quest'ultimo contiene un oggetto di tipo `ClockImageGenerationStrategy` che implementa l'algoritmo di generazione dell'immagine dal quale deriva al momento della consegna la strategia che utilizza le *HTMLCanvas*.

3.2.2.3 Design patterns utilizzati

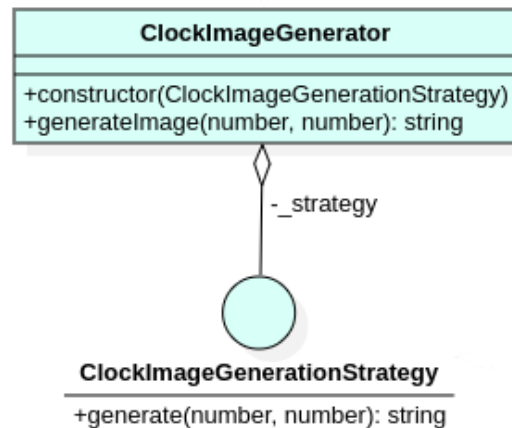
3.2.2.3.4 Dependency injection pattern



Nella funzione `generateData()` viene utilizzato il *dependency injection pattern* per rimuovere la responsabilità di creazione dell'immagine dalla classe che la contiene.

L'utilizzo del *pattern* favorisce la modularità del codice, facilitando la fase di test e migliorando la manutenibilità del prodotto.

3.2.2.3.5 Strategy pattern



La funzione `generateData()` richiede a questo punto un oggetto che si occupi della generazione dell'immagine che è rappresentato da `ClockImageGenerator`.

Durante la fase di progettazione è stato scelto di estrarre l'algoritmo di generazione dell'immagine dalla classe dedicata e di applicare quindi lo *strategy pattern*.

In questo modo si facilita estremamente l'introduzione di nuove tecnologie e relative metodologie al prodotto, rendendo più facile e gestibile anche la fase di test del codice.

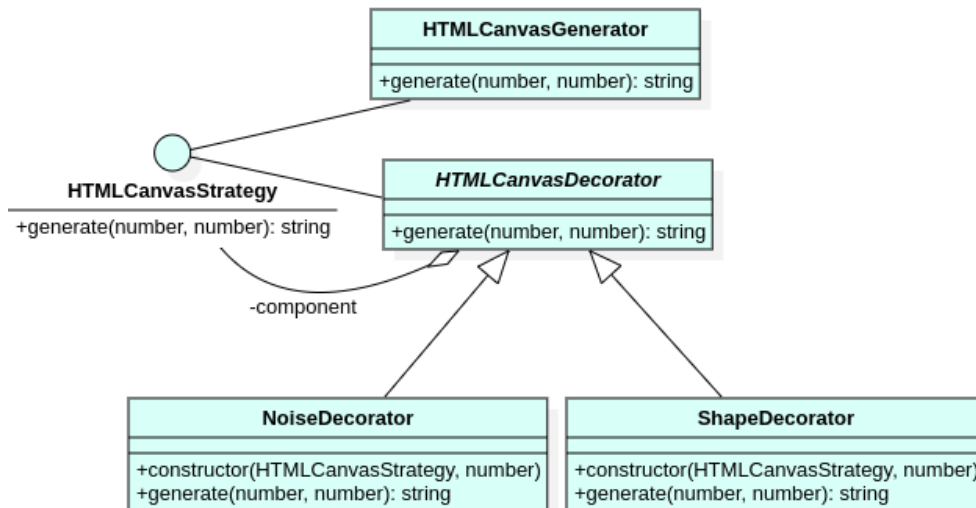
L'implementazione di `generateImage(hours, minutes)` è quindi la seguente

```

generateImage(hours: number, minutes: number) : string{
    return this._strategy.generate(hours, minutes)
}

```

3.2.2.3.6 Decorator pattern



In fase di progettazione viene quindi sviluppata una strategia che vede l'utilizzo delle **HTMLCanvas**. È stata quindi creata inizialmente la componente **HTMLCanvasGenerator** discendente diretta di **ClockImageGeneratorStrategy**.

All'introduzione delle funzionalità di aggiunta del disturbo però la soluzione è stata poi modificata ed è stato introdotto il *decorator pattern*. Il prodotto presentato infatti consente la creazione di immagini con grado e tipo di disturbo personalizzabile. Il *pattern* utilizzato consente di concatenare diverse tipologie di disturbo.

Per utilizzare il *pattern* si procede prima a creare la strategia base che genera l'orologio analogico, poi applicandoci sopra i disturbatori desiderati, utilizzando una concatenazione di costruttori come segue:

```

var orologio =
    new HTMLCanvasGenerator();

var orologio_con_disturbo =
    new NoiseDecorator(orologio, <grado_disturbo>);

var orologio_con_disturbo_con_forme_geometriche =
    new ShapeDecorator(orologio_con_disturbo, <numero_forme_geometriche>);

var generatore_immagine =
    new ClockImageGenerator(orologio_con_disturbo_con_forme_geometriche);
  
```

A questo punto la funzione `generateData()` può ricevere `generatore_immagine` come parametro. Ovviamente quello riportato è un esempio ed esistono diverse possibilità per creare una strategia di generazione dell'immagine, consentendo anche l'utilizzo del servizio in situazioni a fattore di rischio variabile. L'accesso ad un sistema bancario e l'invio di una recensione possono richiedere due generatori diversi.

3.2.3 Database

Il *database* viene gestito con *MySQL* e presenta le seguenti tabelle:

- "users": contenente le informazioni relative agli utenti registrati al *forum*,
- "apikey": contenente le chiavi segrete delle applicazioni a cui è consentito l'utilizzo del *back-end*, per esigenze di progetto è stato prodotto un unico *front-end* ma la tabella consente l'utilizzo di più applicazioni e di conseguenza di più piattaforme,
- "blackList": contenente tutti i *token* generati mediante *clockCAPTCHA* che sono già stati utilizzati almeno una volta. Il *back-end* dell'applicazione infatti genera dei *token* per *ClockCAPTCHAView* che possono essere utilizzati per accedere a funzionalità quali registrazione o autenticazione. Quando riceve una richiesta che prevede la validazione dei dati di *ClockCAPTCHA*, viene controllata la presenza del *token* ricevuto all'interno della tabella e solo se la risposta è negativa viene utilizzato. A prescindere dal risultato della funzione di validazione infatti il tentativo di utilizzo del *token* comporta il suo inserimento all'interno della tabella.

3.2.3.1 Diagramma ER

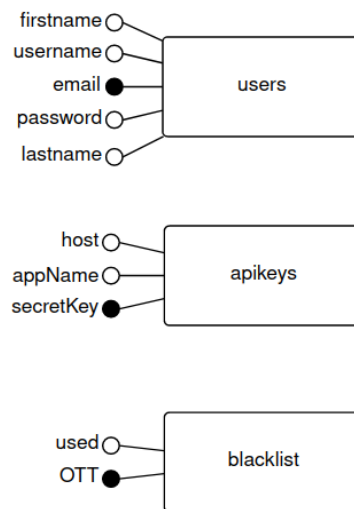


Figura 7: Diagramma ER del database utilizzato dall'applicazione.

3.2.3.2 Diagramma UML

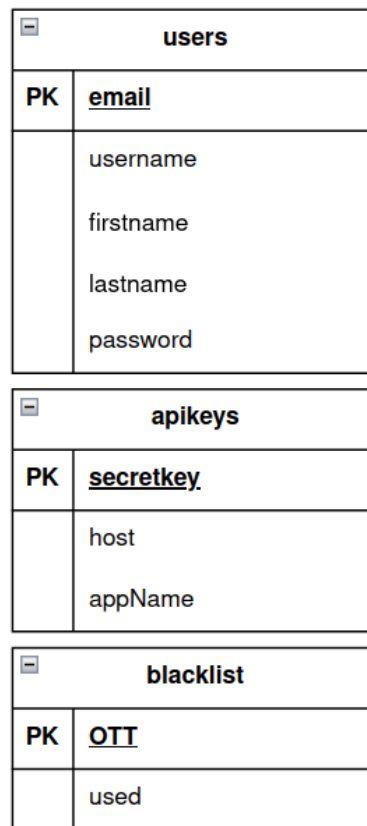


Figura 8: Diagramma UML del database utilizzato dall'applicazione.