



“ *If a test fails, there is a problem  
in the program. If a test succeeds,  
there is a problem in the test.* ”

# Indice

<b>1</b>	<b>Introduzione al progetto ed applicativi utilizzati</b>	<b>4</b>
1.1	Introduzione a JSON & JWT . . . . .	5
1.1.1	Formato JSON . . . . .	5
1.1.2	JWT: JSON Web Token . . . . .	5
1.2	Introduzione Docker . . . . .	9
<b>2</b>	<b>Keycloak</b>	<b>10</b>
2.1	Introduzione a Keycloak . . . . .	10
2.2	Docker e configurazione Keycloak . . . . .	11
2.3	Creazione di un Realm . . . . .	13
2.4	Creazione di un nuovo user . . . . .	14
2.5	Creazione di un Client . . . . .	18
2.6	Creazione di un ruolo associato ad un client in keycloak . . . . .	20
2.7	Creazione utenti ed associazione ruoli . . . . .	21
2.7.1	Come ottenere un token una volta definito un realm ed i relativi utenti . . . . .	22
<b>3</b>	<b>Progetto Back-End</b>	<b>25</b>
3.1	NodeJS . . . . .	25
3.1.1	Introduzione a NodeJS . . . . .	25
3.1.2	Applicazione pratica . . . . .	26
3.2	Proteggere le APIs tramite Keycloak . . . . .	30
3.3	Connessione al database NoSQL MongoDB . . . . .	33
3.3.1	Introduzione MongoDB . . . . .	33
3.3.2	Applicazione pratica . . . . .	33
<b>4</b>	<b>Progetto Front-End</b>	<b>39</b>
4.1	Flutter . . . . .	39
4.1.1	Introduzione . . . . .	39
4.1.2	Applicazione pratica . . . . .	40
4.1.3	Come testare il prodotto finale . . . . .	48

# Capitolo 1

## Introduzione al progetto ed applicativi utilizzati

Il progetto associato a tale documentazione punta a mostrare la semplicità con cui è possibile autenticare un utente tramite un determinato client e, sulla base delle proprie autorizzazioni effettuare un controllo selettivo sulle API a cui esso potrà accedere.

Viene costruita quindi un architettura Client-Server formata da da:

- Client sviluppato mediante Flutter e Dart
- Server (authentication + authorization) Keycloak
- Server (provider di servizi) sviluppato mediante framework NodeJS.

Prima di poter procedere con la definizione dei passi da effettuare per raggiungere il risultato finale bisogna fare prima una breve introduzione delle tecnologie utilizzate.

## 1.1 Introduzione a JSON & JWT

### 1.1.1 Formato JSON

JSON è un formato di serializzazione basato su testo per lo scambio di dati, principalmente tra server e applicazione web. JSON è l'acronimo di JavaScript Object Notation, e utilizza l'estensione di file .json.

JSON è un concorrente di XML, ma possiede una sintassi più semplice e compatta rispetto al rivale.

Il formato JSON è basato su due tipi di strutture di dati:

- serie di coppie nome/valore

```
{  
  "name1": "value1",  
  "name2": "value2"  
}
```

- e liste ordinate di valori, chiamate array

```
{  
  "elementi": [  
    "valore1",  
    "valore2"  
  ]  
}
```

Questi dati vengono letti da un parser JSON che li converte nel tipo di dati appropriato per il linguaggio di programmazione usato nel recupero dei dati.

### 1.1.2 JWT: JSON Web Token

Lo standard definito dall'RFC 7519, rappresentato da JSON Web Token (di seguito JWT) definisce una metodologia che permette di trasmettere un'informazione come oggetto JSON in maniera sicura fra due parti. JWT infatti può essere firmato digitalmente mediante algoritmo HMAC (segreto condiviso) o mediante coppia di chiavi pubbliche e private mediante algoritmo RSA.

Inoltre, proprio perchè JWT contiene al suo interno le informazioni che vorremmo trasmettere, è possibile applicare su di esso uno strato di crittografia in maniera tale da garantire allo stesso tempo fiducia e segretezza.

Un token JWT è costituito da 3 differenti parti che sono:

1. Header
2. Payload
3. Signature

Si presenta nella forma `header.payload.signature` ed è codificato sotto forma di stringa base64-encoded:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaXNTb2NpYWwiOnRydWV9.4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Figura 1.1: Esempio token JWT

L'utente della nostra applicazione quindi una volta eseguito il login tramite il server di autenticazione riceverà come risposta un *access\_token* rappresentato sotto forma di oggetto JWT tale per cui ogni volta che l'utente vuole effettuare l'accesso ad una risorsa protetta dovrà aggiungere all'header della richiesta il seguente:

Authorization: Bearer <JWT Token>

Il server, provider di servizi, comunicherà con il server di autenticazione in maniera tale da verificare l'autenticità del token ricevuto e quindi in funzione dell'autorizzazione concessa all'utente elargirà il servizio richiesto.

Un esempio di flusso di questo tipo è rappresentato infatti dalla seguente figura:

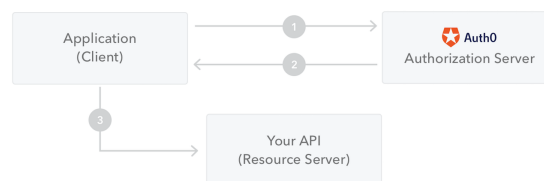


Figura 1.2: Flusso di autorizzazione utente mediante richiesta token

Prima di procedere quindi ad eseguire un confronto tra JWT e SAML, è importante sottolineare che:

- Il periodo di validità di un token JWT deve essere tale da garantire un compromesso tra usabilità e sicurezza. Di base, infatti, un token JWT scade dopo un periodo di 300 secondi (modificabile in funzione della necessità).
- Nonostante un token JWT sia signed, in assenza di uno strato di crittografia superiore, il suo contenuto è pubblicamente accessibile (non modificabile), per cui all'interno del payload JWT non dovrebbero essere inserite informazioni che devono rimanere segrete.

I vantaggi legati all'utilizzo di JWT piuttosto che la controparte SAML sono legati alle seguenti caratteristiche:

- JSON è (molto) meno verboso di XML implicando così che la dimensione di un token codificato JWT risulta essere più efficiente in termini di spazio rispetto a SAML, garantendo così minor overhead quando questo viene trasferito tramite header HTTP.
- Da un punto di vista di sicurezza invece sia JWT che SAML presentano le medesime caratteristiche, tuttavia il processo di firma e crittografia di un token JWT risulta essere più semplice ed efficiente.
- Infine, proprio per la semplicità di interpretazione delle informazioni rappresentate sotto forma di documento JSON, ad oggi quasi ogni linguaggio di programmazione supporta un meccanismo di traduzione JSON-Object implicando così maggior semplicità nell'utilizzo dei token JWT.

Di seguito è riportato un confronto delle dimensioni di un token JWT e l'equivalente SAML.

The figure illustrates the size difference between JWT and SAML tokens. The top screenshot shows a JWT token being decoded on `jwt.io`. The token is a long string of base64-encoded characters. The decoded payload is a small JSON object containing user information. The bottom screenshot shows a SAML token being decoded on `samitool.io`. The token is a long string of base64-encoded XML. The decoded payload is a large, complex XML document containing SAML assertions and signatures. A green circle with the text "VS" is placed between the two screenshots to indicate the comparison.

**JWT Token (Encoded):**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij01YWRtaW41OnRydWV9.TjVA95OrM7E2cBab30RMhrrHDceFxfjoYZgeFONfh7HgQ
```

**JWT Token (Decoded):**

```
{
  "alg": "HS256",
  "typ": "JWT"
}

{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

**SAML Token (Encoded):**

```
PHNhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij01YWRtaW41OnRydWV9.TjVA95OrM7E2cBab30RMhrrHDceFxfjoYZgeFONfh7HgQ
```

**SAML Token (Decoded):**

```
<?xml version='1.0' encoding='UTF-8'>
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" ID="s_621c4...
  <saml:Issuer xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" urn:matu...
  </saml:Issuer>
  <saml:Status>
    <saml:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status...
  </saml:Status>
  <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" Version="...
    <saml:Issuer urn:matuigit.auth0.com/<saml:Issuer>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo>
        <CanonicalizationMethod Algorithm="http://www.w3.org/2...
        <SignatureMethod Algorithm="http://www.w3.org/2000/09...
        <Reference URI="#_SVK7LT77fUkkaQuW64brFODG5E3...
      </SignedInfo>
      <SignatureValue>Fgpt7AaHcMEZgTA158achvGGVdDwHSH...
```

Figura 1.3: JWT vs SAML



## 1.2 Introduzione Docker

Docker è una piattaforma open-source per lo sviluppo, il lancio e l'esecuzione delle applicazioni.



Figura 1.4: Docker Logo

Docker permette di separare la specifica applicazione dall'infrastruttura su cui essa poggia così da ridurre significativamente il tempo necessario al rilascio del codice.

Docker sfrutta un architettura client-server dove, il client interagisce (direttamente localmente o, nel caso in cui *dockerd* fosse attivo su un host remoto, mediante REST API, Sockets o tramite interfaccia di rete) con un demone detto *Docker Daemon* il cui compito è quello di buildare, eseguire e distribuire i container.

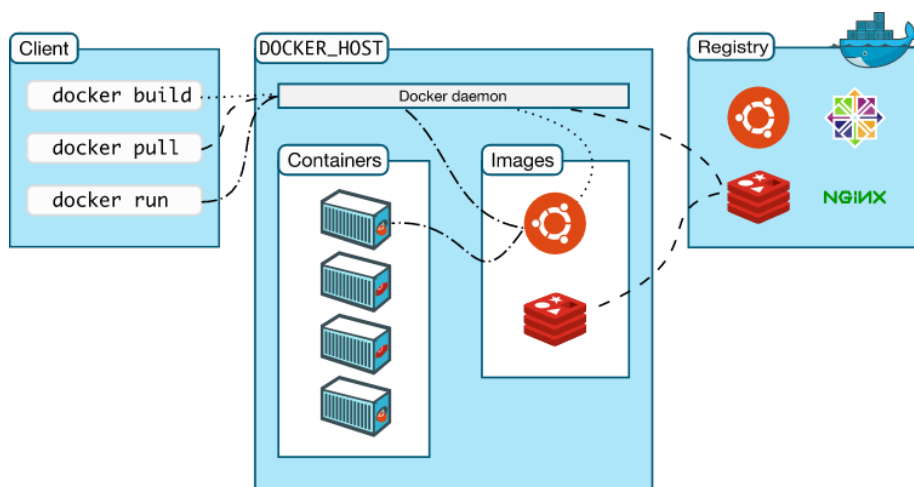


Figura 1.5: Architettura Docker

Il vantaggio di utilizzare Docker è legato al fatto che le applicazioni sviluppate vengono appunto eseguite in ambienti chiamati 'Container' i quali garantiscono di lavorare con un sistema operativo virtualizzato senza dover gestire l'overhead legato alla virtualizzazione di un sistema operativo.

Successivamente appunto verranno creati ed utilizzati una serie di container Docker con l'obiettivo di rendere più semplice il delivery di tale elaborato.

## Capitolo 2

# Keycloak

### 2.1 Introduzione a Keycloak

Keycloak è una soluzione open-source per la gestione di autenticazione ed autorizzazione. Esso presenta internamente un database relazionale utilizzato per gestire, conservare e interrogare le informazioni degli utenti.

“ Add authentication to applications and secure services with minimum fuss. No need to deal with storing users or authenticating users. It's all available out of the box. ”

Cioè che Keycloak permette di sviluppare un'applicazione sicura senza dover affrontare la complessità di gestire internamente i differenti livelli di sicurezza associabili all'utente.

Keycloak può essere riassunto in 3 differenti componenti:

1. **Realm:** Un realm gestisce le informazioni legate alla sicurezza di un insieme di utenti, applicazioni e clients.
2. **Client:** Un client rappresenta l'entità che può richiedere l'autenticazione di un utente in un determinato realm.
3. **Role:** I ruoli identificano le differenti categorie di utenti, permettendo di applicare un controllo a grana fine sulle capabilities che possiede ognuna di esse nella nostra applicazione. Keycloak permette di gestire 3 differenti tipologie di ruoli che sono:
  - Realm-Role level

- Client-Role level
- Composite-Role level che permette di comporre ruoli differenti fra loro e che verrà sfruttato successivamente per associare il ruolo a livello client al ruolo a livello reame.

## 2.2 Docker e configurazione Keycloak

Affinchè sia possibile interagire con keycloak bisogna configurare un server Keycloak. Per far ciò creiamo un container Docker.

Avviamo quindi un ambiente Keycloak con username e password "admin" sfruttando il container "quay.io/keycloak/keycloak:15.0.2" su porta 5050

```
docker run
-p 5050:8080
-e KEYCLOAK_USER=admin
-e KEYCLOAK_PASSWORD=admin
quay.io/keycloak/keycloak:15.0.2
```

a questo punto viene aperto in browser la pagina localhost:5050 dalla quale è possibile accedere a:

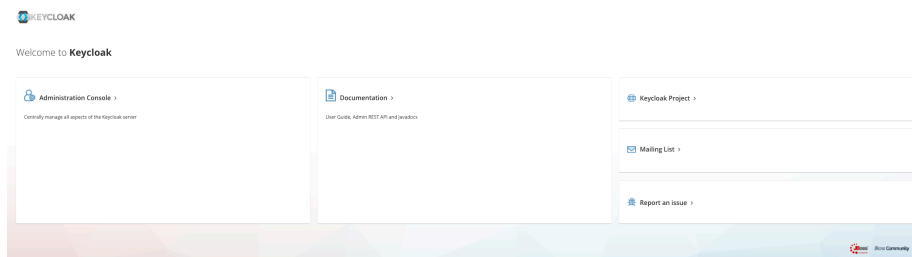


Figura 2.1: Home page Keycloak

- Administration Console
- Documentation

Oltre che a presentare una serie di hyperlinks per:

- Accedere al Keycloak project (url che punta al sito ufficiale di keycloak)
- Accedere alla Mailing List

- Effettuare una segnalazione di malfunzionamento

Il prossimo passo quindi è quello di accedere al pannello di amministrazione tramite Administration Console.

A questo punto si inseriscono le credenziali specificate su tale per cui:

- username: admin
- password: admin

Una volta effettuato l'accesso sarà possibile accedere a differenti sezioni del progetto.

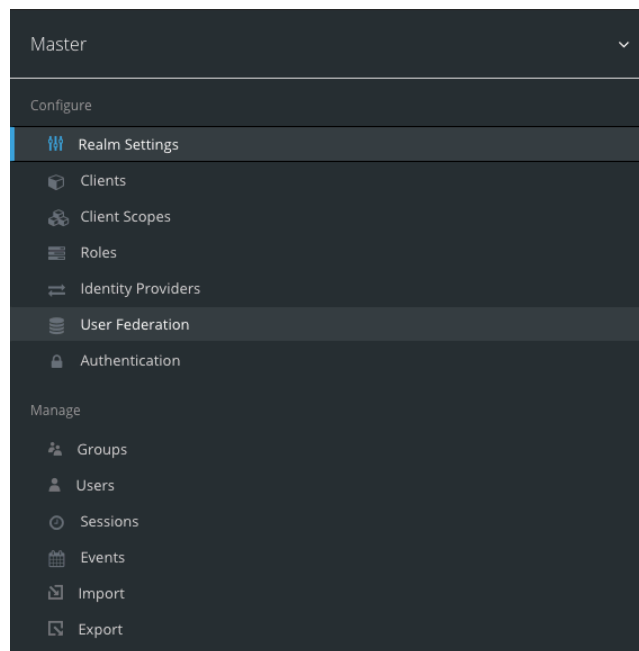


Figura 2.2: Pannello di amministrazione del server Keycloak

## 2.3 Creazione di un Realm

Un reame in Keycloak permette di creare degli ambienti isolati per la gestione di utenti e applicazioni. Keycloak stesso fa utilizzo di un reame "master" il quale è utilizzato per gestire le risorse keycloak.

E' importante sottolineare che dal punto di vista pratico il realm "master" non dovrebbe essere utilizzato per applicazioni terze, le quali dovrebbero interagire con keycloak mediante un realm dedicato.

Per creare un reame, bisogna fare hover sul dropdown che riporta il realm corrente e selezionare "add realm".

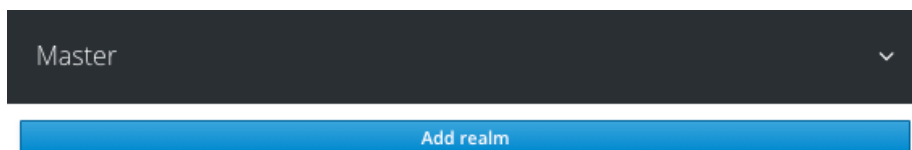


Figura 2.3: Creazione nuovo Reame in Keycloak 1/2

Una volta selezionato "add realm", bisognerà specificare il nome del progetto a cui il realm è associato, nel nostro caso "SOASEC" e quindi procedere con la creazione del nuovo realm.

Add realm

Import

Name \*

Enabled

☒ ON

Figura 2.4: Creazione nuovo Reame in Keycloak 2/2

## 2.4 Creazione di un nuovo user

Appena creato il realm, il passo successivo sarà quello di creare un nuovo utente, con privilegi minori, il quale sarà in grado di accedere tramite keycloak alle applicazioni associate.

Affinchè sia possibile creare un nuovo utente dobbiamo selezionare sotto la voce "Manage" il pulsante "Users" e quindi "Add User".

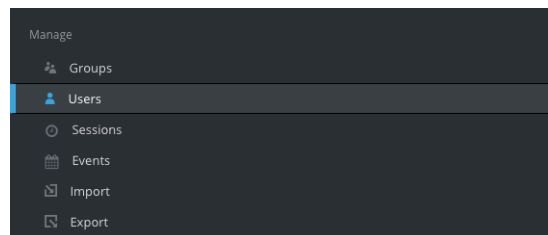


Figura 2.5: Sezione Utenti

Vengono quindi inseriti i dati dell'utente come segue:

Add user

ID

Created At

Username \*

Email

First Name

Last Name

User Enabled ⓘ  
☒ ON

Email Verified ⓘ  
☐ OFF

Groups ⓘ  
No group selected

Required User Actions ⓘ

Figura 2.6: Creazione di un nuovo utente

Si evidenzia che:

- L'unico campo non opzionale è rappresentato dal campo Username (associato ad un asterisco rosso)
- L'utente può essere "abilitato" o "disabilitato", un utente disabilitato è un utente registrato il quale non può effettuare l'accesso alla piattaforma.
- Si può specificare se la mail associata all'utente è una mail verificata (o alternativamente effettuare richiesta di verifica tramite la voce riportata all'ultimo punto)
- E' possibile associare un utente ad un gruppo in maniera tale da permettere all'utente di ereditare determinate caratteristiche comuni al gruppo.
- Infine, è possibile tramite la voce "Required User Actions" è possibile specificare delle operazioni che dovranno essere eseguite in maniera mandatoria da parte dell'utente, come ad esempio le seguenti:

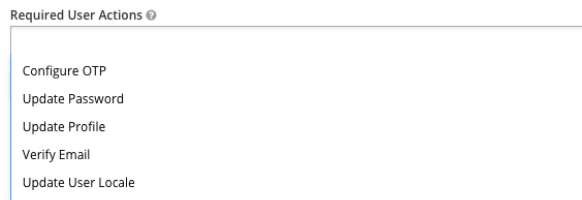


Figura 2.7: Differenti azioni richiedibili all'utente

Una volta inseriti tutti i dati e confermate le opzioni associate, è possibile creare il nuovo utente tramite il pulsante "save" che ne attesta la creazione dell'utente ed il salvataggio delle varie informazioni associate allo stesso.

Verrà quindi effettuato un redirect alla pagina di dettaglio dell'utente appena creato. Tale pagina di dettaglio ci permette di navigare le impostazioni associate all'utente tramite le seguenti voci:

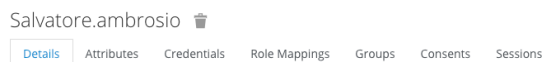


Figura 2.8: Lista sezioni utente salvatore.ambrosio

Una volta giunti a tale fase, l'obiettivo attuale è quello di associare all'utente appena creato una nuova password, in maniera tale così da rendere possibile la fruizione dell'utente stesso presso le applicazioni descritte nei capitoli successivi.

Per far ciò bisognerà navigare la dashboard utente alla voce "Credentials".

The screenshot shows the 'Credentials' tab for the user 'Salvatore.ambrosio'. The interface includes tabs for 'Details', 'Attributes', 'Credentials' (selected), 'Role Mappings', and 'Groups'. Below these are 'Consents' and 'Sessions'. The 'Manage Credentials' section contains a table with columns: Position, Type, User Label, Data, and Actions. Below the table is the 'Set Password' section, which includes a 'Password' field, a 'Password Confirmation' field, and a 'Temporary' toggle switch set to 'ON'. A 'Set Password' button is at the bottom.

Position	Type	User Label	Data	Actions
----------	------	------------	------	---------

Set Password

Password

Password Confirmation

Temporary ☒

Set Password

Figura 2.9: Impostazione nuova password per l'utente salvatore.ambrosio 1/2

Tramite tale sezione sarà possibile inserire la password che l'utente potrà utilizzare per accedere all'ambiente Keycloak.

E' importante sottolineare la possibilità di rendere tale password una password temporanea o definitiva. Nel caso in cui lo switch "Temporary" risultasse attivo, appena l'utente effettuerà il primo login, gli verrà richiesto di impostare una nuova password definitiva.

Si procede dunque con l'associazione della password all'utente appena creato come in figura 2.10:



## Set Password

Password



Password Confirmation



Temporary 

☒ ON

Set Password

Figura 2.10: Impostazione nuova password per l'utente salvatore.ambrosio 2/2

Keycloak utilizza l'algoritmo di hashing *pbkdf2-sha256* che effettua un round di 27500 iterazioni al fine di poter memorizzare la password in maniera sicura all'interno del database e proteggere tali password da un possibile leak.

E' possibile osservare nella sezione di dettaglio account, che sotto la voce "Required Actions" è comparsa la spunta "Update Password". Questo è dovuto al fatto che la password precedentemente specificata era temporanea.

Proviamo quindi ad effettuare l'accesso tramite le credenziali appena memorizzate tramite la Keycloak Account Console all'endpoint `/auth/realms/SOASEC/account/`, e, come aspettato, il sistema ci presenta un form di aggiornamento della password. Tale operazione è necessaria per l'attivazione dell'utente.

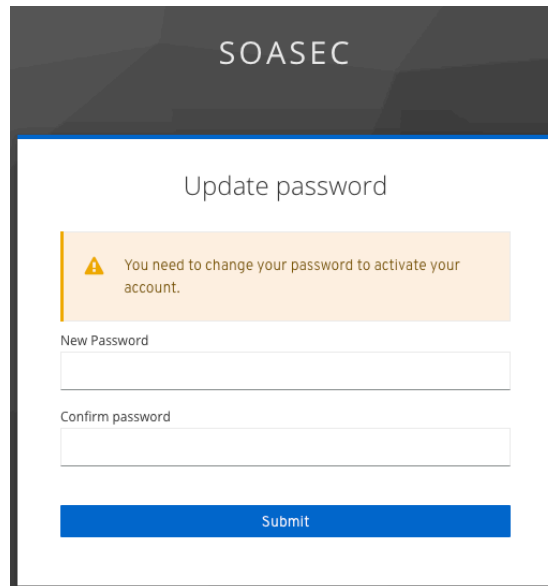


Figura 2.11: Azione richiesta all'utente poichè password temporanea

Procediamo quindi con l'impostazione della nuova password (super) sicura

*Password123!*

Affinchè quindi sia possibile definire una determinata politica di accesso alla nostra applicazione e quindi consentire alla stessa di autenticare un utente risulterà necessario associarvi un Client.

## 2.5 Creazione di un Client

I client sono le entità che possono richiedere a Keycloak di autenticare un utente. Molto spesso, i client sono associati ad applicazioni che vogliono usufruire di Keycloak per proteggere i servizi esposti sfruttando la tecnologia SSO (Single-Sign-On).

Affinchè sia possibile creare un nuovo client bisogna selezionare la voce "Clients" sotto la sezione "Configure" e dunque dalla dashboard proposta cliccare sul pulsante "Create".

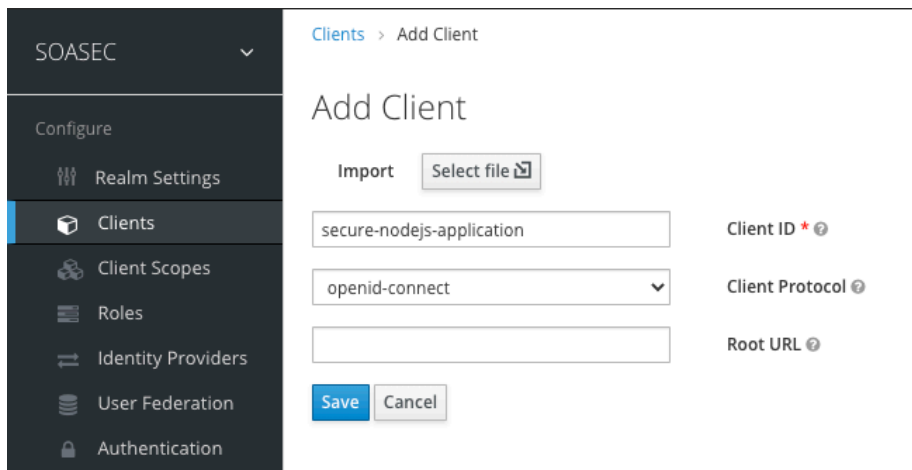


Figura 2.12: Dashboard creazione client

Una volta specificato l'identificativo del client, possiamo concludere la creazione. Si sottolinea che è una buona pratica associare all'identificativo del client una stringa rappresentativa dell'applicazione, in maniera tale da rendere più semplice la gestione nel caso in cui vi fossero svariati client nell'ecosistema SSO offerto da Keycloak.

Creato il client, si imposta l'accesso come *confidential* e si attivano i servizi *Service Accounts Enabled* e *Authorization Enabled*, fatto ciò saremo in grado di autenticare il client e quindi ottenere token dedicati per tale client.

A questo punto si sarà in grado di recuperare sotto la voce *Credentials* associata al client appena creato il *Secret* associato allo stesso:

*e51bfd3e-6634-4319-bafa-e8022ee9f03b*

Keycloak rende quindi possibile definire dei ruoli per-client in maniera tale da poter associare a determinate API un determinato ruolo richiesto.

Tipicamente infatti un'applicazione presenta almeno due differenti ruoli, Utente ed Amministratore, per tale tipologia di applicazione dunque potrebbero essere esposte differenti tipologie di servizi sotto forma di API:

- Alcune API potrebbero essere pubblicamente accessibili
- Alcune API potrebbero essere accessibili solo da Utenti
- Alcune API potrebbero essere accessibili solo da Amministratori
- Alcune API potrebbero essere accessibili sia da Utenti che Amministratori

Per tale motivazione dunque procediamo alla creazione di questi ruoli che utilizzeremo successivamente per autorizzare le richieste provenienti dall'esterno.

## 2.6 Creazione di un ruolo associato ad un client in keycloak

Affinchè sia possibile creare un nuovo ruolo in keycloak bisogna navigare la dashboard associata al client con identificativo *secure-nodejs-application* alla sezione *Roles* e dunque creare i ruoli:

- user
- administrator

Corrediamo in entrambi i casi una breve descrizione del ruolo, in maniera tale da rendere più semplice la manutenzione in presenza di più ruoli associati ad un singolo client.

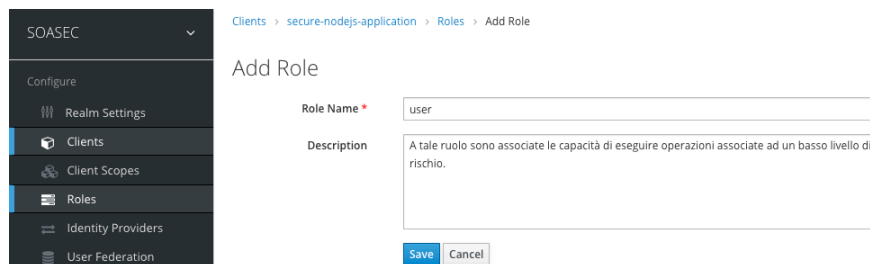
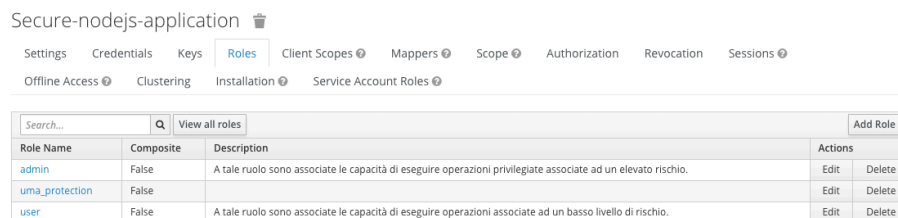


Figura 2.13: Creazione client-role

Viene quindi creato anche il ruolo *app-user*.

In figura è quindi riportato un riepilogo dei ruoli attualmente presenti ed associati al client *secure-nodejs-application*.



Secure-nodejs-application			
Settings Credentials Keys Roles Client Scopes Mappers Scope Authorization Revocation Sessions			
Offline Access Clustering Installation Service Account Roles			
Search... View all roles Add Role			
Role Name	Composite	Description	Actions
admin	False	A tale ruolo sono associate le capacità di eseguire operazioni privilegiate associate ad un elevato rischio.	Edit Delete
uma_protection	False		Edit Delete
user	False	A tale ruolo sono associate le capacità di eseguire operazioni associate ad un basso livello di rischio.	Edit Delete

Figura 2.14: Lista ruoli per client d'esempio

In maniera analoga per il Realm, è possibile creare dei ruoli app-specific, i quali saranno associati ai ruoli appena creati.

Questo perchè spesso le applicazioni garantiscono accesso e autorizzazione a specifici ruoli, piuttosto che ad utenti in quanto un controllo di questo tipo potrebbe essere un controllo a grana troppo fine, diventando così difficile da mantenere all'aumentare del numero degli utenti della nostra applicazione.

Per far ciò dunque selezioniamo dalla sezione *Configure* la voce *Realms* ed a questo punto creiamo i ruoli app-specific che saranno:

- app-user
- app-administrator

Una volta aggiunti tali ruoli, li rendiamo *Composite Role* ed associamo ad entrambi il rispettivo *Client role*.



Figura 2.15: Role composition, associazione realm role app-administrator al client role admin

## 2.7 Creazione utenti ed associazione ruoli

Come detto precedentemente, gli utenti sono le entità che possono eseguire accesso al sistema. Ad ogni utente possono essere associati differenti attributi, come username, email, nome, cognome etc e possono essere associati a gruppi o essere assegnabili a specifici ruoli.

Ciò che faremo quindi sarà quello di associare agli utenti precedentemente creati

- antonio.elefante
- salvatore.ambrosio

dei ruoli, in maniera tale così da poter sfruttare tali ruoli per interagire successivamente con il sistema.

Per far ciò basterà selezionare l'utente a cui vogliamo associare il ruolo, e conseguentemente sotto la sezione *Role Mappings* specificare la categoria desiderata.

Ad esempio, procediamo con l'assegnazione del ruolo *app-administrator* all'utente caratterizzato dall'username *antonio.elefante*.

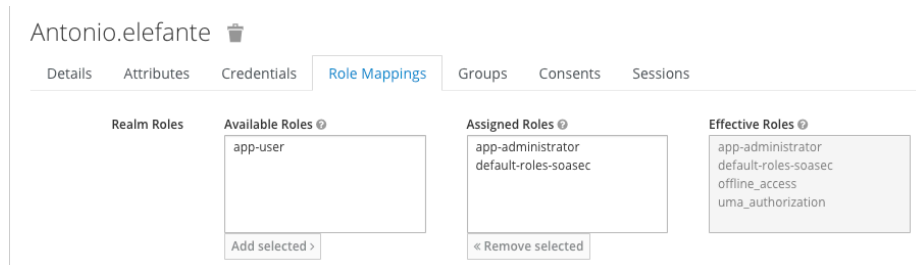


Figura 2.16: Ruoli associati all'utente antonio.elefante

In maniera analoga, associamo all'utente *salvatore.ambrosio* il ruolo *app-user*

### 2.7.1 Come ottenere un token una volta definito un realm ed i relativi utenti

Affinchè sia possibile recuperare il token di autorizzazione, bisogna preventivamente determinare qual è l'endpoint utile per poter richiedere a Keycloak lo stesso.

Per far ciò bisognerà selezionare *Realm settings* dal pannello laterale e quindi cliccare sul collegamento *OpenID Endpoint Configuration*.

Tale link punta ad un endpoint il quale fornisce in output una stringa in formato JSON.

Di tale stringa JSON ci interessa la chiave *token\_endpoint* da cui:

```
{
  "...": "...",
  "token_endpoint": "http://base_url:port/auth/realms
/REALM/protocol/openid-connect/token",
  "...": "...",
}
```

E' possibile quindi ottenere l'access token semplicemente effettuando una richiesta http POST fornendo i seguenti valori di input:

<b>grant_type</b>	password
<b>client_id</b>	secure-nodejs-application
<b>client_secret</b>	e51bfd3e-6634-4319-bafa-e8022ee9f03b
<b>username</b>	antonio.elefante
<b>password</b>	123456789

Tabella 2.1: Coppie Key-Value necessari per l'ottenimento del token JWT

Tale richiesta fornirà in output una stringa in formato JSON contenente in particolare i seguenti:

- *access\_token*: Il token in formato JWT
- *expires\_in*: Il tempo di validità del token appena ricevuto, una volta esaurito tale tempo le richieste effettuate risponderanno con *http code 401-unauthorized*

Tale token può essere autonomamente decodificato tramite [jwt.io](https://jwt.io) e riporta alcuni riferimenti all'utente processato, fra cui il ruolo ad esso associato.

E' importante sottolineare che Keycloak permette di modificare l'algoritmo di generazione dei token come ad esempio per aumentare o diminuire la durata di validità del token stesso.

Una volta completati tali passaggi, eseguiamo un *docker commit* delle modifiche effettuate al server così da poter congelare lo stato attuale del server keycloak e quindi successivamente poter utilizzare il seguente comando per poter riprendere dallo stato attuale:

```
docker run
-p 5050:8080
-e KEYCLOAK_USER=admin
-e KEYCLOAK_PASSWORD=admin
soasec/keycloak
```

PASTE A TOKEN HERE

EDIT THE PAYLOAD AND SECRET

---

*Università degli Studi di Milano La Statale*



## Capitolo 3

# Progetto Back-End

### 3.1 NodeJS

#### 3.1.1 Introduzione a NodeJS

NodeJS è un runtime environment JavaScript open-source e cross-platform.



Figura 3.1: NodeJS logo

Node.js sfrutta a pieno il motore JavaScript V8 sviluppato per rendere più efficiente l'interpretazione del codice JavaScript nel browser Google Chrome.

Un'applicazione NodeJS viene eseguita interamente in un singolo processo, permettendo, tramite un insieme di primitive asincrone di I/O di sviluppare codice non bloccante in grado di rispondere a numerose richieste senza la necessità di creare un nuovo thread per ogni richiesta sopraggiunta.

Ciò permette a NodeJS di poter rispondere a migliaia di richieste concorrenti senza la necessità di dover gestire la concorrenza fra differenti thread che, talvolta, può portare all'introduzione di bugs che potrebbero essere difficili da risolvere.

### 3.1.2 Applicazione pratica

Anche in questo caso viene fatto utilizzo di docker al fine di rendere il lavoro effettuato non solo semplice da mantenere, data proprio la caratteristica di docker nel rendere deterministico il funzionamento di un dato sistema, ma anche di rendere semplice il trasferimento dell'intero progetto.

Creiamo quindi un nuovo progetto nodejs mediante il comando *npm init* il quale ci fa definire una serie di informazioni associate al progetto stesso.

```
{
  "name": "progetto-soasec",
  "version": "1.0.0",
  "description": "Progetto SOASEC:
NodeJS (Back-End),
MongoDB (DB NoSQL),
Keycloak (Authentication+Authorization),
Flutter (Front-End)",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "author": "Antonio Elefante, Salvatore Ambrosio",
  "license": "ISC"
}
```

Creiamo un file *index.js* che rappresenterà la nostra applicazione NodeJS mediante il comando *touch index.js* ed il DockerFile, *touch DockerFile*.

```
FROM node:17

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both
# package.json and package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "index.js" ]
```

Affinchè sia possibile esporre delle API tramite framework back-end NodeJS bisognerà far utilizzo del package *express* il quale fornisce le metodologie per la creazione di un listener, l'interpretazione della richiesta e la generazione di risposta su protocollo http/https.

Pertanto tramite il seguente comando installiamo le dipendenze necessarie per l'utilizzo di express:

```
npm install express
```

A questo punto saremo in grado di creare ed utilizzare un applicazione express mediante il seguente codice, il cui scopo è semplicemente quello di creare un server in ascolto su porta 8080 in grado di rispondere alla sola richiesta *http://base\_url:port/* con un semplice messaggio di stato.

```
var express = require('express');
var app = express();

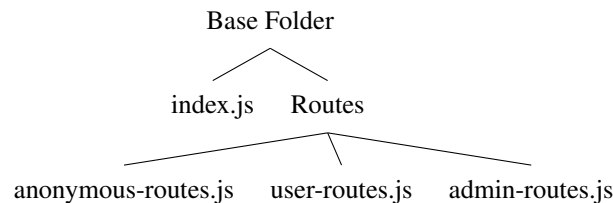
app.get('/', function(req, res){
  res.send("Il server è attivo e funzionante!");
});

app.listen(8080);
```

Una volta verificata la raggiungibilità del server si andrà ad aggiungere funzionalità al fine di esporre la seguente lista di servizi:

- Visualizzazione lista candidati (public/anonymous)
- Votazione candidato (app-user)
- Aggiunta nuovo candidato (app-admin)

Per far ciò dunque strutturiamo il progetto mediante la seguente struttura ad albero:



Nello specifico, i files *\*-routes.js* presenteranno una struttura simile, utile per poter esporre determinati servizi.

Tale caratteristica infatti è legata alla modalità in cui i *moduli* sono gestiti da NodeJS. Ogni modulo infatti richiederà solo ed esclusivamente i package da esso richiesti per poter funzionare, e quindi esporrà all'esterno solo i metodi necessari al corretto funzionamento della funzionalità che si vuole esporre.

Un esempio di file `*-routes.js` è il seguente:

```
var express = require('express');
var router = express.Router();

router.get('/route', function(req, res){
    res.send("Ottimo, hai raggiunto la rotta  
http://base_url/nome_modulo/route");
});

module.exports = router;
```

Una volta definite le rotte esposte dai singoli files, possiamo integrarle facilmente nella nostra applicazione semplicemente segnalando all'applicazione di usufruire delle determinate rotte.

Per far ciò bisogna quindi:

1. Importare il file tramite `require(file)`
2. Utilizzare le funzionalità esposte dal modulo importato.

```
var anonymous_routes = require('./Routes/  
anonymous-routes.js');
var user_routes = require('./Routes/  
user-routes.js');
var admin_routes = require('./Routes/  
admin-routes.js');

app.use('/anonymous', anonymous_routes);
app.use('/user', user_routes);
app.use('/admin', admin_routes);
```

Il problema però è legato al fatto che la semplice strutturazione del progetto non garantisce alcuna funzionalità, né alcun livello di sicurezza.

Nello specifico, per poter aggiungere funzionalità al progetto, collegheremo alla nostra applicazione NodeJS un database di tipo NoSQL il quale verrà utilizzato per memorizzare sotto forma di documenti JSON le informazioni relative alla votazione.

Viceversa, per quanto riguarda la sicurezza, proteggeremo le API con i relativi ruoli mediante Keycloak.

## 3.2 Proteggere le APIs tramite Keycloak

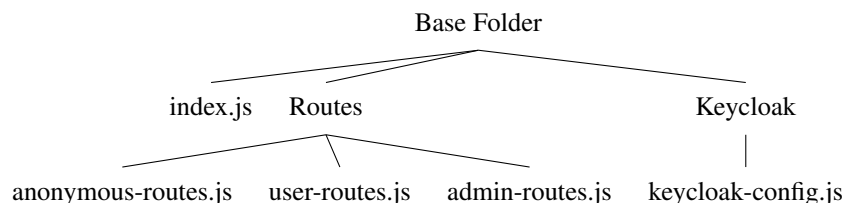
Per poter proteggere le API precedentemente esposte pubblicamente tramite server NodeJS bisogna far utilizzo dei seguenti:

- keycloak-connect
- express-session

Dove, nello specifico, keycloak-connect deve essere configurato in maniera tale da rendere raggiungibile lo specifico server keycloak. Una volta configurato possiamo sfruttare il middleware `keycloak.protect(...)` per aggiungere il layer di sicurezza alle API esposte.

```
npm install keycloak-connect
npm install express-session
```

Viene quindi creata una nuova cartella all'interno del progetto al fine di rendere più semplice la strutturazione del codice.



In particolare si specifica che per finalità progettuali, le informazioni come le chiavi private vengono inserite in maniera `base64-encoded` all'interno del progetto, tuttavia però in ambiente di produzione, sarebbe più opportuno far utilizzo del pacchetto `dotenv` il quale ci permette di memorizzare le chiavi private in un file, appunto chiamato `.env` il quale verrà blacklistato all'interno del file `.gitignore`. Tutto questo per rendere quanto meno possibile pubbliche informazioni che dovrebbero rimanere private.

Procediamo quindi con l'analisi del file `keycloak-config.js`.

Anzitutto, come anticipato sarà necessario importare i moduli `keycloak-connect` e `express-session`:

```
var session = require('express-session');  
var Keycloak = require('keycloak-connect');
```

Una volta importati, procediamo quindi con la specificazione dei parametri di configurazione di keycloak:

```
var keycloakConfig = {  
  clientId: 'secure-nodejs-application',  
  bearerOnly: true,  
  serverUrl: 'http://localhost:5050/auth',  
  realm: 'SOASEC',  
  credentials: {  
    secret: 'e51bfd3e-6634-4319-bafa-e8022ee9f03b'  
  }  
};
```

Viene quindi definita una funzione per ottenere accesso all'oggetto Keycloak:

```

let _keycloak;
function getKeycloak() {
  if (_keycloak) {
    console.log("Keycloak già inizializzato");
    return _keycloak;
  }
  else {
    console.log("Inizializzazione
      connessione al server Keycloak...");
    var memoryStore = new session.MemoryStore();
    _keycloak = new Keycloak(
      {
        store: memoryStore
      }, keycloakConfig);
    return _keycloak;
  }
}

```

Fatto ciò, non ci resta altro che esporre l'accesso alla funzione `getKeycloak()` all'esterno:

```

module.exports = {
  getKeycloak
}

```

Ora modifichiamo il file `index.js` al fine di integrare quanto appena fatto con le seguenti modifiche:

```

const keycloak = require('./Keycloak/
  keycloak-config.js').getKeycloak();
app.use(keycloak.middleware());

```

E quindi implementiamo infine una politica di accesso alle API basata su ruoli mediante l'impiego del middleware come segue:



```
app.use('/anonymous', anonymous_routes);  
app.use('/user', keycloak.protect("user"), user_routes);  
app.use('/admin', keycloak.protect("admin"),  
    admin_routes);
```

### 3.3 Connessione al database NoSQL MongoDB

#### 3.3.1 Introduzione MongoDB



Figura 3.2: MongoDB logo

MongoDB è un database NoSQL orientato ai documenti, che nasce nel 2007 in California come servizio da utilizzare nell'ambito di un progetto più ampio, ma che presto è diventato un prodotto indipendente ed open-source. MongoDB memorizza le informazioni sotto forma di documenti i quali sono a loro volta rappresentati mediante formato JSON.

Le caratteristiche chiave di MongoDB sono:

- Alta replicabilità
- Altamente scalabile: MongoDB infatti permette di distribuire i documenti su più nodi in modo tale così da supportare grandi quantità di dati senza impattare negativamente sulle performance globali del sistema.

MongoDB si adatta a molti contesti, in generale quando si manipolano grandi quantità di dati eterogenei e senza uno schema ben definito.

Non è invece opportuno quando si devono gestire molte relazioni tra oggetti, e si vuole garantire l'integrità referenziale tra essi.

#### 3.3.2 Applicazione pratica

Affinchè sia possibile connettere il server NodeJS ad una base dati MongoDB vi sono due differenti possibilità:

1. Creare una istanza Atlas gratuita presso MongoDB
2. Creare ed eseguire un istanza locale

Ovviamente, per le finalità che si vogliono raggiungere risulta più comodo avviare una istanza locale di MongoDB mediante container Docker.

Per far ciò scarichiamo ed eseguiamo una nuova istanza MongoDB mediante i seguenti comandi:

```
docker pull mongo
```

E successivamente avviamo l'istanza mongo tramite il seguente:

```
docker run  
-p 6060:27017  
-d mongo
```

A questo punto, sempre da terminale all'interno della cartella in cui è sito il progetto è possibile eseguire il seguente comando che assocerà al progetto la dipendenza da MongoDB:

```
npm install mongoose
```

Una volta fatto ciò si potrà importare dove necessario il *package mongoose* il quale potrà essere sfruttato come *driver* per le comunicazioni con il database:

```

var mongoose = require('mongoose');
var mongoDB = 'mongodb://base_url/database';
mongoose.connect(mongoDB,
  {
    useNewUrlParser: true,
    useUnifiedTopology: true
  }
);

```

Non ci resta quindi che definire e strutturare lo schema del database, così da poter interagire ed applicare maggiori controlli ai dati che dovranno essere memorizzati.

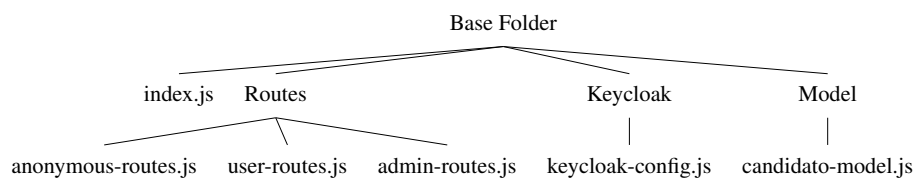
Le informazioni che verranno quindi memorizzate all'interno del database sfrutteranno il seguente schema:

Candidato
nome
cognome
votanti

Tabella 3.1: Candidato Schema

E' importante sottolineare che in un progetto reale, un database costituito da questa singola tabella risulterebbe essere inutile, tuttavia una base dati di questo tipo risulta sufficiente a garantire un livello di astrazione tale da dimostrare di aver compreso le modalità di protezione delle API CRUD.

Per tale motivazione quindi viene creata la cartella *Model* e quindi definiti all'interno di tale cartella il file necessario alla gestione dello schema appena presentato:



Di cui, il file candidato-model.js conterrà il seguente snippet:

```
const mongoose = require('mongoose')

const Candidato = new mongoose.Schema({
  nome: {
    type: 'String',
    required: true
  },
  cognome: {
    type: 'String',
    required: true
  },
  votanti: [{
    type: 'String'
  }]
})

module.exports = mongoose.model("Candidati", Candidato)
```

Una volta fatto ciò, basterà utilizzare tale schema all'interno dei metodi precedentemente definiti al fine di garantire una corretta funzionalità degli stessi.

Per semplicità viene riportato solo un esempio di API (CRUD Creazione nuovo candidato), rimandando al lettore la possibilità di osservare il progetto nella sua interezza.

```

var express = require('express');
var router = express.Router();
var Candidato = require('../Model/candidato-model.js')

router.get('/creazione-nuovo-candidato',
  function(req, res){
    try {
      const nuovoCandidato = new Candidato({
        'nome': req.body.nome,
        'cognome': req.body.cognome,
        'votanti': []
      });
      await nuovoCandidato.save();
      res.send(
        {
          "error": "false",
          "message": "candidato
            creato correttamente"
        }
      )
    } catch (err) {
      res.status(500).send(err);
    }
  }
);

module.exports = router;

```

Una volta definita l'implementazione delle API, è possibile trasformare tale progetto in un container docker al fine di rendere completo e finale il progetto Back-End.

Per far ciò definiamo il file `.dockerignore` al fine di ignorare la copia locale dei package installati tramite npm (*node package manager*) ed infine viene creata la build del container docker:

```

docker build .
-t soasec/nodejs-application

```

A questo punto possiamo eseguire l'applicazione tramite docker mediante il seguente comando:

```
docker run  
-p 49160:8080  
-d soasec/nodejs-application
```

## Capitolo 4

# Progetto Front-End

### 4.1 Flutter

#### 4.1.1 Introduzione



Figura 4.1: Flutter logo

Flutter è un framework open-source creato da Google per lo sviluppo front-end di interfacce grafiche native per iOS e Android, oltre ad essere il metodo principale per la creazione di applicazioni per Google Fuchsia.

Con la versione 1.9, Google introduce il supporto per le applicazioni web e per siti statici scritti in linguaggio Dart.

Tra le caratteristiche peculiari di Flutter si evidenziano:

- Flutter è un linguaggio Cross-Platform

- Sviluppo di interfacce mediante programmazione dichiarativa
- La possibilità di effettuare fast-reload e fast-restart garantendo così migliore efficienza in fase di sviluppo.
- Compatibilità con i sistemi più disparati, Flutter infatti permette di sfruttare una singola code-base su:
  - Mobile: Android, iOS
  - Desktop: Linux, Windows, Mac
  - Web

#### 4.1.2 Applicazione pratica

Il progetto termina il suo sviluppo mediante un applicazione mobile, la quale sfrutta le API precedentemente discusse per garantire per permettere all'utilizzatore di accedere ai contenuti esposti.

Come ampiamente discusso sono previsti 3 livelli differenti di autorizzazione:

- Anonymous/Public
- Utente
- Amministratore

Questo per garantire all'utilizzatore la possibilità di eseguire le 4 differenti operazioni CRUD:

- Create/Creazione [Admin]
- Read/Lettura [Anonymous/Public + User + Admin]
- Update/Aggiornamento [User]
- Delete/Cancellazione [Admin]

E' importante sottolineare che la funzionalità di Aggiornamento, rappresentata dalla possibilità di votare un determinato candidato è attuabile solo ed esclusivamente da un utente con privilegi User. Un utente admin quindi sarà limitato alle sole funzioni di creazione e/o cancellazione nuovi candidati.

L'utilizzatore di tale applicazione sarà quindi in grado di eseguire la seguente lista di operazioni:

- 1.R Visualizzare all'apertura dell'applicazione la lista di candidati ed il relativo punteggio.



2. Effettuare login tramite credenziali Keycloak

3. role Admin:

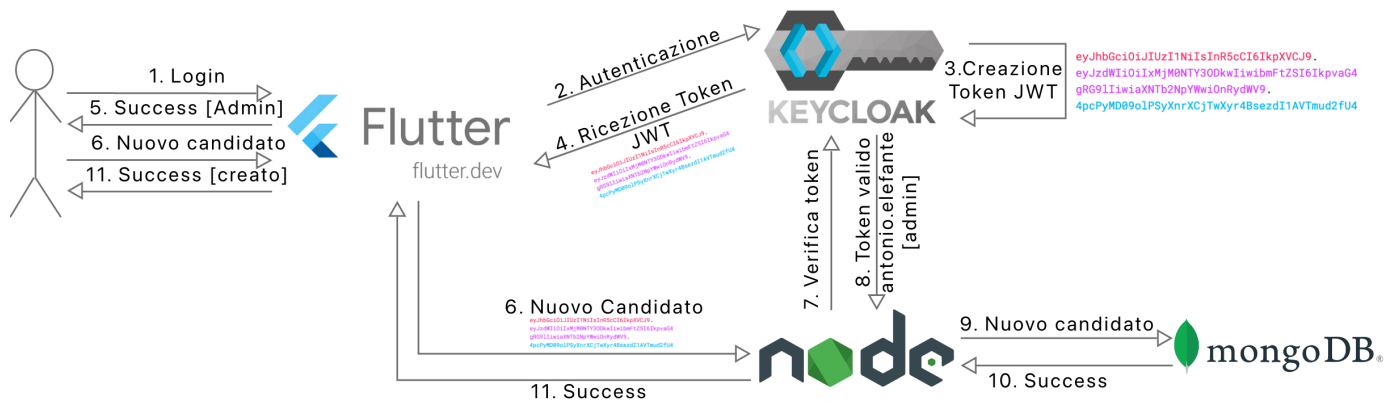
3.C Inserire nuovo candidato

3.D Cancellare un candidato pre-esistente

• role User:

3.U Esprimere la propria preferenza su un candidato

Si riporta di seguito un esempio di flow in l'utente Antonio Elefante [ADMIN] effettua il login e richiede la creazione di un nuovo utente.



Da qui in poi invece sono riportate le schermate che gli utenti Salvatore Ambrosio[USER] e l'utente Antonio Elefante[ADMIN] dovranno far utilizzo per interagire con il sistema.

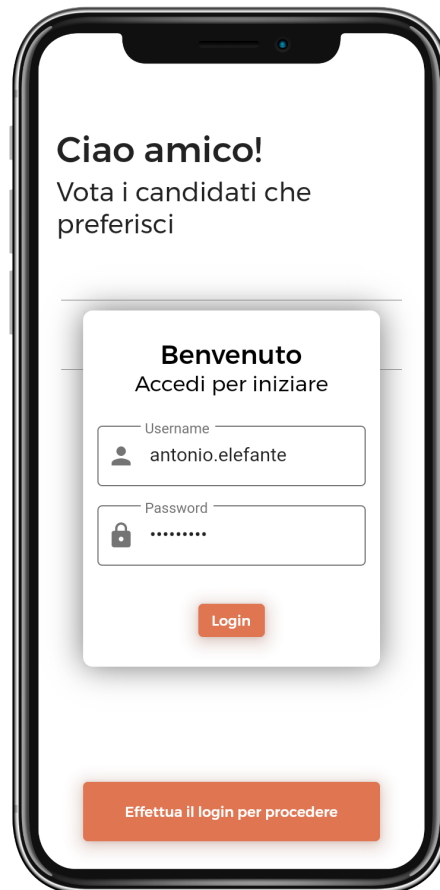


Figura 4.2: Accesso utente Admin



Figura 4.3: Creazione nuovo candidato

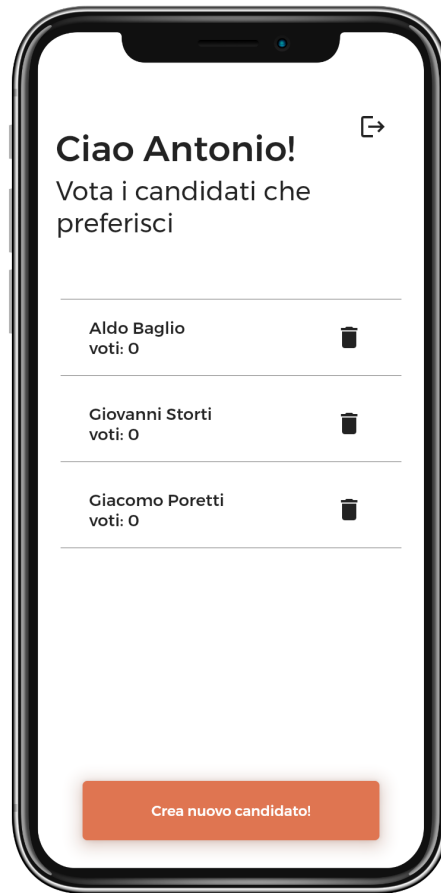


Figura 4.4: Lista candidati in visualizzazione Amministratore

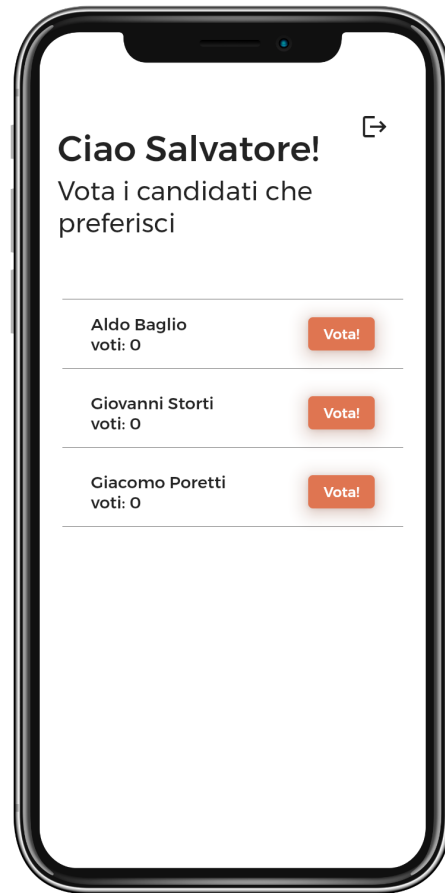


Figura 4.5: Lista candidati in visualizzazione Utente

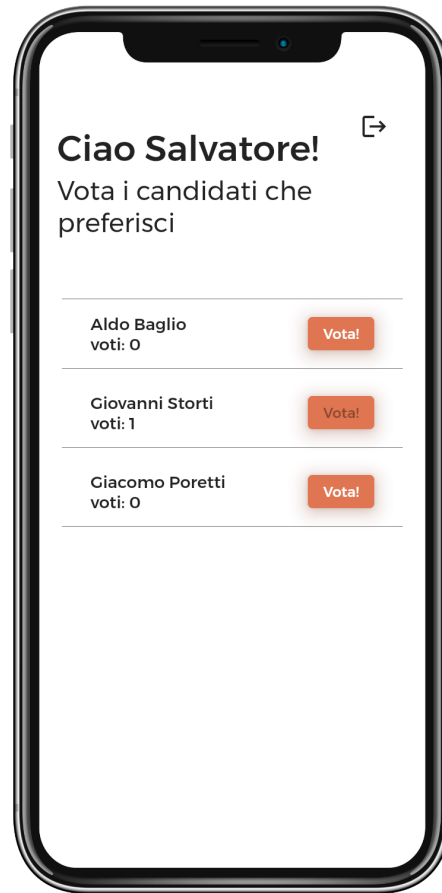


Figura 4.6: Votazione candidato Giovanni Storti

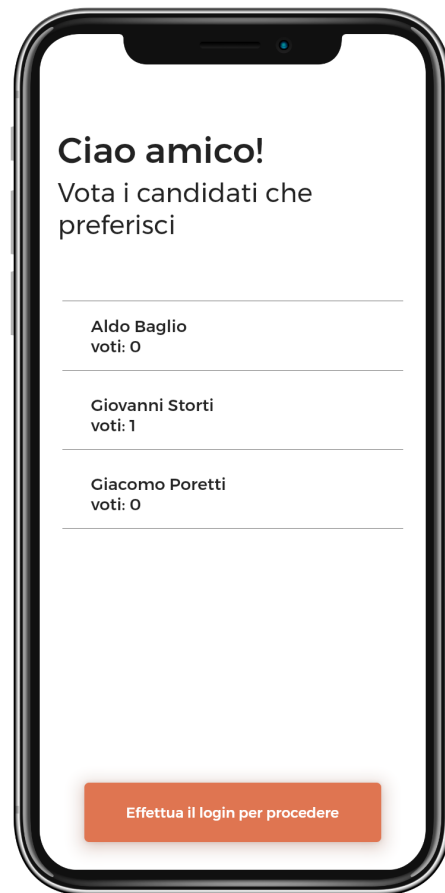


Figura 4.7: Lista candidati in visualizzazione Anonima

### 4.1.3 Come testare il prodotto finale

Una volta effettuato l'accesso al file zip associato a tale documentazione, comunque disponibile in maniera pubblica al seguente link github TODO: inserire link, è possibile tramite docker avviare i 3 differenti server:

- Server NodeJS (API)
- Server Keycloak (Authentication + Authorization)
- Server MongoDB (Persistenza dei dati)

Per far ciò basterà recarsi tramite terminale nella cartella del progetto ed eseguire il comando:

```
docker-compose up
```

Il quale si occuperà di scaricare (ed eseguire) tutti i container necessari per testare il progetto finale. Una volta fatto ciò basterà quindi recarsi sul TODO: GitHub Pages (preferibilmente da Desktop/Tablet) per accedere all'interfaccia front-end e quindi interagire con il sistema ricordando che:

Username	Password	Role
antonio.elefante	123456789	Amministratore
salvatore.ambrosio	Password123!	User

Tabella 4.1: Credenziali utente con relativo ruolo

Inoltre è predisposta la possibilità di osservare in maniera anonima la lista dei candidati senza ovviamente però poter procedere con la votazione.



# Elenco delle figure

1.1	Esempio token JWT . . . . .	6
1.2	Flusso di autorizzazione utente mediante richiesta token . . . . .	6
1.3	JWT vs SAML . . . . .	8
1.4	Docker Logo . . . . .	9
1.5	Architettura Docker . . . . .	9
2.1	Home page Keycloak . . . . .	11
2.2	Pannello di amministrazione del server Keycloak . . . . .	12
2.3	Creazione nuovo Reame in Keycloak 1/2 . . . . .	13
2.4	Creazione nuovo Reame in Keycloak 2/2 . . . . .	13
2.5	Sezione Utenti . . . . .	14
2.6	Creazione di un nuovo utente . . . . .	14
2.7	Differenti azioni richiedibili all'utente . . . . .	15
2.8	Lista sezioni utente salvatore.ambrosio . . . . .	15
2.9	Impostazione nuova password per l'utente salvatore.ambrosio 1/2 . . . . .	16
2.10	Impostazione nuova password per l'utente salvatore.ambrosio 2/2 . . . . .	17
2.11	Azione richiesta all'utente poichè password temporanea . . . . .	18
2.12	Dashboard creazione client . . . . .	19
2.13	Creazione client-role . . . . .	20
2.14	Lista ruoli per client d'esempio . . . . .	20
2.15	Role composition, associazione realm role app-administrator al client role admin . . . . .	21
2.16	Ruoli associati all'utente antonio.elefante . . . . .	22
2.17	Esempio token JWT ottenuto dall'autenticazione tramite Keycloak . . . . .	24
3.1	NodeJS logo . . . . .	25
3.2	MongoDB logo . . . . .	33
4.1	Flutter logo . . . . .	39
4.2	Accesso utente Admin . . . . .	42
4.3	Creazione nuovo candidato . . . . .	43
4.4	Lista candidati in visualizzazione Amministratore . . . . .	44
4.5	Lista candidati in visualizzazione Utente . . . . .	45

4.6	Votazione candidato Giovanni Storti . . . . .	46
4.7	Lista candidati in visualizzazione Anonima . . . . .	47

# Elenco delle tabelle

2.1	Coppie Key-Value necessari per l'ottenimento del token JWT . . . . .	23
3.1	Candidato Schema . . . . .	35
4.1	Credenziali utente con relativo ruolo . . . . .	48