UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

SVILUPPO DI UNA WEB APP DEDICATA ALL'ANALISI DI SICUREZZA INFORMATICA

Relatore: Prof. Mauro Migliardi

Laureando: Daniele Zebele

ANNO ACCADEMICO 2021 – 2022

Data di laurea: 20 luglio 2022

Indice:

Indice ……..... 1
Informazioni generali ……... 2
Introduzione …… 2
La struttura ospitante ……..... 2
Scopo del tirocinio …… 2
Dettagli del software Nmap …….. 3
Scopo dell'applicazione ……... 4
Cos'è Zenmap e perché non è adatta allo scopo dell'azienda …….. 4
Requisiti dell'applicazione ……... 5
Struttura dell'applicazione ……... 6
Dettagli sul database utilizzato ……... 10
Gestione del database ……. 10
Dettagli sulla gestione degli utenti …….. 11
Dettagli sulla gestione delle scansioni ……..... 11
Dettagli sul framework javascript utilizzato ……...… 13
Struttura e descrizione del codice …….. 14
Librerie utilizzate ……... 25
Considerazioni finali e possibili miglioramenti ……. 25

Informazioni generali:

Nome e Cognome del tirocinante: Daniele Zebele

Matricola: 1229056

Struttura ospitante: Azienda CyBrain srl

Tutor della struttura ospitante: Francesco Piga

Periodo di svolgimento: Dal 14 febbraio 2022 al 22 aprile 2022

Introduzione:

Il tirocinio è stato svolto presso l'azienda CyBrain srl di Monteviale. Durante le 10 settimane di tirocinio, è stata realizzata un'applicazione web nell'ambito di un progetto dedicato alla sicurezza informatica.

La struttura ospitante:

CyBrain s.r.l. si occupa di consulenza e sviluppo nell'ambito della sicurezza informatica per grandi gruppi bancari, società di telecomunicazioni ed altre realtà Enterprise di importanza nazionale. Le soluzioni offerte vengono progettate e sviluppate in modo autonomo e sono rivolte sia ad utenti tecnici che ad utenti con un minor background tecnologico e con stampo più manageriale.

Scopo del tirocinio:

Realizzare una web app per interfacciare un software da integrare in un progetto realizzato in azienda, implementando nuove funzionalità in maniera modulare, tramite tecnologie innovative e maggiormente performanti.

L'applicazione oggetto sarà tipicamente un portale che include sia front-end che back-end.

Il progetto verrà realizzato con framework opensource disponibile su mercato scelto in base alle caratteristiche dell'applicazione.

Per la realizzazione del progetto si terrà conto di problematiche di cybersecurity, quindi, verranno adottate alcune modalità di sviluppo sicuro per renderlo meno vulnerabile ad attacchi.

Scopo dell'applicazione:

In ambito aziendale, l'applicazione realizzata è necessaria per comporre l'inventario di tutti i dispositivi presenti su una rete e controllare se esistono degli indirizzi sui quali non è installato un proprio agent.

CyBrain ha infatti installato un agent su ciascun dispositivo (server, client, apparati, ecc.) che serve ad inoltrare alcuni log di accesso ad un proprio server, in ottemperanza alle disposizioni di legge (GDPR). Per sapere se nel frattempo nella rete del cliente vengono aggiunte o tolte macchine viene effettuata la scansione con Nmap per ottenere l'inventario.

Questo metodo viene applicato a reti complesse di grossi clienti, che spesso non possono essere in grado di raccogliere tempestivamente informazioni su nuove installazioni di PC.

L'applicazione, inoltre, rende più semplice l'utilizzo di Nmap. migliora la leggibilità dell'output e permette di salvare automaticamente i dati in un database per poterli analizzare in futuro.

Il software Nmap produce come output un file xml e le informazioni sono racchiuse nei tag e nei rispettivi attributi, dopo la realizzazione dell'app invece il contenuto viene esposto sotto forma di tabella, e può essere facilmente navigabile attraverso l'aiuto di filtri e paginazione.

Dettagli del software Nmap:

Il software su cui è stata effettuata l'integrazione è Nmap. un software libero, distribuito con licenza GNU GPL da Insecure.org creato per effettuare port scanning, cioè mirato all'individuazione di porte aperte su un computer bersaglio o anche su range di indirizzi IP, in modo da determinare quali servizi di rete siano disponibili.

Come molti strumenti usati nel campo della sicurezza informatica, può essere utilizzato sia dagli amministratori di sistema che dai cracker o script kiddies, divenendo uno degli strumenti praticamente indispensabili nella "cassetta degli attrezzi" di un sistemista, usato per test di penetrazione e compiti di sicurezza informatica in generale.

Nmap («Network Mapper») è uno strumento opensource per la network exploration e l'auditing. È stato progettato per scansionare rapidamente reti di grandi dimensioni, ma è indicato anche per l'utilizzo verso singoli host. Nmap usa pacchetti IP "raw" (grezzi, non formattati) in varie modalità per determinare quali host sono disponibili su una rete, che servizi (nome dell'applicazione e versione) vengono offerti da questi host, che sistema operativo (e che versione del sistema operativo) è in esecuzione, che tipo di firewall e packet filters sono usati, e molte altre caratteristiche. Nonostante Nmap sia comunemente usato per audits di sicurezza, molti sistemisti e amministratori di rete lo trovano utile per tutte le attività giornaliere, come ad esempio l'inventario delle macchine presenti in rete, per gestire gli aggiornamenti programmati dei servizi e per monitorare gli host o il loro uptime.

L'output di Nmap è un elenco di obiettivi scansionati, con informazioni supplementari per ognuno a seconda delle opzioni usate. Tra queste informazioni è vitale la «tabella delle porte interessanti».

Questa tabella elenca il numero della porta e il protocollo, il nome del servizio e lo stato attuale. Lo stato può essere open (aperto), filtered (filtrato), closed (chiuso), o unfiltered (non filtrato). Aperto significa che vi è sulla macchina obiettivo un'applicazione in ascolto su quella porta per connessioni o pacchetti in entrata. Filtrato significa che un firewall, un filtro o qualche altro ostacolo di rete sta bloccando la porta al punto che Nmap non riesce a distinguere tra aperta o chiusa. Le porte chiuse non hanno alcuna applicazione in ascolto, anche se potrebbero aprirsi in ogni momento. Le porte vengono classificate come non filtrate quando rispondono ad una scansione di Nmap. ma non è stato possibile determinare se sono aperte o chiuse. Nmap mostra le combinazioni aperta|filtrata e chiusa|filtrata quando non può determinare quale dei due stati descrive una porta. La tabella delle porte può anche includere dettagli quali le versioni dei software disponibili se è stata usata l'opzione appropriata. Quando viene richiesta una scansione IP (-sO), Nmap fornisce informazioni sui protocolli IP supportati anziché sulle porte in ascolto.

In aggiunta alla tabella delle porte notevoli, Nmap può fornire ulteriori informazioni sugli obiettivi come, ad esempio, i nomi DNS risolti (reverse DNS names), il probabile sistema operativo in uso, il tipo di device e l'indirizzo fisico (MAC address).

Una tipica scansione con Nmap:

```
# nmap -A -T4 scanme.nmap.org

nmap scan report for scanme.nmap.org (74.207.244.221)
Host is up (0.029s latency).
rDNS record for 74.207.244.221: li86-221.members.linode.com
Not shown: 995 closed ports
PORT     STATE    SERVICE     VERSION
22/tcp   open     ssh         OpenSSH 5.3p1 Debian 3ubuntu7 (protocol 2.0)
| ssh-hostkey: 1024 8d:60:f1:7c:ca:b7:3d:0a:d6:67:54:9d:69:d9:b9:dd (DSA)
|_2048 79:f8:09:ac:d4:e2:32:42:10:49:d3:bd:20:82:85:ec (RSA)
80/tcp   open     http        Apache httpd 2.2.14 ((Ubuntu))
|_http-title: Go ahead and ScanMe!
646/tcp  filtered ldp
1720/tcp filtered H.323/Q.931
9929/tcp open     nping-echo  Nping echo
Device type: general purpose
Running: Linux 2.6.X
OS CPE: cpe:/o:linux:linux_kernel:2.6.39
OS details: Linux 2.6.39
Network Distance: 11 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:kernel

TRACEROUTE (using port 53/tcp)
HOP RTT      ADDRESS
[Cut first 10 hops for brevity]
11  17.65 ms li86-221.members.linode.com (74.207.244.221)

nmap done: 1 IP address (1 host up) scanned in 14.40 seconds
```

Le uniche opzioni usate di Nmap in questo esempio sono -A, per abilitare la rilevazione del sistema operativo e della versione, lo script scanning e il traceroute, -T4 per un'esecuzione più rapida e infine l'host obiettivo.

(fonte: https://nmap.org/man/it/index.html#man-description)

Cos’è Zenmap e perché non è adatta allo scopo dell’azienda:

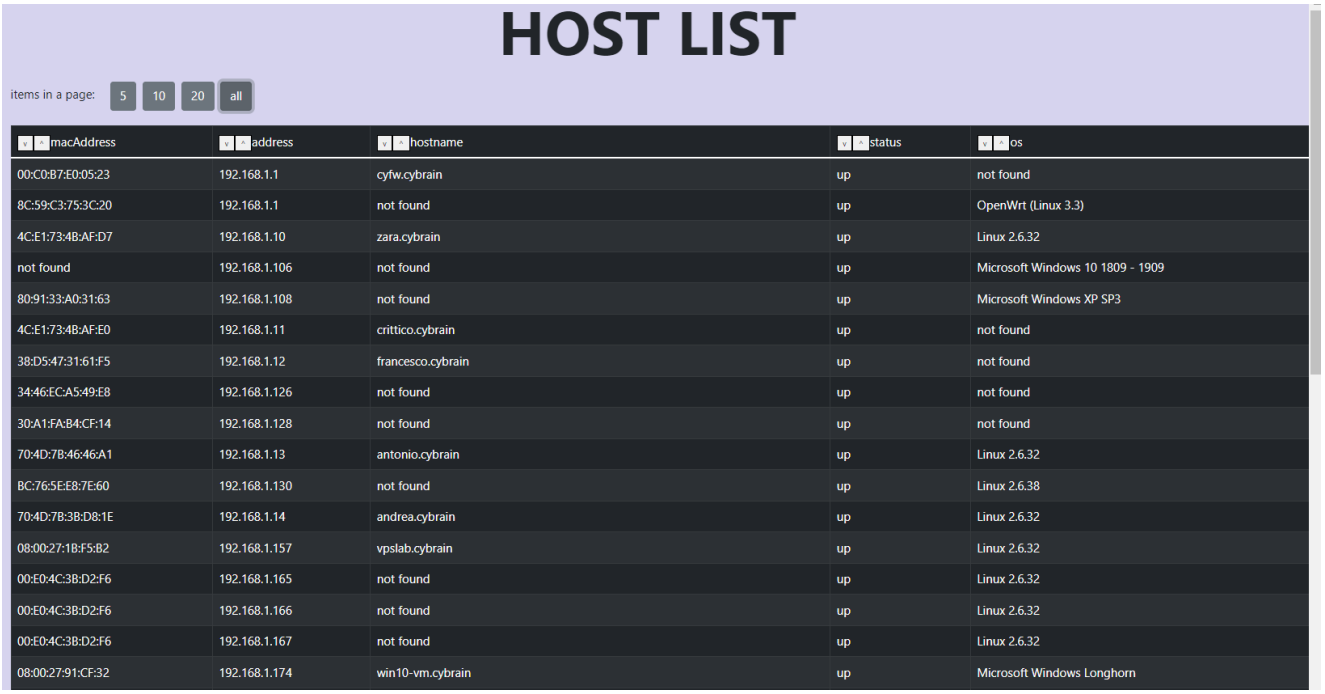
Zenmap è la GUI ufficiale di Nmap Security Scanner. È un'applicazione multipiattaforma (Linux, Windows, Mac OS X, BSD, ecc.) gratuita e opensource che mira a rendere Nmap facile da usare per i principianti fornendo funzionalità avanzate per utenti Nmap esperti. I risultati delle scansioni possono essere archiviati, visualizzati in un secondo momento e possono essere confrontati tra loro per vedere come differiscono.

La WebApp realizzata, a differenza di Zenmap, nel corso del tempo mira a produrre un inventario di hosts rilevati nelle varie scansioni, rendendo quindi facile ottenere una lista di dispositivi che hanno effettuato l’accesso alla rete locale.

L’applicazione inoltre permette la creazione di diversi profili utente che possono accedere ai dati delle scansioni e viene realizzata con framework, strumenti, metodologie e strumenti di sicurezza compatibili con il progetto con il quale dovrà essere integrata.

Questo rende la WebApp più adatta rispetto a Zenmap in ambito aziendale.

L’inventario prodotto dall’applicazione ottenuto dalle scansioni:



HOST LIST

items in a page: 5 10 20 all

macAddress	address	hostname	status	os
00:C0:B7:E0:05:23	192.168.1.1	cyfw.cybrain	up	not found
8C:59:C3:75:3C:20	192.168.1.1	not found	up	OpenWrt (Linux 3.3)
4C:E1:73:4B:AF:D7	192.168.1.10	zara.cybrain	up	Linux 2.6.32
not found	192.168.1.106	not found	up	Microsoft Windows 10 1809 - 1909
80:91:33:A0:31:63	192.168.1.108	not found	up	Microsoft Windows XP SP3
4C:E1:73:4B:AF:E0	192.168.1.11	crittico.cybrain	up	not found
38:D5:47:31:61:F5	192.168.1.12	francesco.cybrain	up	not found
34:46:EC:A5:49:E8	192.168.1.126	not found	up	not found
30:A1:FA:B4:CF:14	192.168.1.128	not found	up	not found
70:4D:7B:46:46:A1	192.168.1.13	antonio.cybrain	up	Linux 2.6.32
BC:76:5E:E8:7E:60	192.168.1.130	not found	up	Linux 2.6.38
70:4D:7B:3B:D8:1E	192.168.1.14	andrea.cybrain	up	Linux 2.6.32
08:00:27:1B:F5:B2	192.168.1.157	vpslab.cybrain	up	Linux 2.6.32
00:E0:4C:3B:D2:F6	192.168.1.165	not found	up	Linux 2.6.32
00:E0:4C:3B:D2:F6	192.168.1.166	not found	up	Linux 2.6.32
00:E0:4C:3B:D2:F6	192.168.1.167	not found	up	Linux 2.6.32
08:00:27:91:CF:32	192.168.1.174	win10-vm.cybrain	up	Microsoft Windows Longhorn

Requisiti dell’applicazione:

- La WebApp deve esporre sotto forma di tabella le informazioni raccolte da Nmap. originariamente prodotte in formato xml e deve permettere la navigazione attraverso l’uso di filtri;
- l’accesso all’applicazione deve essere condizionato da un processo di autenticazione;
- sono previsti diversi profili di autenticazione che permettono di vedere contenuti differenti;
- la WebApp deve permettere la gestione delle utenze e i rispettivi profili di autorizzazione;
- solo l’utente admin può gestire gli utenti;

- la WebApp deve gestire le sessioni degli utenti per impedire l'accesso ai contenuti da parte di utenti non verificati e per evitare l'accesso da parte di utenti senza privilegi alle pagine riservate all'admin;
- ogni password salvata nel database deve venire criptata;
- devono essere presenti dei meccanismi per il controllo degli input, in linea con i requisiti di sviluppo sicuro;
- la scansione di Nmap deve funzionare in background in modo indipendente dal funzionamento dell'applicazione web.

Struttura dell'applicazione:

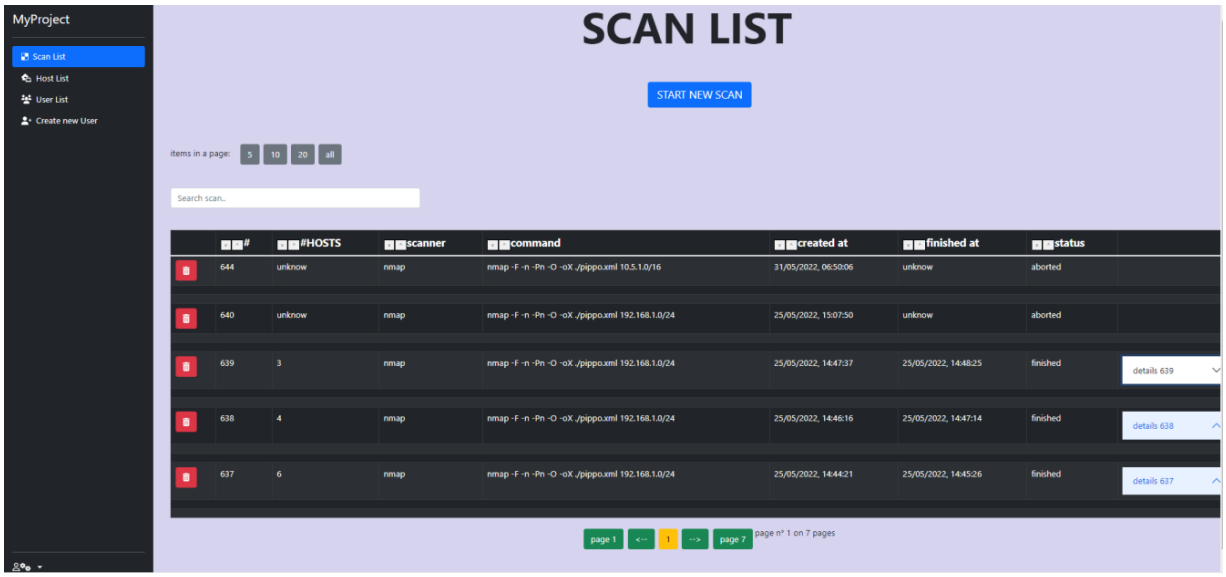
L'applicazione prevede, in un ipotetico utilizzo aziendale, la creazione di utenti che possono accedere ai dati del database in base al rispettivo privilegio.

Gli utenti senza privilegi hanno solamente la possibilità di visualizzare i dati rispettivi alle scansioni, mentre l'utente admin può lanciarne di nuove, visualizzare dati aggiuntivi (porte aperte, sistema operativo e stato degli hosts), creare nuovi utenti, eliminarli o resettare le password.

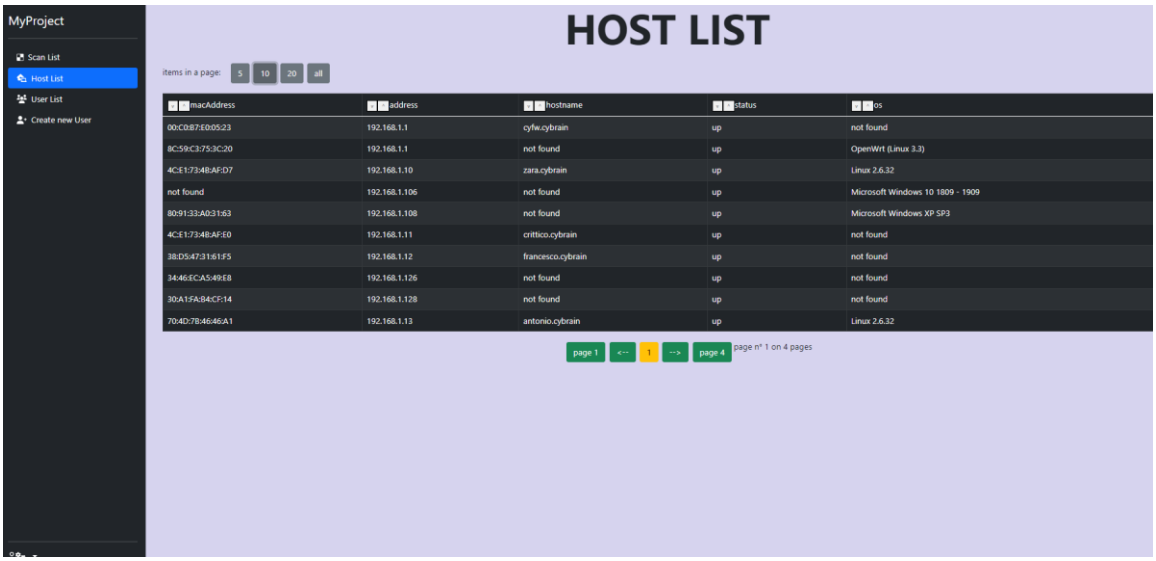
Al primo avvio viene creato in automatico l'utente admin con password admin.

Le pagine visitabili sono presentate in una barra di navigazione a sinistra dello schermo e sono:

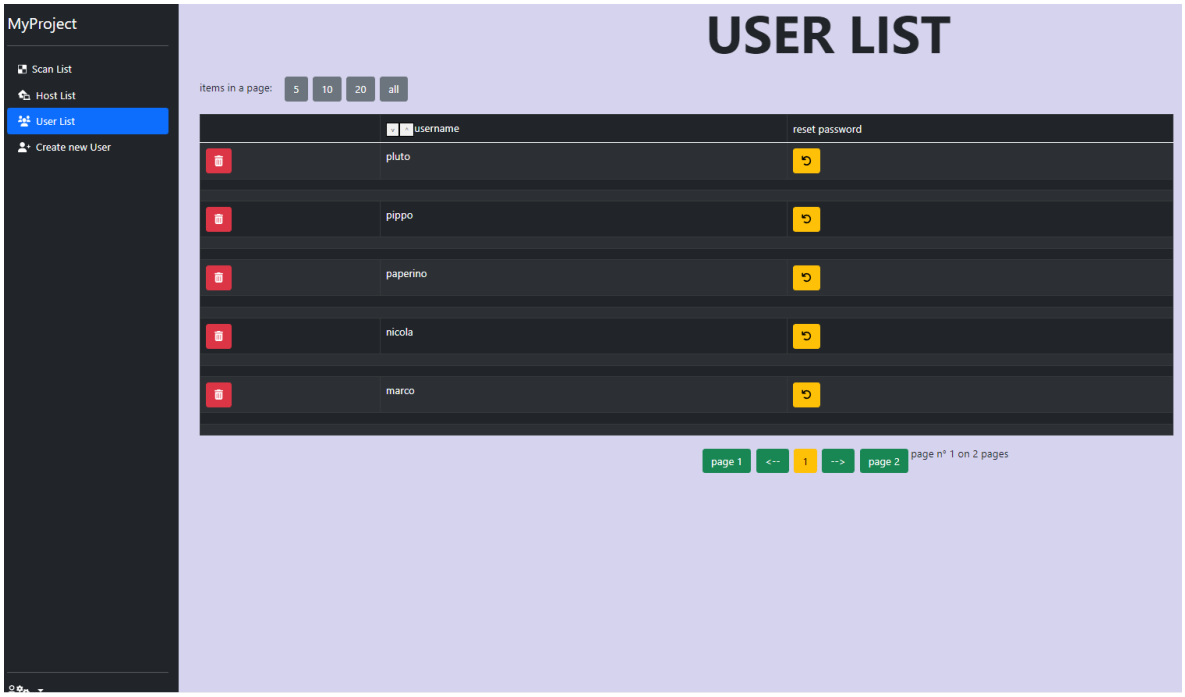
- Scan list: pagina che permette di visualizzare i dettagli di ogni scansione e, solo per l'utente admin un bottone che permette la richiesta di avvio di una nuova scansione;



- Host list: pagina che permette di visualizzare l'inventario di tutti gli hosts rilevati in tutte le scansioni lanciate;

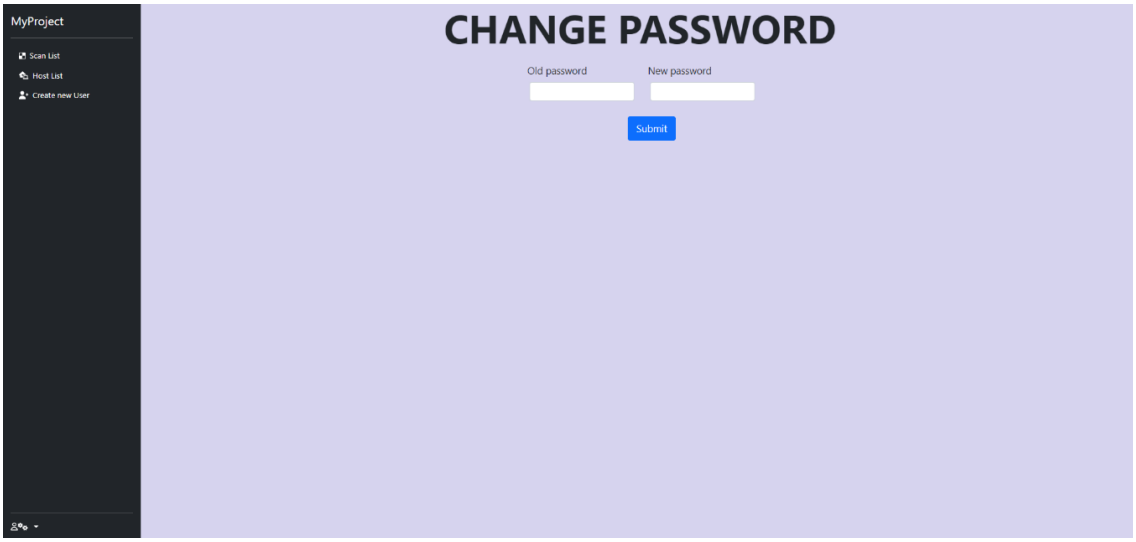


- User list: pagina che permette all'admin di visualizzare tutti gli utenti per poterli eliminare o effettuare un reset della password;

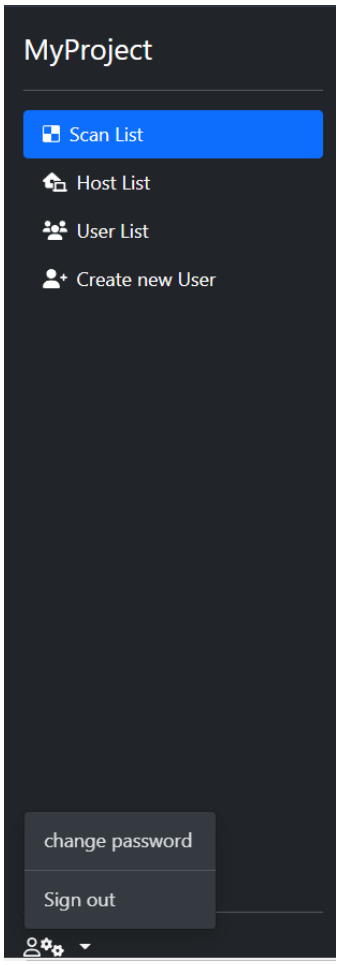


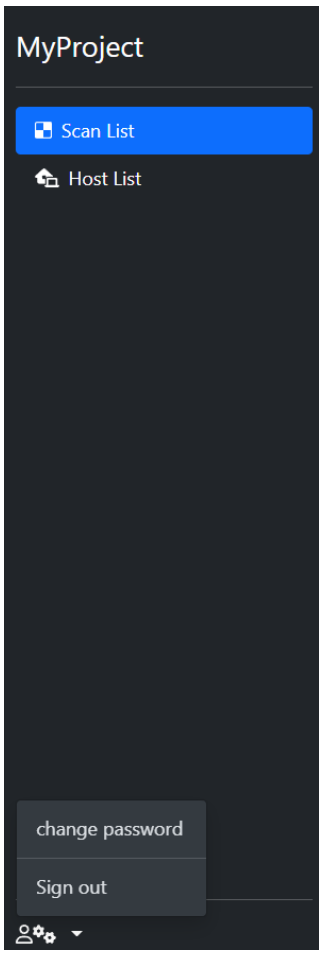
- Create new user: pagina che permette all’admin di creare nuovi utenti.

- Change Password (si accede premendo in basso a sinistra dove è presente il bottone impostazioni).

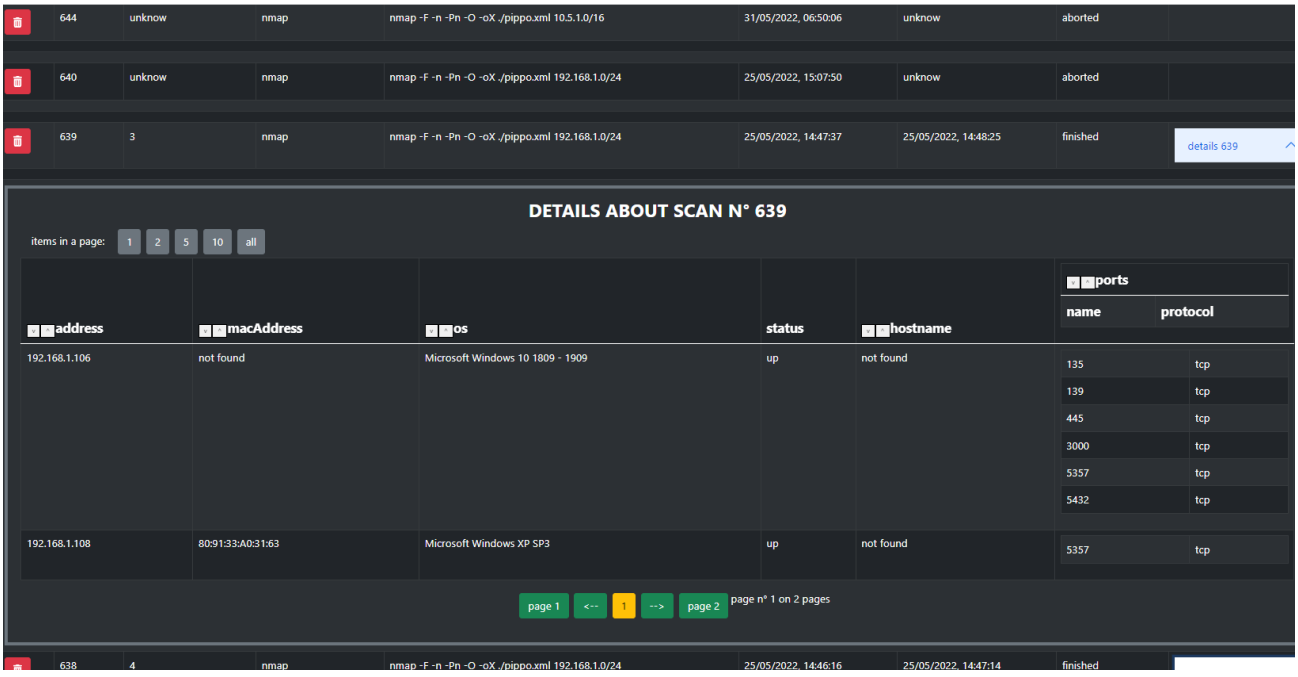


A sinistra, la barra di navigazione dell'utente admin, a destra di un utente normale.

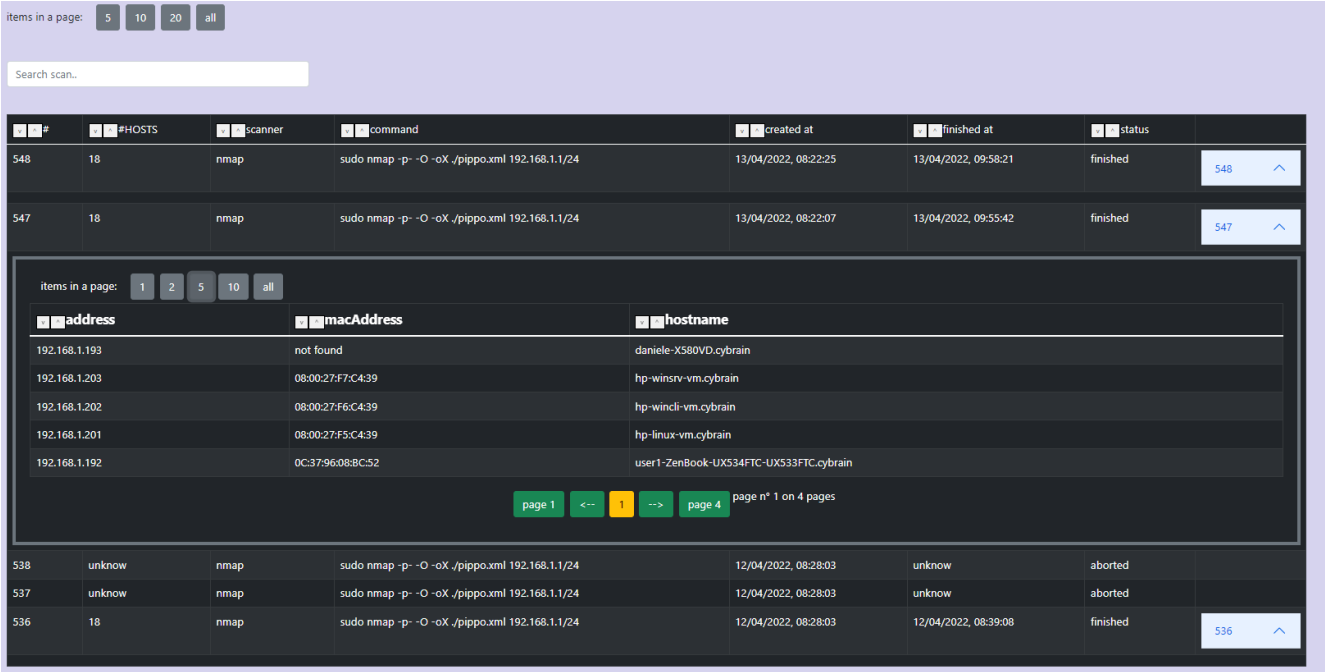




Visualizzazione dei dettagli di una scansione da parte dell'admin:



Visualizzazione dei dettagli di una scansione di un utente comune:



Dettagli sul database utilizzato:

Per la realizzazione dell'applicazione è stato usato un database MongoDB.

Mongo è un database NoSQL: a prima vista può sembrare che sia un antagonista del classico e canonico SQL; in realtà, NoSQL significa Not only SQL, che invece sta ad indicare un ampliamento dello stesso con nuove funzionalità e nuove prospettive di sviluppi applicativi, a partire dalla eliminazione delle Relazioni.

MongoDB è un database opensource con la particolarità principale di gestire i dati in Documenti (Documents), rappresentati tramite BSON (simile al JSON) e organizzati in Collezioni (Collections).

A riguardo, è importante ricordare che in questo caso con il termine Documento si intende un'entità generica che porta con sé informazioni e dati.

La scelta di un database NoSQL per l'applicazione dipende dal fatto che il progetto già realizzato dall'azienda prevede un flusso continuo importante di dati in arrivo (log di diversi sistemi e di diversi formati) che necessitano di essere scritti con velocità, e inoltre alcune collezioni contengono dati con strutture dinamiche per le quali i database non relazionali si prestano intrinsecamente meglio.

Date queste premesse, poiché la letteratura suggerisce performance migliori in scrittura dei database NoSQL rispetto ai SQL in termini di lettura/scrittura se le tabelle non raggiungono dimensioni molto significative, abbiamo scelto di usare MongoDB con collezioni capped (collezioni con funzionamento simile ai buffer circolari) per quelle che prevedevano molti dati.

Prima di adottare questo strumento, l'azienda ha fatto anche delle prove sul campo tra MongoDB e MySQL a parità di risorse, con l'invio massivo di dati. MongoDB ha avuto delle performances decisamente migliori.

Gestione del database:

Le collections su cui si basa il salvataggio dei dati sono:

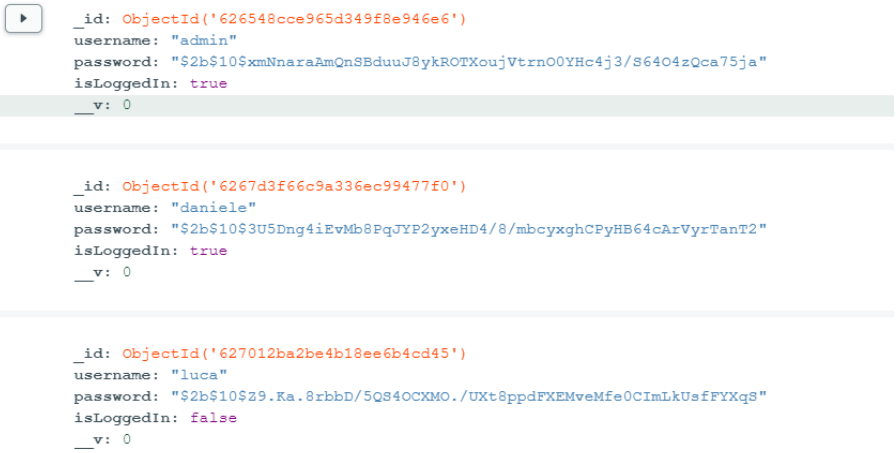
- Users: qui vengono salvati i dati rispettivi agli utenti, in particolare: username, password e un flag di controllo che verifica ulteriormente se l'utente ha effettuato il login;
- Scans: qui vengono salvati i dati relativi alle scansioni, in particolare: stato, comando utilizzato (per una possibile compatibilità nel caso si utilizzino altri tool di Nmap), un contatore, il numero di hosts rilevati, data di inizio e fine scansione;
- Hosts: qui vengono salvati gli hosts rilevati dalle scansioni, in particolare: addres, mac address, le porte rilevate, lo stato del dispositivo, sistema operativo e l'id della scansione che li ha rilevati (unica relazione all'interno del database);
- Counter: un semplice contatore che incrementa ad ogni scansione avviata, la sua presenza velocizza la creazione di nuove scansioni poiché non è necessario effettuare una ricerca per identificare il nuovo indice da assegnare;
- AgendaJobs: una collection creata e gestita interamente dalla libreria agenda, utilizzata per verificare lo stato delle scansioni e avviare l'esecuzione di Nmap.

Dettagli sulla gestione degli utenti:

Ogni volta che viene creato un utente dall'admin, per semplicità operativa in ambiente di sviluppo, la password di default è lo stesso username; successivamente però gli utenti potranno modificarla.

Ogni password prima di essere salvata nel database viene criptata, per garantire maggiore sicurezza interna.

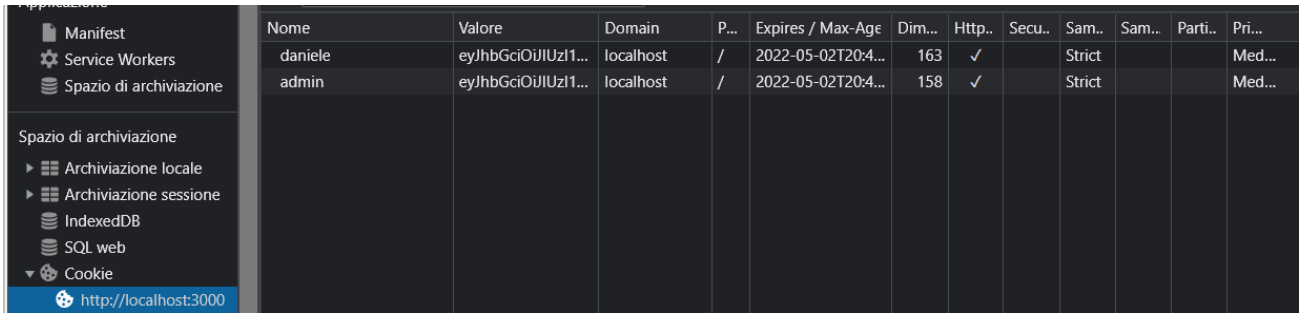
Come vengono salvati gli utenti nella collection "Users" del database:



Attraverso l'utilizzo di cookies vengono controllate le sessioni degli utenti, imponendo a chiunque voglia accedere a ciascuna pagina di aver eseguito il login in precedenza. Ciò previene, ad esempio, l'utilizzo di pagine riservate all'admin da parte di utenti normali o l'utilizzo di pagine da utenti non loggati.

Ogni cookie viene creato al momento del login e cancellato al sign out, o si autodistrugge allo scadere di un intervallo di tempo predefinito.

Esempio dei cookies presenti dopo aver eseguito il login due volte con utenti diversi (il valore dei cookies iniziano con le stesse lettere ma sono diversi; notare la dimensione):



Dettagli sulla gestione delle scansioni:

Ogni scansione viene creata nel momento in cui l'admin ne fa una richiesta nella pagina scan list.

Nel momento in cui viene richiesta, la scansione viene salvata in un document (in coda ad altre richieste, se presenti) nel database, con stato: "starting…". Oltre allo stato viene memorizzato il comando Nmap utilizzato, il file di output temporaneo, la data (timestamp) di creazione e di terminazione della scansione, il numero di hosts rilevati e un numero progressivo. Inizialmente la data di termine e gli hosts vengono posti a un valore nullo, verranno aggiornati in seguito. La scelta di inserire anche il comando utilizzato deriva dal fatto che l'azienda in futuro vuole poter utilizzare dei comandi Nmap diversi o altri software di scansione di rete.

Successivamente lo script parallelo ''script_agenda/scheduler.js'', a ogni periodo di tempo determinato (per ragioni di testing durante lo sviluppo impostato a 1 minuto), ricerca una scansione nel database in stato di ''starting…''. Se non la trova, aspetta un altro minuto, altrimenti esegue il comando Nmap salvato nel document rispettivo, aggiornando lo stato in: ''scanning...''. Lo script può eseguire solo una scansione alla volta, quindi se una scansione dura più di un minuto, non ne viene lanciata una nuova.

Una volta terminata la scansione viene creato da Nmap il file di output in linguaggio xml, lo script effettua un parsing dell'output in json e crea nel database ogni istanza di host rilevata. Prima di ciò lo stato viene aggiornato in: "parsing…''. Viene inoltre eliminato il file di output di Nmap e riempiti i campi lasciati nulli in precedenza come il numero di hosts rilevati e la data (timestamp) di fine scansione.

Una volta terminato il parsing lo stato viene impostato a "finished".

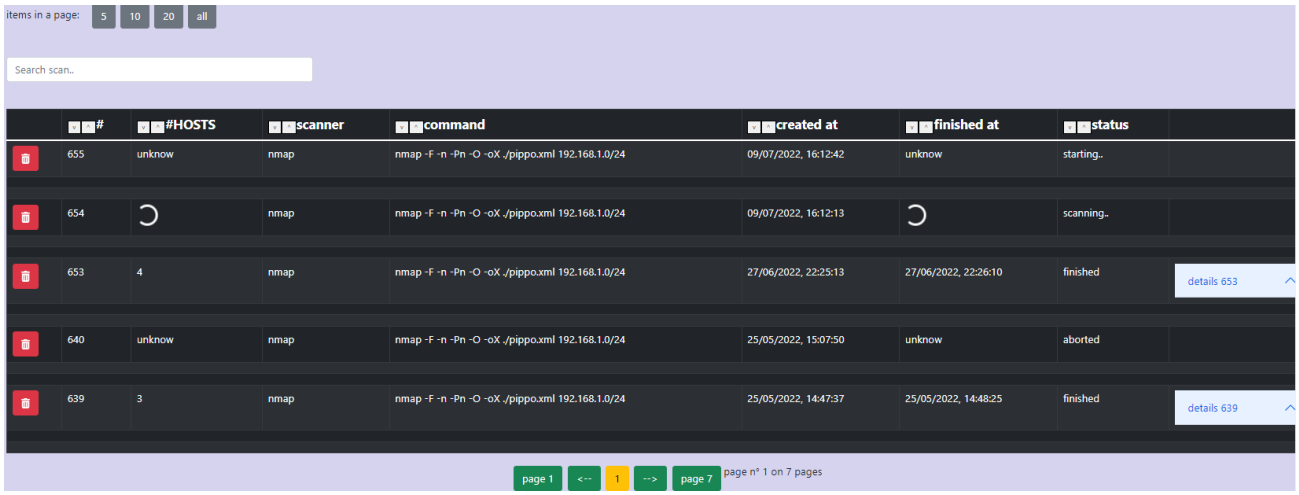
Se lo script viene interrotto o per qualsiasi altro problema, per semplicità operativa in ambiente di sviluppo, all'avvio successivo la scansione viene abortita e lo stato viene impostato ad "aborted". I campi lasciati nulli rimangono tali e sarà compito del codice front-end gestirli.

Come accennato in precedenza, lo script scheduler, soddisfa una richiesta di scansione alla volta; quindi, se delle richieste vengono inviate durante una scansione o durante il parsing, esse rimarranno in coda in stato di "starting…".

I file javascript prodotti per realizzare questo meccanismo sono:

- la pagina front-end scan list (pages/protectedPages/usersDashboard/admin.js) che semplicemente fa la richiesta di aggiungere in coda una nuova scansione nel momento in cui l'admin preme il bottone "start new scan";
- la pagina back-end scanAndStore (pages/api/scan/scanAndStore.js) che riceve la richiesta, la elabora e la inserisce nel database in stato di "starting…";
- lo script parallelo (script_agenda/scheduler.js) descritto in precedenza.

Esempio di situazione con diversi stati per le scansioni:



Dettagli sul framework javascript utilizzato:

Il framework utilizzato per la realizzazione della webApp è Next.js

Next.js è un framework JavaScript back-end per applicazioni React, e consente il rendering automatico lato server (SSR, server side rendering).

Con Next.js si possono sviluppare applicazioni web, app mobile, desktop e web app progressive: è costruito secondo il principio di "Build once, run anywhere".

Si tratta di un framework che non richiede alcun setup, utilizza il filesystem come un'API. Altre caratteristiche di Next.js sono: routing delle pagine automatico (che verrà spiegato meglio successivamente), hot code reloading (che permette di ricaricare solo il codice modificato) ed esportazione statica (che con un solo comando può esportare un sito statico).

Vantaggi rispetto ad altri framework:

- server-side rendering: le pagine vengono renderizzate lato server e non lato client, ciò permette di ridurre il tempo di attesa prima di visualizzare una pagina;
- file-based routing: il routing delle pagine è automatico. Viene definito attraverso l'utilizzo dei file e delle cartelle e non a livello di codice. Ad esempio, se nella cartella pages viene creata una sottocartella users e dentro a quest'ultima il file admin.js, la pagina admin verrà visualizzata in automatico accedendo al percorso /users/admin;
- fullstack Capabilities: si può inserire facilmente codice back-end nel progetto all'interno della cartella pages/api.

In generale, le pagine web dell'applicazione prelevano i dati comunicando con gli script della cartella pages/api attraverso request e response (in formato JSON). Gli script a loro volta riescono a fornire i dati comunicando con il database.

Un progetto NextJs si può creare dopo l'installazione di Node.Js con il comando "npx create-next-app" e può essere avviato con il comando "npm run dev". Si può visualizzare l'output dell'esecuzione accedendo in locale alla pagina "localhost:3000/".

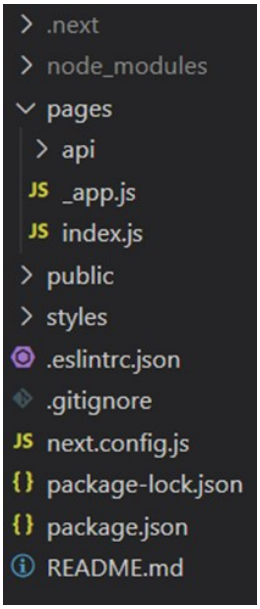
NextJs si appoggia a npm, abbreviazione di Node Package Manager, è il gestore di pacchetti ufficiale che viene installato con la piattaforma Node.js. Si tratta quindi di un'interfaccia a riga di comando (CLI) che aiuta nell'installazione dei pacchetti, nella gestione delle versioni e delle dipendenze.

Npm fa anche riferimento al registro, un grande database pubblico di applicazioni JavaScript, e al sito web che consente di scoprire pacchetti, impostare profili e gestire altri aspetti dell'esperienza npm.

Esiste un grande numero di librerie e applicazioni Node.js pubblicate su npmjs.com: i pacchetti possono essere pubblici o privati, in base alle impostazioni di accesso scelte dall'autore del pacchetto.

Per maggiori informazioni: https://nextjs.org/docs.

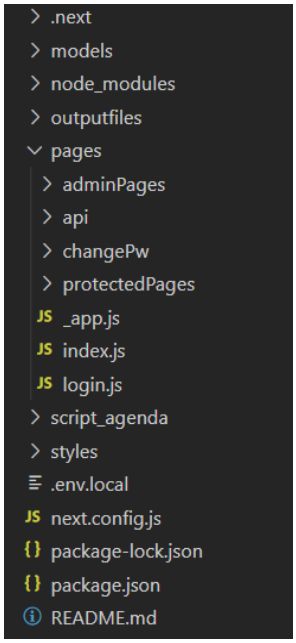
Struttura e descrizione del codice:



L'immagine a sinistra rappresenta l'aspetto di un progetto nextJs appena creato dopo il comando npx create-next-app.

Il progetto iniziale contiene:

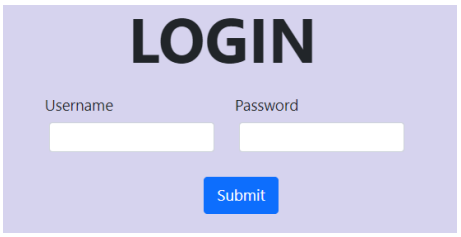
- la cartella .next: viene creata in automatico dopo il primo avvio dell'applicazione e contiene dei file utili all'esecuzione dell'app;
- la cartella node_modules: contiene tutte le librerie e pacchetti utilizzati nel progetto;
- pages, public e styles: sono cartelle da riempire per personalizzare l'applicazione. Nella cartella pages sono già presenti i file index.js , _app.js e la cartella api. Index all'inizio contiene una pagina generica creata da Next, _app invece contiene tutte le informazioni comuni alle pagine web. Di solito in questo file vengono inseriti gli import per le librerie utilizzate e i collegamenti ai fogli di stile (css). Anche la cartella api inizialmente contiene un file generico.
- next.config: è un file che contiene le impostazioni personalizzate di next, mentre i packages.json servono a npm per gestire le dipendenze tra i pacchetti installati.

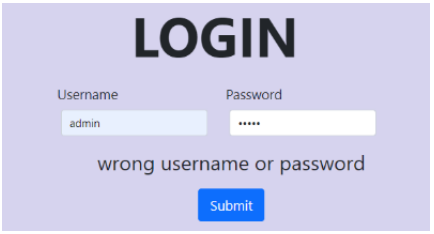


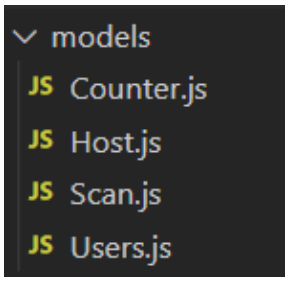
L'immagine a sinistra invece rappresenta lo stato del progetto attuale, ed in particolare sono stati aggiunti:

- la cartella models, descritta successivamente;
- la cartella outputfiles senza un utilizzo pratico ma utile in fase di progettazione per tenere qualche output di Nmap;
- le cartelle adminPages, api, changePw e protectedPages, descritte successivamente;
- la cartella script_agenda, contenente lo scheduler descritto in precedenza;
- .env.local: file contenente delle variabili d'ambiente;
- il file index.js: contiene il codice front-end della pagina ottenuta accedendo a "localhost:3000/". Il codice all'interno si occupa solamente di verificare attraverso una richiesta all'endpoint "api/isFirstTime" (descritta successivamente) se l'applicazione è stata avviata per la prima volta, per poi reindirizzare l'utente alla pagina di login. Nel frattempo viene visualizzata una scritta di loading;

- il file login.js: contiene il codice front-end della pagina ottenuta accedendo a "localhost:3000/login". Il codice all'interno si occupa di inviare una richiesta all'endpoint "api/user/login" (descritta successivamente) per verificare la correttezza dei dati inseriti e, in base alla risposta, reindirizzare l'utente nella pagina "Scan List" se i dati inseriti sono corretti, mentre stampa un messaggio di errore in caso contrario.







CARTELLA MODELS:

Questa cartella contiene le definizioni delle collections usate nel database mongoDB attraverso la libreria mongoose.

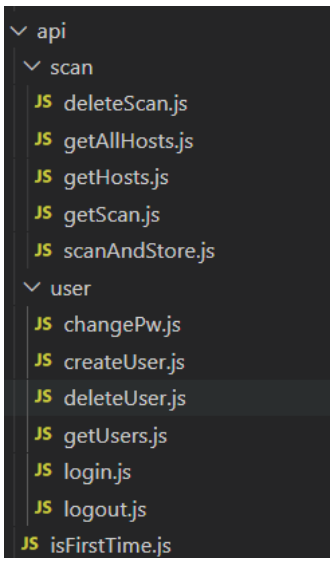
```
const userSchema = new mongoose.Schema({
    username:{
        type: String,
        required: true,
        unique: true
    },
    password:{
        type: String,
        required: true,
    },
    isLoggedIn:{
        type: Boolean,
    }
})
```

Ogni collection viene definita attraverso uno "schema mongoose", ovvero un oggetto che definisce tutti i campi presenti in ogni istanza (o in linguaggio mongo, "document"). Nella definizione dei campi si possono assegnare degli attributi come il tipo di dato (obbligatorio) oppure, come nel caso dello username, l'unicità all'interno della collection.

```
userSchema.pre('save', function(next) {
    let user = this;
    console.log('crypting password...')

    // only hash the password if it has been modified (or is new)
    if (!user.isModified('password')) return next();

    // generate a salt
    bcrypt.genSalt(SALT_WORK_FACTOR, function(err, salt) {
        if (err) return next(err);

        // hash the password using our new salt
        bcrypt.hash(user.password, salt, function(err, hash) {
            if (err) return next(err);
            // override the cleartext password with the hashed one
            user.password = hash;
            next();
        });
    });
});
```

Oltre allo schema, si possono definire dei metodi che verranno eseguiti in un preciso istante.

Nell'esempio riportato dall'immagine, viene mostrato il metodo nello schema degli utenti che viene eseguito prima di ogni salvataggio: se la password è stata modificata (o appena creata), essa viene criptata attraverso l'utilizzo della libreria bcrypt (come annunciato precedentemente).



CARTELLA PAGES/API:

Questa cartella contiene la maggior parte del codice back-end presente nel progetto.

Ogni file al suo interno rappresenta un endpoint, può dunque ricevere richieste, elaborarle, comunicare con il database ed inviare risposte. Generalmente i dati inviati e ricevuti sono in formato JSON.

Per comodità ho raggruppato i file che operano sulle scansioni nella cartella "scan", mentre quelli che gestiscono gli utenti in "user".

Il file "isFirstTime.js" invece si occupa di verificare se il database è stato inizializzato prima di ogni accesso. Come accennato in precedenza, viene fatto eseguire dalla pagina index.

isFirstTime.js

```
export default async function handler(req,res) {

    if(req.method === 'GET'){
        const isFirst = await Users.where('username').equals('admin')
        if(isFirst.length > 0){
            console.log('isFirst time--admin esistente')
            res.status(200).json(false)
        }else{
            await Counter.create({num : 0})
            await Users.create({username : "admin", password : "admin", isLoggedIn : false})
            console.log('isFirst time--admin non esistente')
            res.status(200).json(true)
        }
    }else{
        res.status(400).end()
    }
}
```

Il funzionamento di "isFirstTime.js" si può riassumere in pochi semplici passaggi:

Viene controllata la presenza dell'admin nel database e se non viene trovata viene creata. Viene inoltre inizializzato il contatore utilizzato per le scansioni.

scanAndStore.js

```
await Scan.create({counter: indx , scanner: scanner, command: command,
    outputFile: outputFile, createdAt: Date.now(), status: 'starting..',
    finishedAt: null, totalHosts: -1})
console.log('ScanAndStore -- Scan added in queue')

indx++
await Counter.updateOne({}, { num: indx })
```

Lo script "scanAndStore.js" si occupa di mettere in coda le richieste di scansioni inviate dall'admin dalla pagina Scan List.

Viene creata quindi la scansione in stato di "starting..", annullando però i campi non conosciuti a priori come il numero di hosts rilevati e il timestamp di fine scansione.

getScan.js

```
if(research === ''){
    scan = await Scan.where()
        .select(['counter','scanner', 'status', 'command', 'totalHosts', 'createdAt', 'finishedAt'])
        .sort({[param] : isAscending === true ? 1 : -1})
        .skip(skip)
        .limit(limit)
    total = await Scan.where().count()
}else{
```

"getScan.js" viene chiamato dalla pagina ScanList nel momento in cui si naviga all'interno della tabella principale delle scansioni. Attraverso i filtri e la paginazione l'utente può richiedere la visualizzazione di scansioni effettuate a proprio piacimento. Nella tabella l'utente può impostare il numero di scansioni da visualizzare per pagina, l'ordinamento delle scansioni (ascendente o discendente) e impostare il parametro su cui ricercare (per data, per numero ecc.). L'utente inoltre può ricercare direttamente delle scansioni inserendo del testo. Lo stato dei filtri viene quindi passato all'interno della richiesta in formato JSON per poi essere elaborato all'interno della query inviata al database.

I filtri inviati sono: research (il testo della ricerca), param (su quale entità effettuare l'ordinamento), isAscending, limit (il numero di elementi per pagina) e skip (quanti elementi saltare. Se ad esempio vengono visualizzati 10 elementi per pagina e sono alla pagina 4, salterò 30 elementi).

La ricerca con il testo inserito dall'utente è stata realizzata similarmente a quella proposta dall'immagine, ma tramite regular expression.
(https://www.mongodb.com/docs/manual/reference/operator/query/regex/)

getHosts.js

Questo script viene chiamato nel momento in cui l'utente preme il bottone "see details" di una determinata scansione nella pagina Scan List. Viene creata una nuova sotto-tabella nella quale vengono visualizzati gli hosts relativi ad una singola scansione. La tabella generata ha le principali caratteristiche della madre, ovvero si può navigare all'interno di essa attraverso filtri e paginazione. Il funzionamento di questo script dunque è molto simile al precedente, la differenza sostanziale è che opera sulla collection Hosts e non in Scans.

deleteScan.js

```
export default async function handler(req,res) {

    const index = req.body.index

    let scan = await Scan.where('counter').equals(index)

    if(scan.length > 0){
        let id = scan[0]._id

        await Host.deleteMany({ 'scanId': id });
        await Scan.deleteOne({ 'counter': index });

    }else{
        console.log('deleteScan --- problema');
        res.status(501).end()
    }

    res.status(200).end()
}
```

"deleteScan.js" può venire chiamato solo dall'admin nel momento in cui preme sull'icona del cestino a fianco ad ogni scansione.

L'obiettivo di questo script è eliminare tutti i dati relativi alla scansione selezionata, compresi tutti gli hosts rilevati da essa. Come parametro della richiesta viene passato l'indice della scansione da eliminare.

Una volta trovata la scansione all'interno della collection, viene memorizzato l'id della stessa per poi eliminare tutti gli host che contengono quel riferimento. Successivamente viene eliminata la scansione stessa.

getAllHosts.js

```
    let a = await Host.where().select('macAddress').distinct('macAddress')

    for(let i = 0; i < a.length ;i++){
        let b = await Host.where('macAddress').equals(a[i]).distinct('address')
        for(let j = 0; j < b.length ;j++){
            let c = await Host.where('macAddress').equals(a[i]).where('address').equals(b[j])
            if(c.length > 0){
                scan [scan.length] = c[0]
            }
        }
    }
```

"getAllHosts.js" viene chiamato dalla pagina User List per ottenere i dati da inserire nella tabella.

Il compito di questo script è produrre l'inventario degli hosts presenti nel database e restituirli alla pagina. Vengono quindi cercate nel database tutti gli hosts con coppie distinte ("macAddress" e "address") e inseriti in un array da inviare come risposta.

changePw.js

L'obiettivo di questo script è quello di aggiornare o resettare la password.

```
async function updatePw(username, pw1, pw2) {

    const x = await Users.where('username').equals(username)

    if(x.length === 0){
        return false
    }

    if(await bcrypt.compare(pw1, x[0].password)){
        x[0].password = pw2
        x[0].save()

        return true
    }else{
        return false
    }
}
```

Nel primo caso, viene effettuata la chiamata a questo endpoint dalla pagina "Change Password".

Lo username, la password da cambiare e quella nuova vengono passati come parametri. Con l'utilizzo della libreria bcrypt viene verificato se la password vecchia inserita combacia con quella criptata presente nel database per poi proseguire con l'aggiornamento.

```
if(req.body.admin === true){

    const x = await Users.where('username').equals(req.body.username)

    if(x.length === 0){
        res.status(200).json(false)
    }

    x[0].password = req.body.password
    x[0].save()

    console.log('reset password di ', req.body.username);

    res.status(200).json(true)
```

Nel secondo caso invece, l'endpoint viene chiamato dalla pagina riservata all'admin "User List".

La password presente nel database viene sovrascritta con quella di default, ovvero lo username stesso.

createUser.js

```
async function create(username, pw) {
    const x = await Users.where("username").equals(username)
    if(x.length > 0){
        //utente gia greato
        return false
    }

    await Users.create({"username" : username, "password" : pw, "isLoggedIn" : false})
    return true
}
```

"createUser.js" viene chiamato dall'admin dalla pagina "Create New User". Il suo scopo è semplicemente quello di verificare che l'username passato come parametro non sia già presente nel database per poter creare una nuova istanza.

deleteUser.js

```
const username = req.body.username

let u = await Users.where('username').equals(username)

if(u.length > 0){

    await Users.deleteOne({ 'username': username });
```

Questo script elimina un utente dal database. Può venire chiamato solo dall'admin dalla pagina "User List", cliccando sul bottone con l'icona del cestino.

Il funzionamento è molto semplice: viene verificata prima la presenza dell'utente nel database, poi viene eliminato.

getUsers.js

```
export default async function handler(req,res) {

    const isAscending = req.body.isAscending
    const skip = req.body.skip
    const limit = req.body.limit

    let tot = await Users.where()
    let u = await Users.where()
        .sort({'username' : isAscending === true ? 1 : -1})
        .select(['username'])
        .skip(skip)
        .limit(limit)

    u[u.length] = tot.length

    res.status(200).json(u)
}
```

Il funzionamento di questo endpoint è molto simile al precedente "getScan" o "getHost", il suo obiettivo infatti è quello di fornire i dati richiesti dalla pagina "User List", per produrre una tabella contenente la lista degli utenti.

I dati richiesti dalla tabella dipendono dai filtri richiesti dall'utente e dalla pagina in cui si trova.

login.js

```
if (await findPw(req.body.username, req.body.password)) {
    await Users.updateOne({ username: req.body.username }, { isLoggedIn: true })

    const token = sign({
            exp: Math.floor(Date.now() / 1000) + 60 * 60, // 1h
            username: req.body.username,
        },
        secret + req.body.username
    );

    const serialised = cookie.serialize(req.body.username, token, {
        httpOnly: true,
        secure: process.env.NODE_ENV !== "development",
        sameSite: "strict",
        maxAge: 60 * 60 ,
        path: "/",
    });

    await res.setHeader("Set-Cookie", serialised);

    res.status(200).json(true)
    console.log('login--trovato utente')
} else {
```

"login.js" viene chiamato dalla pagina di login, nel momento in cui un utente tenta di accedere. Se i dati inseriti non sono corretti, viene inviata una risposta negativa. Se al contrario lo username e la password combaciano nel database (il controllo della password viene effettuato attraverso la libreria bcrypt come descritto in precedenza), viene creato un cookie con l'utilizzo della libreria jsonwebtoken.

La durata di validità del cookie è di un'ora dalla sua creazione ed è associato all'utente che lo ha creato. Il valore del cookie dipende sia da una stringa (la variabile secret) contenuta nelle variabili locali d'ambiente che dallo username stesso. In questo modo i valori dei cookies generati sono diversi da utente ad utente.

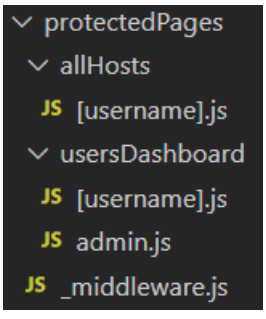
logout.js

```
const serialised = cookie.serialize(req.body.username, '', {/
    httpOnly: true,
    secure: process.env.NODE_ENV !== "development",
    sameSite: "strict",
    maxAge: 0,
    path: "/",
});
```

Questo script viene chiamato nel momento in cui un utente preme il bottone "logout".

Oltre ad aggiornare a 'false' il campo "isLoggedin" dell'utente, viene eliminato il cookie precedentemente creato, sostituendolo con un cookie di durata 0 che si autodistrugge immediatamente.

CARTELLA PAGES/PROTECTEDPAGES

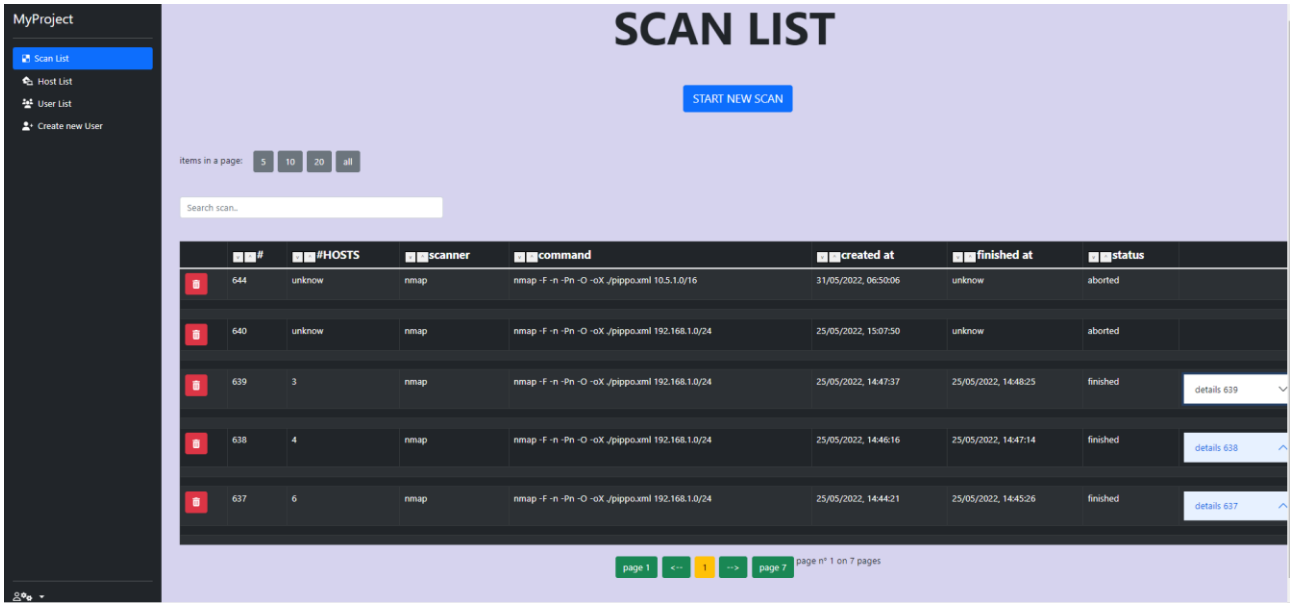


Questa cartella contiene il codice front-end delle pagine: Host List (/allHost/[username].js), Scan List (/usersDashboard/[username].js e admin.js) e del codice backend (_middleware.js).

La presenza del file _middleware.js permette a questa cartella di effettuare delle operazioni (in questo viene controllata la presenza del cookie ottenuto dal login) prima dell'accesso delle pagine al suo interno.

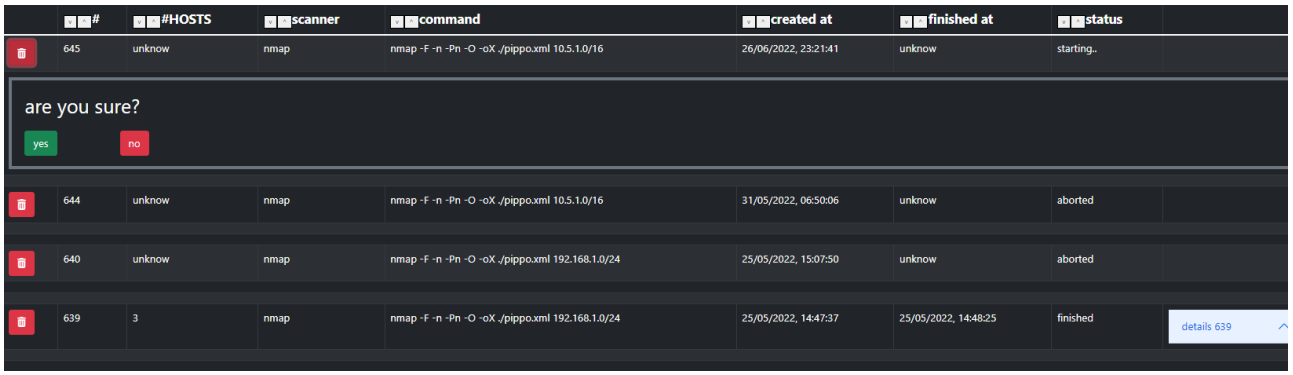
I file [username].js permettono il routing automatico delle pagine (come accennato in precedenza). Spiegandolo attraverso un esempio, se un utente accede alla pagina "/protectdPages/usersDashboard/luca" oppure alla pagina "/protectdPages/usersDashboard/daniele", la pagina caricata è sempre la stessa.

usersDashboard/admin.js

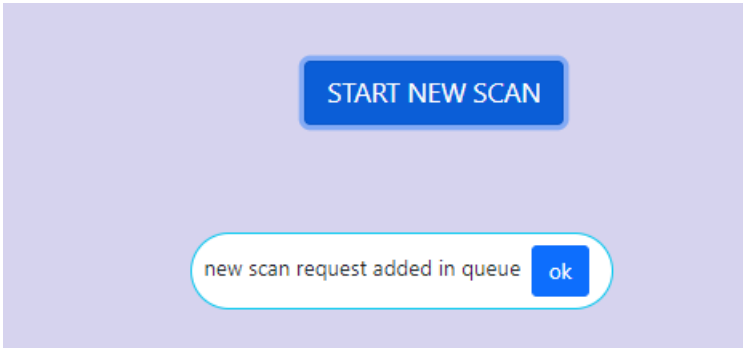


Questo file contiene il codice front-end della pagina “Scan List” visualizzata dall’admin. Contiene più dati e operazioni disponibili rispetto a quella riservata agli utenti comuni come:

- visualizzare più dettagli di ogni scansione (come accennato precedentemente);
- eliminare una scansione:



- creare una nuova scansione:



Ogni azione che può fare l’utente è gestita tramite delle richieste agli endpoint (cartella api) descritti in precedenza.

```
async function handleTrash(index){

    await fetch('/api/scan/deleteScan', {
        method : 'DELETE',
        body: JSON.stringify({index: index}),
        headers:{
            'Content-Type' : 'application/json'
        }
    })
    seeScanList()
}
```

Esempio di chiamata all'api "api/scan/deleteScan" per poter eliminare una scansione. Viene passato come parametro l'indice della scansione.

usersDashboard/[username].js

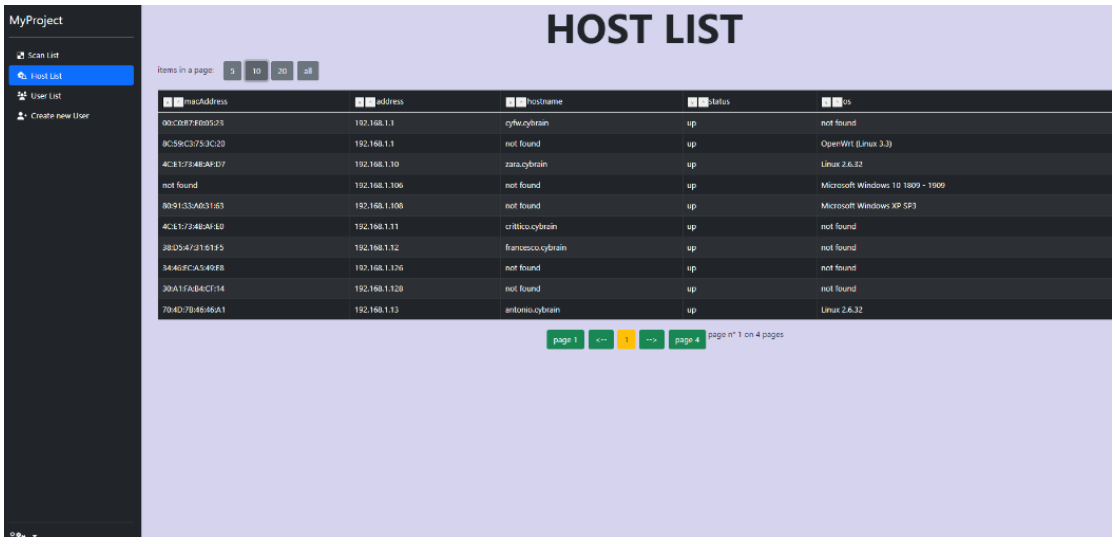
Contiene il codice front-end della pagina "Scan List" visualizzata dagli utenti comuni, le differenze da quella riservata all'admin sono state descritte in precedenza.

```
const router = useRouter()
const username = router.query.username

async function handleLogOut(){
    const response = await fetch('/api/user/logout',{
        method : 'POST',
        body: JSON.stringify({'username': username}),
        headers:{
            'Content-Type' : 'application/json'
        }
    })

    router.replace('../../login')
}
```

Questa immagine rappresenta un utilizzo pratico del routing automatico offerto da NextJs chiamando il file [username].js.

Dal codice si può notare come viene immagazzinato il nome dell'utente in una variabile per poter inviare correttamente una richiesta all'api "api/user/logout" nel caso in cui l'utente voglia effettuare il logout da questa pagina.

allHost/[username].js



Contiene il codice front-end della pagina “Host List”. In questa pagina non sono presenti differenze sostanziali tra la visualizzazione dei dati dell’admin da quella degli utenti comuni. I dati mostrati nella tabella vengono prelevati dall’api “api/scan/allHosts”.

```
<div className = 'd-flex flex-column bd-highlight justify-content-center mb-3'>
    <div className = 'ms-5 d-flex justify-content-center'>
        <button className="btn btn-success m-1" onClick={()=>{setSkip(0)}}>page 1</button>
        <button className="btn btn-success m-1" onClick={()=>{setSkip(skip < limit ? 0 : skip - limit)}}>{'<--'}</button>
        <button className="btn btn-warning m-1" onClick={()=>{}}>{Math.ceil(skip/limit) + 1}</button>
        <button className="btn btn-success m-1" onClick={()=>{setSkip(skip >= total - limit ? skip : skip + limit)}}>{'-->'}</button>
        <button className="btn btn-success m-1" onClick={()=>{setSkip(total < limit ? 0 : total - limit)}}>page {Math.ceil(total/limit)}</button>
        <span>page n° {Math.ceil(skip/limit) + 1} on {Math.ceil(total/limit)} pages</span>
    </div>
</div>
```

In questa immagine viene mostrato il funzionamento dei bottoni che permettono la navigazione all’interno delle tabelle.

_middleware.js

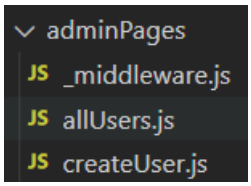
```
const { cookies } = req;
const jwt = cookies[username];

if( jwt === undefined){
    console.log('no cookies founded')
    return NextResponse.redirect(`${server}/login`)
}

try{
    verify(jwt, secret + username)
    return NextResponse.next()
}catch(e){
    console.log('no cookies verified')
    return NextResponse.redirect(`${server}/login`)
}
```

Il middleware all’interno di questa cartella ha il compito di verificare che gli utenti che accedono alle pagine appena descritte abbiano effettuato l’accesso attraverso la pagina di login.

Con l’aiuto della libreria jwt, vengono prelevati i cookies e verificati.

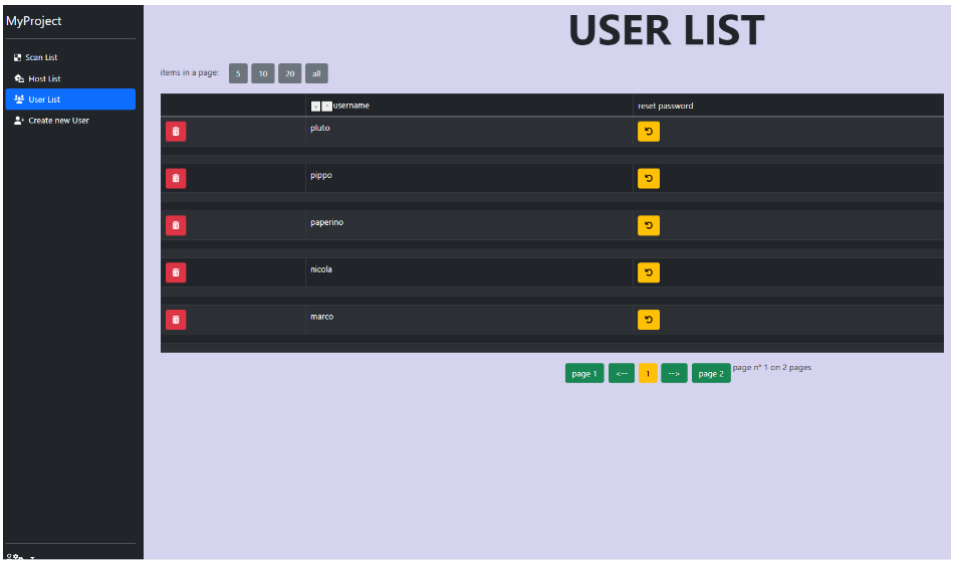
Se sono corretti, si può procedere al caricamento della pagina richiesta attraverso la chiamata .next(), altrimenti si viene reindirizzati alla pagina di login.

CARTELLA PAGES/ADMINPAGES



Questa cartella contiene delle pagine alle quali può effettuare l’accesso solo l’admin.

allUser.js



Il codice di questo file implementa la pagina "User List". Da qui l'admin può gestire gli utenti resettando le password o eliminando qualche istanza. Questa pagina comunica con gli api "deleteUser", "getUsers" , "logout" e "changePw" per effettuare le azioni previste e prelevare i dati dal database.

```
<div id={`_${d.username}trash`} className="accordion-collapse collapse m-0">
    <div className="accordion-body border border-secondary border-5">
        {
            <>
                <p>do you really want to delete {d.username}?</p>
                <button className="btn btn-success me-5" data-bs-toggle="collapse" data-bs-target={`#_${d.username}trash`} onClick={()=>{handleTrash(d.username)}}>yes</
                <button className="btn btn-danger mx-5" data-bs-toggle="collapse" data-bs-target={`#_${d.username}trash`}>no</button>
            </>
        }
    </div>
</div>
```

Nel pezzo di codice sopra viene mostrato come viene effettuato un "accordion" utilizzando la libreria bootstrap, per ottenere una finestra "a comparsa", nel momento in cui viene premuto il pulsante con l'icona del cestino.

createUser.js

Questo file implementa la pagina "Create New User". Da qui l'admin può creare dei nuovi utenti. Se la password inserita è oltre i 10 caratteri o il nome utente è già presente compare un messaggio di errore.

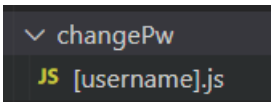
Questa pagina comunica con l'api "createUser" per poter inserire il nuovo utente nel database o ricevere una risposta negativa nel caso l'utente sia già stato creato.

_middleware.js

Come nella cartella precedentemente descritta, qui è presente il file "_middleware.js" che permette di verificare, prima del caricamento delle pagine, l'autenticità dell'utente.

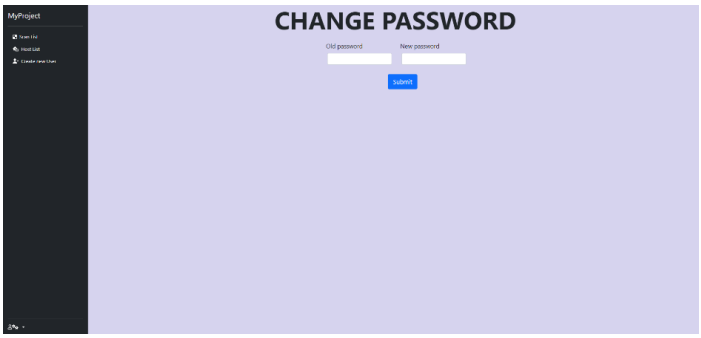
L'unica differenza dal middleware precedente è che viene verificata la presenza del cookie associato all'admin e non ad un utente comune.

CARTELLA PAGES/CHANGEPW



Contiene un solo file, "[username].js".

Lo scopo della pagina è quello di permettere di modificare la password a qualsiasi utente. Essendo un file dinamico, si può ottenere dall'url lo username dell'utente che vuole cambiare la password.



Come nel caso della creazione di un utente, se la password inserita supera i 10 caratteri o viene inserita una password errata, compare un messaggio di errore.

Questa pagina comunica con l'api "changePw" per poter verificare la correttezza della password e modificarla nel database.

Librerie utilizzate:

BACK-END:

- xml2js: utilizzata per convertire l'output di Nmap da formato xml in json;
- bcrypt: utilizzata sia per criptare le password degli utenti nell'azione di creazione e di modifica password che per confrontare le password inviate nell'azione di login con quelle criptate nel database;
- agenda: utilizzata per lo script parallelo che rileva le scansioni da effettuare, permette di eseguire un algoritmo ogni periodo di tempo desiderato;
- jsonwebtoken: utilizzata sia per creare i token da inserire come cookies che per confrontare i cookies posseduti con quelli richiesti per l'accesso ad una determinata pagina;
- mongoose: ampiamente utilizzata per comunicare con il database MongoDB.

FRONT-END:

- react: ampiamente utilizzata per la realizzazione della struttura delle pagine.
- bootstrap: ampiamente utilizzata per lo stile delle pagine (in particolare per lo stile delle tabelle, della barra di navigazione, per i pulsanti e per gli accordion).
- @fortawesome: utilizzata per le icone.

Considerazioni finali e possibili miglioramenti:

Gli obiettivi del tirocinio sono stati raggiunti in buona parte, nonostante le diverse difficoltà riscontrate a causa del primo approccio ad un linguaggio front-end del sottoscritto.

Al termine del tirocinio, sono stati discussi dei possibili miglioramenti per perfezionare il funzionamento dell'applicazione, come:

- migliorare i filtri di ricerca nelle tabelle (come l'ordinamento degli indirizzi ip che attualmente è in ordine alfabetico);
- fondere lo scheduler con il resto dell'applicazione per evitare di lanciarlo manualmente da riga di comando;
- aumentare il livello di sicurezza per l'accesso agli script nella cartella api;
- a livello di funzionalità potrà essere aggiunto un sistema che permetta all'utente amministratore di avviare una sequenza di scansioni ad una distanza di tempo determinata;
- dato il mio basso livello di esperienza, il codice front-end potrebbe risultare leggermente contorto e poco mantenibile, in futuro potrà essere riscritto per sfruttare a pieno le potenzialità del framework NextJs.

Il progetto integrale è disponibile su: https://github.com/Zeppo00/tirocinio .