Gestione whitelist di un proxy

[1 Introduzione 3](#_Toc777879362)

[1.1 Informazioni sul progetto 3](#_Toc1520044224)

[1.2 Abstract 3](#_Toc1120212846)

[1.3 Scopo 3](#_Toc1215197914)

[2 Analisi 3](#_Toc307781526)

[2.1 Analisi del dominio 3](#_Toc2062769517)

[2.2 Analisi e specifica dei requisiti 4](#_Toc526557234)

[2.3 Use Case 5](#_Toc849944095)

[2.4 Pianificazione 6](#_Toc574115073)

[2.5 Analisi dei mezzi 2](#_Toc402411840)

[3 Progettazione 3](#_Toc21423734)

[3.1 Design dell’architettura di sistema 3](#_Toc319359754)

[3.2 Design dei dati e database 3](#_Toc918496903)

[3.3 Design procedurale 4](#_Toc956280273)

[3.4 Schema di rete 10](#_Toc569183492)

[4 Implementazione 11](#_Toc384143433)

[4.1 Ubuntu Server 11](#_Toc1448245937)

[4.2 Template MVC 12](#_Toc2052415515)

[4.3 Pagina login 13](#_Toc20065881)

[4.4 Pagina amministrativa 16](#_Toc345625162)

[5 Test 17](#_Toc596786953)

[5.1 Protocollo di test 17](#_Toc2049835867)

[5.2 Mancanze/limitazioni conosciute 17](#_Toc502664715)

[6 Consuntivo 17](#_Toc43860083)

[7 Conclusioni 17](#_Toc1482088537)

[7.1 Sviluppi futuri 17](#_Toc1075271476)

[7.2 Considerazioni personali 17](#_Toc1877635208)

[8 Bibliografia 17](#_Toc2083717211)

[8.1 Sitografia 17](#_Toc1091115777)

[Allegati 17](#_Toc280777956)

# Introduzione

## Informazioni sul progetto

Autore: Carlo Pezzotti

Scuola: Arti e Mestieri Trevano

Classe: I4AC

Anno scolastico: 2019-2020

Sezione: Informatica

Docenti responsabili: Ivan Raimondi

Data di inizio: 03.09.2019

Data di consegna: 20.12.2019

## Abstract

----

## Scopo

Lo scopo di questo progetto è quello di creare un’applicazione web che semplifichi la gestione dei siti nella whitelist di un proxy in ogni suo aspetto.

# Analisi

## Analisi del dominio

Il dominio per questa applicazione è pressoché inesistente, in quanto l'applicazione per la gestione di un proxy deve venir fatta da capo partendo dai requisiti del cliente e non esistono applicazioni che potremmo usare come modello dalle quali ispirarci.

Il prodotto dovrà funzionare in contesto scolastico o lavorativo, dove un proxy è fondamentale per la sicurezza informatica.

Gli utenti che utilizzeranno il prodotto saranno sicuramente di due tipi,

gli informatici che si occuperanno della parte amministrativa e l'utente medio che lo utilizzerà. Siccome si basa per una fascia di utenti variabile, il suo funzionamento dovrà essere semplificato al massimo.

## Analisi e specifica dei requisiti

Il committente ha bisogno di un prodotto che gli fornisca la possibilità di gestire un proxy da un pagina web. Dovrà gestire,inoltre , gli utenti che possono accedere a questa gestione del proxy, perchè, come specificato durante gli incontri, per accedere alla gestione bisogna passare da una pagina di login.

Il prodotto finale quindi dovrà essere sia il sistema che ospiterà il prodotto e il proxy che verrà gestito.

Il prodotto, sotto richiesta, avrà delle interfacce con un grafica minima per renderlo leggero e trasportabile in caso di necessità.

Le prestazione minima che il prodotto dovrà fornire è la gestione del proxy.

Siccome il proxy server per garantire sicurezza all'interno di una rete, il grado di sicurezza dovrà essere elevato.

|  |  |
| --- | --- |
| **ID: REQ-001** | |
| Nome | Ubuntu server |
| Priorità | 1 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Proxy Squid |
| 002 | PHP |
| 003 | Apache |

|  |  |
| --- | --- |
| **ID: REQ-002** | |
| Nome | Proxy Squid |
| Priorità | 1 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Deve girare dulla porta 8080 |
| 002 | Non deve avere limitazioni |

|  |  |
| --- | --- |
| **ID: REQ-003** | |
| Nome | Interfaccia login |
| Priorità | 2 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Dovrà essere presente un capo per l’username e uno per la password |

|  |  |
| --- | --- |
| **ID: REQ-004** | |
| Nome | Interfaccia di gestione utenti |
| Priorità | 2 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Pagina per poter modificare/rimuovere/aggiungere utenti. |
| 002 | Ogni utente dovrà avere un grado di permesso (user/admin) |

|  |  |
| --- | --- |
| **ID: REQ-005** | |
| Nome | Gestione lista whiteslist |
| Priorità | 1 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Controllo esistenza sito |
| 002 | Controllo sintassi |
| 003 | Scrittura corretta su file |
| 004 | Auto riavvio alla modifica della lista |

|  |  |
| --- | --- |
| **ID: REQ-006** | |
| Nome | Grafica minima |
| Priorità | 2 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Pagine con grafica minimale |

|  |  |
| --- | --- |
| **ID: REQ-007** | |
| Nome | Database |
| Priorità | 2 |
| Versione | 1.0 |
| Note |  |
| *Sotto-requisiti* | |
| 001 | Tabella per gestione utenti |

## Use Case

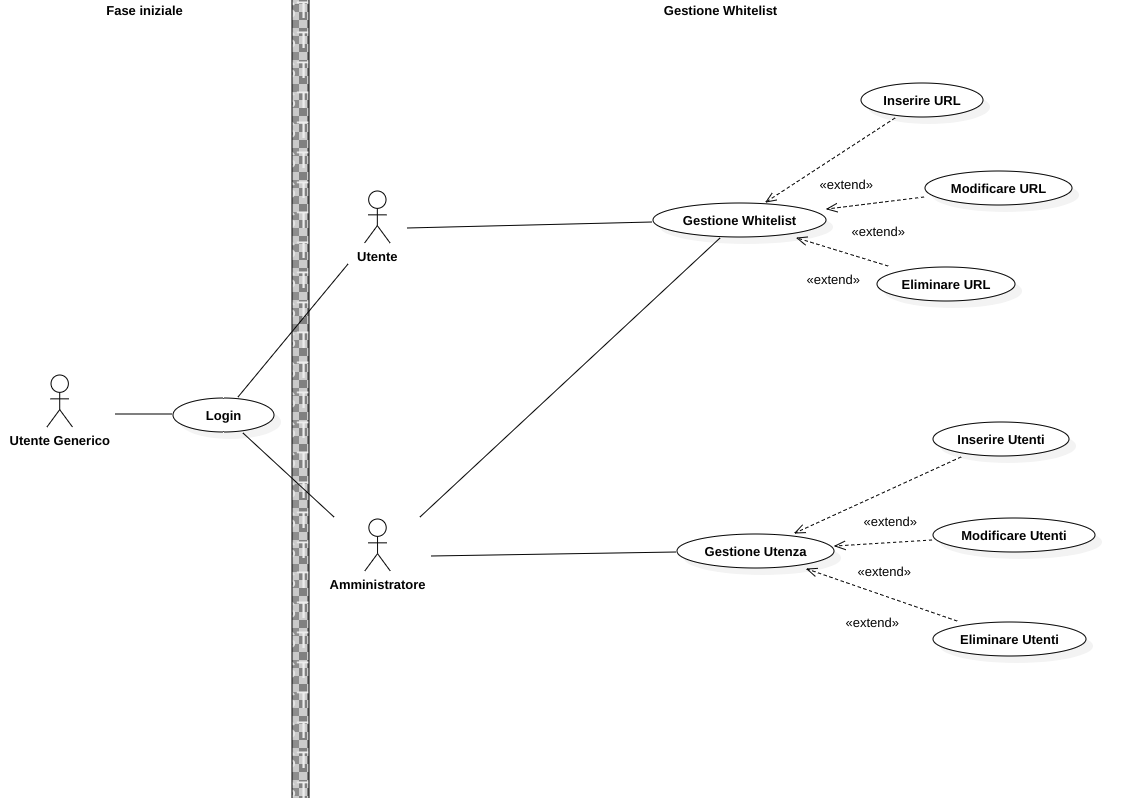


Figure 1 Use Case

Un utente generico fa il login alla pagina.  
Gli utenti si dividono in 2 categorie  
Utenti normali e Amministratori.

Gli **utenti normali**, possono solo interagire con la gestione della whitelist.

Gli **amministratori** possono sia fare quello che possono fare gli utenti normale e in più gestire la utenza del sito.

## Pianificazione

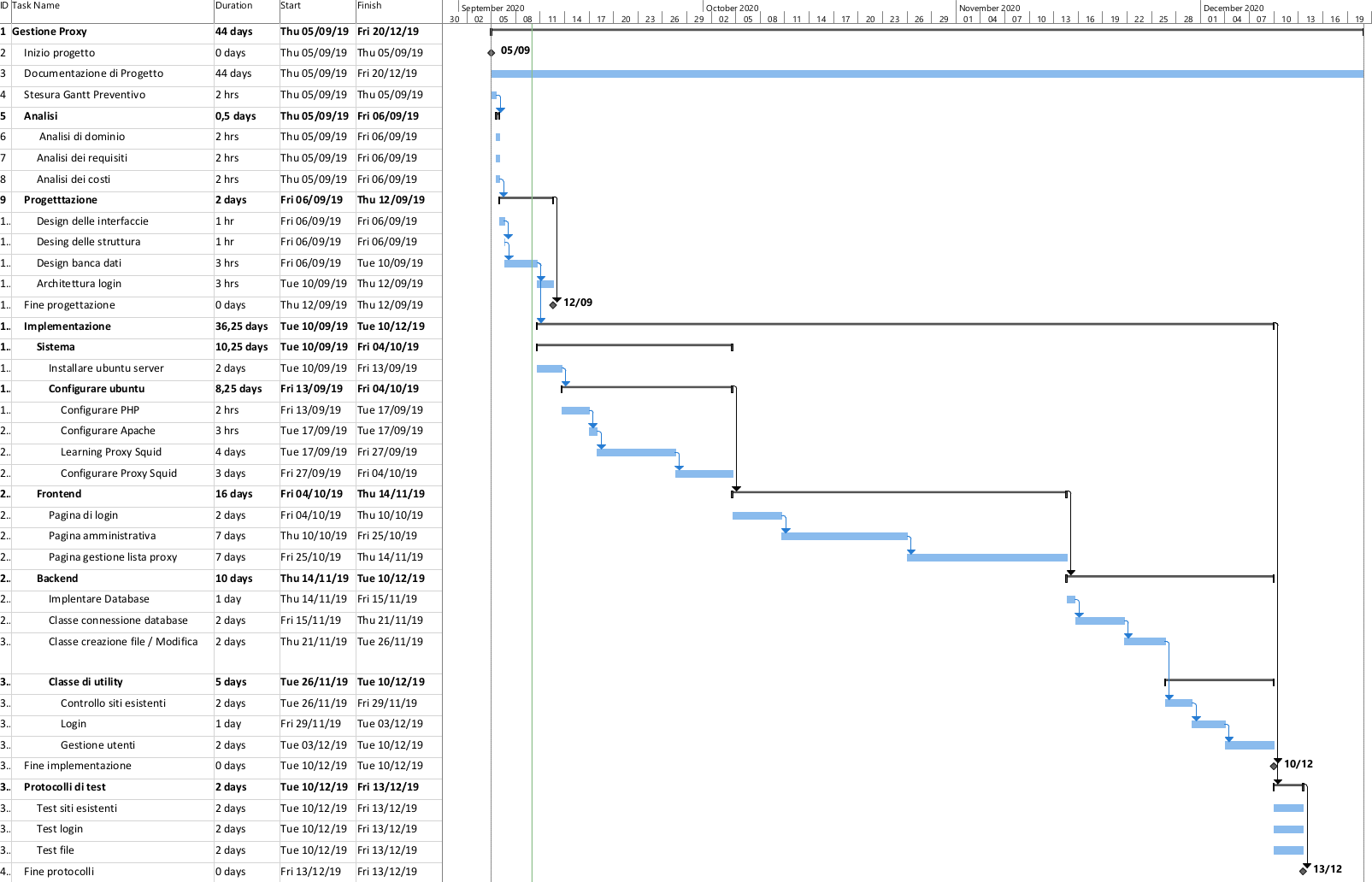


Figure 2 Pianificazine

### Analisi

L’attività delle analisi si divide in 3 sotto attività:

**Analisi di dominio:** ovvero l’analisi della situazione prima della realizzazione del progetto per considerare la validità di iniziare un progetto oppure no.

**Analisi dei requisiti:** ovvero l’analisi più approfondita di quello che il committente chiede che venga realizzato come progetto, nel mio caso ho dovuto valutare che sistema operativo scegliere, che versione di proxy squid usare e come realizzare (bozza) le varie interfacce presenti.

**Analisi dei costi:**ovvero il costo completo complessivo che questo progetto verrà a costare al mio committente in questo caso nullo.

### Progettazione

La progettazione è consistita nell’organizzare l’implementazione del progetto in modo da dividere il lavoro in modo efficace e non avere problemi.

Essa si divide in:

Design banca dati, ovvero la progettazione del database tramite la realizzazione di un diagramma Entità/Relazioni e del relativo schema logico.

Infrastruttura del server, ossia la progettazione di ogni componente che verrà poi installato sul server di produzione.

### Implementazione

L’implementazione è stata la parte più lunga del progetto oltre alla documentazione. È l’attività durante la quale ho dovuto realizzare il progetto in ogni suo aspetto, basandomi sulla progettazione fatta nel punto precedente. Essa è divisa nelle seguenti operazioni:

* Installare ambiente, che a sua volta è categorizzato da:
  + Installare server, ovvero l’installazione e la configurazione di tutti i componenti di base necessari per l’utilizzo del sistema operativo.
* Configurare ambiente, che si divide in:
  + Installare MySQL
  + Installare PHP
  + Installare Apache2
  + Installare Proxy Squid
    - Configurare Proxy Squid.
* Realizzare Frontend, ovvero realizzare le seguenti pagine web:
  + Pagina di login
  + Pagina gestione utenti
  + Pagina gestione lista
* Realizzare Backend
  + Gestione delle connessi con il database
  + Gestione del login
  + Getione dei siti della whitelist
* Protocolli di test
  + Controllo che i siti siano esistenti
  + Controlli anti sql injection
  + Controllo sui file

### Documentazione

Infine c’è un’attività rimasta costante durante tutto il progetto: la documentazione. Infatti durante i tempi morti documentavo il progetto e aggiugevo informazioni al file che state leggendo in questo momento in modo da non doverla scrivere tutta in poco tempo alla fine

## Analisi dei mezzi

### Software

Macchina server:

* Proxy Squid
* Apache 2
* PHP 7.2

Macchina sviluppo:

* Apache 2
* PHP 7.2
* Visual Studio Code (Ide e Debugger)

### Hardware

Macchina server:

* Ubuntu Server 18.4
* 1 GB di memoria RAM
* 25 GB di disco disponibili

# Progettazione

## Design dell’architettura di sistema

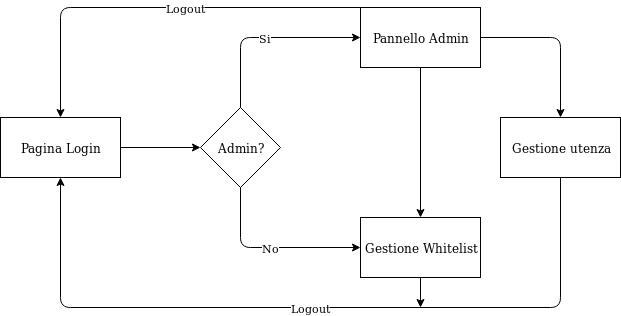


Figure 3 Architettura

## Design dei dati e database

Il database sviluppato per essere utilizzato con questa applicazione è stato progettato tramite il seguente schema E/R e logico:

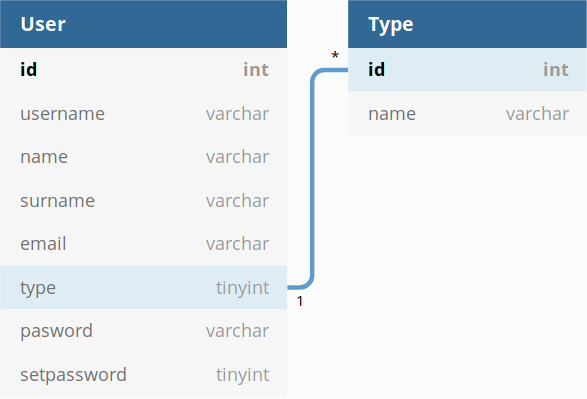


Figure 4 Database

**User:**

1. ID: L’identificatore del utente che verrà utilizzato per identificarlo nelle pagine
2. Username: L’username dell’utente utilizzato per loggarsi nel sito
3. Name: nome dell’utente
4. Surname: cognome dell’utente
5. Email: email dell’utente
6. Type: tipo dell’utente che si collega alla tabella type
7. Password: password dell’utente

**Type:**

1. ID: L’identificare della tabella
2. Nome: nome (admin/user)

## Design procedurale

Siccome ho deciso di usare una struttura MVC ho dovuto separare chiaramente il progetto per classi di seguito vi è una foto dell’UML finale che ho generato:

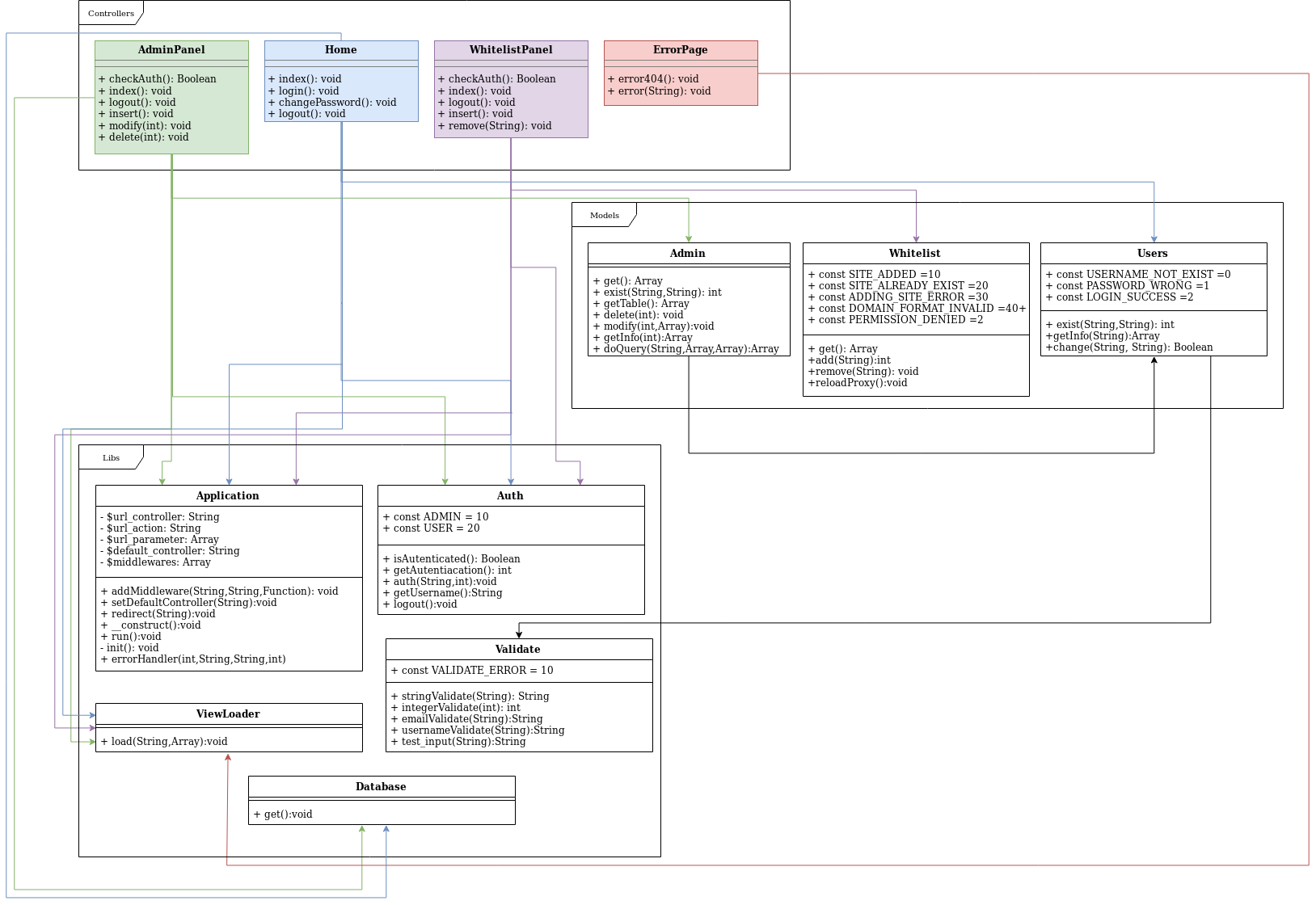


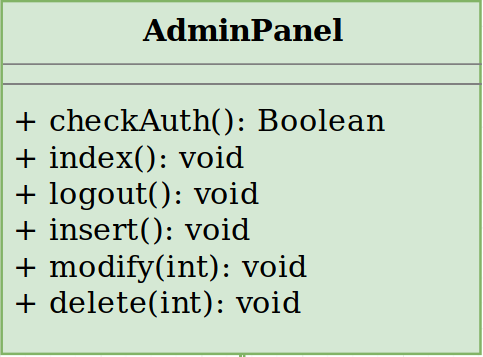
Figure 5 Diagramma UML

A prima vista può sembrare complicato siccome vi sono molte classi e molto di loro si ritrovano a chiamare spesso la stessa classe.Per facilitare la lettura ho deciso di divere le varie classi per la cartella in cui si trovano “Controllers,Model,Libs”.

Di seguito vi è la spieazione delle diverse classi e funzioni.

### Admin panel

La classe AdminPanel, come si può vedere nello schema più in alto, è un controller. Siccome ogni controller è diverso non sono riuscito a generalizzarlo in un super controller, per poi estenderlo.



**checkAuth():**Funzione che controlla che tipo di autenticazione è stato eseguito all'interno del sito. Siccome ci troviamo nel pannello admin il seguente metodo ritorna true solo nel caso che l'accesso sia di tipo admin

**index():** Metodo che mostra l’index del controller

**logout():** Metodo di logout, esegue logout pulendo e distruggendo la sessione.

**insert():** Metodo di aggiunta di utenti nel database. Il seguente metodo riceve via post i dati degli utenti, prima di inserirli vengono effettuati tutti i controlli del caso. Nel caso i controlli non dovessero passare viene mostrato a schermo un messaggio di errore.

**modify(int):** Metodo che va a modificare i dati degli utenti. Riceve via POST tutte le informazioni necessarie.Se nella variabile POST non è settato il valore username, quest'ultimo viene utilizzato come get per andare ad aggiungere nei campi di input della pagine le informazione dell'utente.

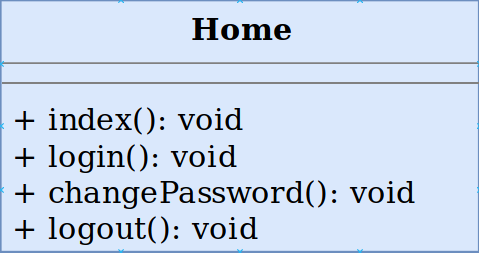
@param: l'id dell'utente da modificare

**delete(int):** Metodo che rimuove dal database un utente.

@param: l’id dell’utente da rimuovere.

### Home

La classe Home, come si può vedere nello schema più in alto, è un controller e anche se il nome può ingannare quest’ultima si occupa di gestire tutto ciò che è login.

****

**index():** Metodo che mostra l’index del controller

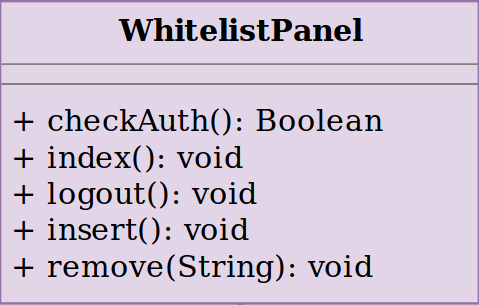
**logout():** Metodo di logout, esegue logout pulendo e distruggendo la sessione.

**changePassword():** Metodo utilizzato per cambiare la password di un utente.

**login():** Metodo che cerca di eseguire il login al database, vengono salvati in sessione l'username dell'utente e il tipo di permessi (ADMIN o USER). Dopodiché si viene reindirizzati su una nuova pagina.

### WhitelistPanel

La classe WhitelistPanel, come si può vedere nello schema più in alto, è un controller che gestisce la pagina principale del sito web, ovvero quella di inserimento e rimozione di contenuti all’interno della whitelist.



**checkAuth():** Funzione che controlla che tipo di autenticazione è stato eseguito all'interno del sito. Siccome ci troviamo nel pagina della whitelist il seguente metodo ritorna true nel caso si è *USER* oppure *ADMIN*.

**Index():** Metodo che mostra l'index del controller dopo aver effettuato dei controlli sul file.

**Logout():** Metodo di logout, esegue logout pulendo e distruggendo la sessione.

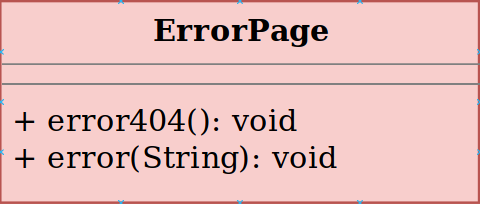
**Insert():** Metodo chiamato per inserire all'interno del file specifico un sito. La pagina può variare il contenuto in base ai messaggi che vengono generati.

**Remove(String):** Metodo che rimuove un sito dal file.

@param il sito da voler rimuovere.

### ErrorPage

La classe WhitelistPanel, come si può vedere nello schema più in alto, è un controller che gestisce le pagine di errore.



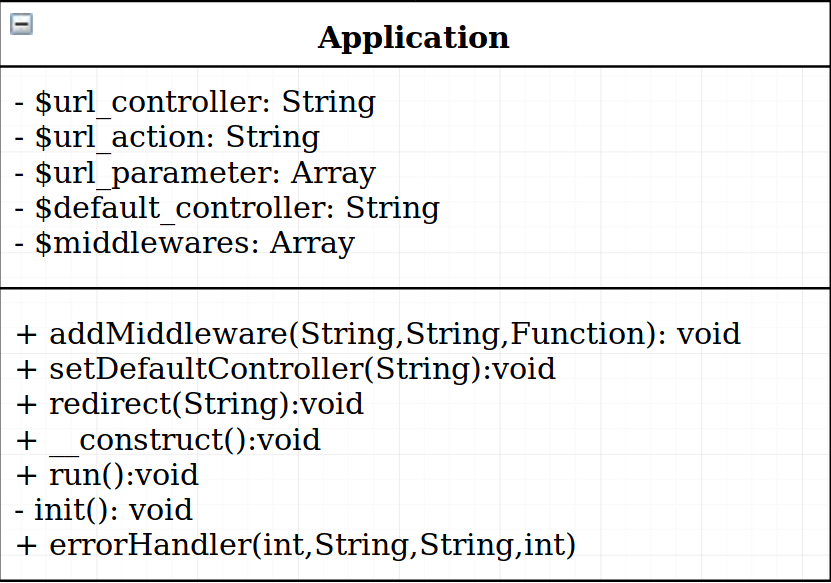
**error404():** Metodo che carica la pagina di errore 404.

**error(String):** Metodo che carica una pagina di errore personalizzata.

@param il messaggio di errore da voler mostrare

### Application

Per completezza mi sembra giusto aggiungere tutte le classe, di questa classe non spiego per filo e per segno ciò che fa siccome era già esistente nel template da me preso.



### Database

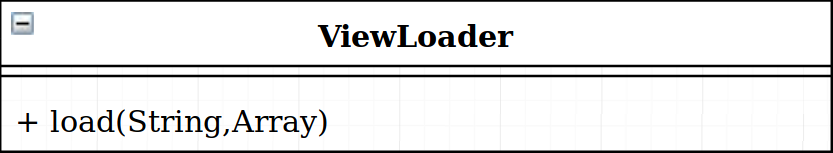
La classe database non fa altro che istanziare la connesione al database appunto.



**get():** ritorna l’istanza di connessione al database.

### ViewLoader

La classe ViewLoader non fa altro che caricare le pagine web.

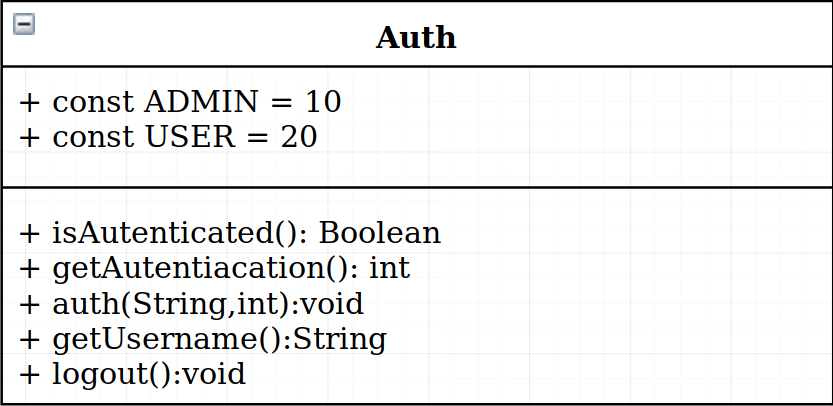


**load():** Metodo che carica una pagina web e gli passa degli argomenti se questi sono passati come parametro.

@param: il file da caricare, un array indicizzato di argomenti da poter passare.

### Auth

La classe Auth gestisce l’autenticazione di un utente alla pagina. Può gestire esclusivamente l’utente comune *USER* e il super utente *ADMIN.*

**

**ADMIN:** costante per il valore di admin

**USER:** costante per il valore di user.

**isAutenticated():** Metodo generico che verifica se si è autenticati oppure no.

**getAutentication():** Metodo get che ritorna il tipo di autenticazione.(ADMIN o USER).

**auth(String,int):** Metodo che esegue l'autenticazione.

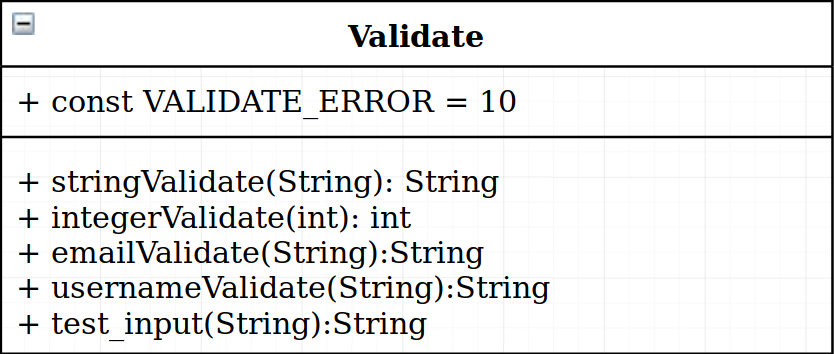
@param: l’username dell’utente che effettua l’autenticazione, il tipo dell’utente.

**getUsername():** Metodo che ritorna l’username dell’utente loggato.

**logout():**Metodo di logout, esegue logout pulendo e distruggendo la sessione.

### Validate

La classe Validate gestisce una serie di metodi statici che permettono di fare dei controlli approfonditi sulle stringe che poi andranno inserite all’interno del database.



**VALIDATE\_ERROR:** costante per il valore di validazione errata.

**stringValidate(String):** Metodo che verifica se il valore sia una stringa senza caretteri speciali.

@param: il valore da controllare

**integerValidate:** Metodo che verifica se il valore sia un intero.

@param: il valore da controllare

**emailValidate:** Metodo che verifica se il valore sia una mail.

@param: il valore da controllare

**usernameValidate:** Metodo che verifica se il valore vada bene come username.

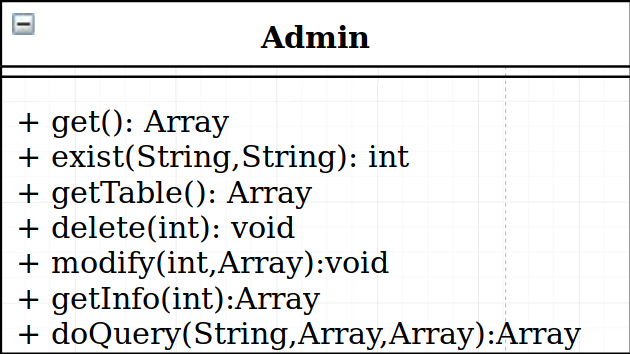
@param: il valore da controllare

**test\_input(String):** Metodo che rimuove tutti i caratteri speciali.

@param, il valore da voler modificare.

### Admin

La classe Admin come si può vedere nello schema più in alto serve per gestire tutte le operazioni logiche del controller AdminPanel.



**Get():** Metodo che ritorna un array di tutti gli utenti presenti nel database.

**Exist(String,String):** Metodo che controlla se un utente esita nel database.

@param: l’username dell’utente, la password dell’utente

**getTable():** Metodo che ritorna la tabella con tutte le informazioni non sensibili di un utente in modo formattato.

**Delete(int):** Metodo che in base ad un id elimina dal database un utente.

@param: l’id dell’utente

**Modify(int,Array):** metodo che in base ad un array indicizzato modifica le informazioni di un utente.

@param: l’id dell’utente da modificare, le informazioni da cambiare.

**getInfo(int):** metodo che ritorna tutte informazioni di un utente.

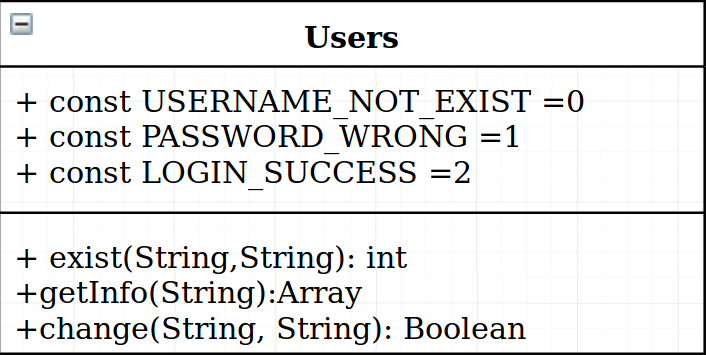
@param: l’id del’utente

**doQuery(String,Array,Array):** metodo generico che esegue una query e fa il bind dei valori passati come parametro.

@param: la query da eseguire, i valori da bindare, il tipo dei valori.

### Users

La classe Admin come si può vedere nello schema più in alto serve per gestire tutte le operazioni logiche del controller Home.



**Exist(String,String):** Metodo che controlla se un utente esita nel database.

@param: l’username dell’utente, la password dell’utente

**getInfo(int):** metodo che ritorna tutte informazioni di un utente.

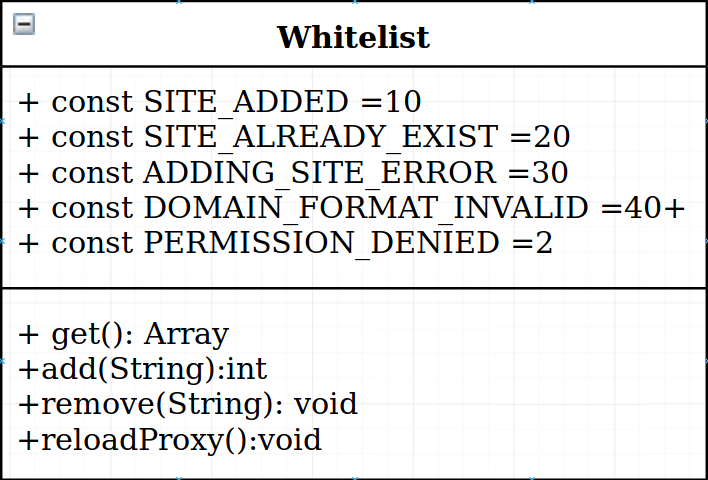
@param: l’id dell’utente

**Change(String,String):** Metodo che cambia la password di un utente.

@param: l’utente, la nuova password.

### Whitelist

La classe Whitelist come si può vedere nello schema più in alto serve per gestire tutte le operazioni logiche del controller WhitelistPanel

.

**SITE\_ADDED:** costante per il valore di sito aggiunto

**SITE\_ALREADY\_EXIST:** costante per il valore di sito esistente

**ADDING\_SITE\_ERROR:**costante per il valore di sito errato

**DOMAIN\_FOMRAT\_INVALID:** costante per il valore di domino errato

**FILE\_NOT\_EXIST:** costante per il valore di file non esistente

**PERMISSION\_DENIED:** costante per il valore di permesso negato.

**get():** Metodo che ritorna tutto il contenuto del file.

**Add(String):** Metodo che aggiunge al file un sito

**Remove(String):** Metodo che rimuove dal file un sito.

**reloadProxy():** Metodo che riconfigura il proxy tenendolo aggiornato costantemente.

## Schema di rete

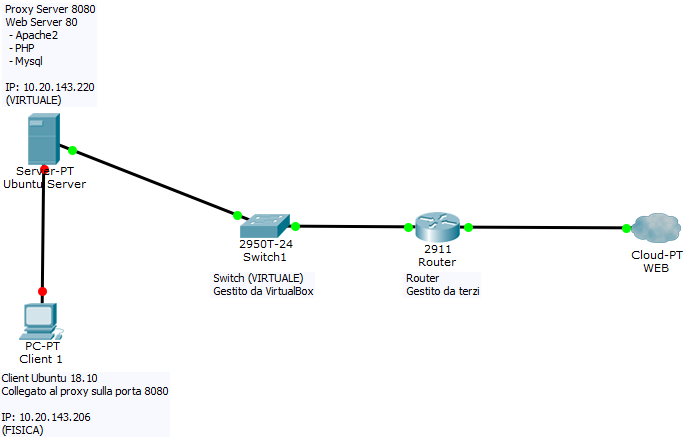


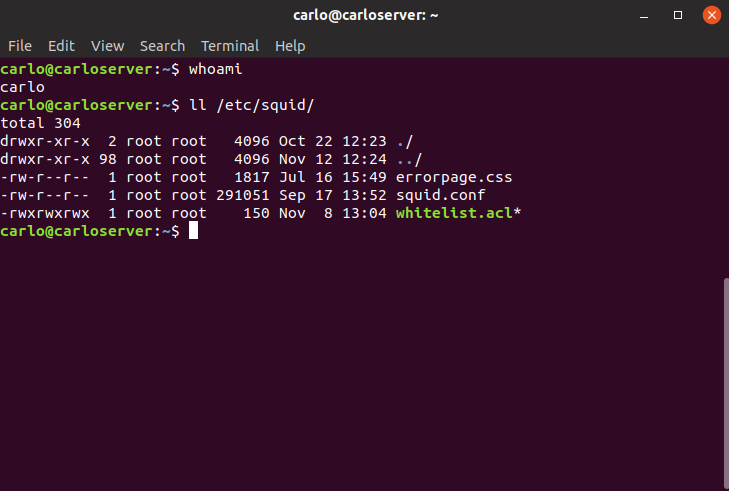
Figure 6 Schema di rete

# Implementazione

L’implementazione, come detto in precedenza, è stata la parte più lunga del progetto. Il tutto è iniziato con l’installazione del ambiente di sviluppo:

## Ubuntu Server

Come prima cosa appunto ho dovuto installare un server sul quale di seguito installare un webserver, nel mio caso **apache2** versione 2.4.29 (Ubuntu), **Php** versione 7.2.24, **mysql** versione Ver 14.14 Distrib 5.7.27 ed infine **proxy squid** versione 3.5.27.



Per l’installazione di apache e PHP, siccome sapevo già come fare, non ho seguito nessuna guida ed ho utilizzato i seguenti comandi per installare:

***sudo apt install apache2***

***sudo apt install php7.2 php7.2-mysql***

***sudo apt install mysql-server***

Invece per l’installazione di *squid,* siccome era una nuova cosa per me, ho cercato una guida online dove mi spiegasse passo per passo come fare cosa. L’installazione è stata semplice, rispetto alla configurazione che mi ha dato un po’ di rogne all’inizio.

Proxy Squid:

Set port line: 1612

**Port** value: 8080

Set http\_access line: 1194

**http\_access** value: allow all

Configurazione per una whitelist:

acl whitelist dstdomain "/etc/squid/whitelist.acl" http\_accessallow whitelist

Di seguito ho aggiunto un paio di siti all’interno del file whitelist.acl per testare il funzionamento ed ha funzionato.

## Template MVC

Premessa, non sono andato a vedere dettagli cosa facesse il template siccome parlando con alcuni compagni ho saputo che funzionava pareccio bene.

Le due principali classi che ho dovuto utilizzare sono le seguenti:

### Application

La seguente classe mette a disposizione un metodo principale e semplie, ovvero *redirect(String).* Il seguente metodo serve per caricare un pagina partendo dal livello piu alto ovvero il suo controller.

public static function redirect($to)

{

header("Location: ".URL.$to);

exit;

}

### ViewLoader

La seguente classe mette a disposizione un solo metodo il quale utlizzo è di caricare le pagine e farle visualizzare all’utente. Oltre ad avere il ruolo di caricare le view, permette pure di passare un array indicizzato di variabili per poter nella view recuperare quest’ultime chiamandole come se fossero delle normali variabili, es. $esempio.

public static function load($template, $args = null)

{

if (gettype($args) == "array") {

foreach ($args as $name => $value) {

$$name = $value;

}

}

require \_\_DIR\_\_ . '/../views/'.$template.'.php';

}

## Pagina login

La pagina di login viene gestista dal controller Home, che di conseguenza utilizza il model Users. Nel controller Home troviamo principalmente i vari redirect alle pagine, la generazioni di errore, la scrittura su sessioni,... Però la parte che più spicca rispetto alle altre è la seguente:

$type = ($info["type"]==1)?Auth::ADMIN:Auth::USER;

Auth::auth($info["username"],$type);

if (session\_status() == PHP\_SESSION\_NONE) {

session\_start();

}

foreach ($info as $key => $value) {

$\_SESSION[$key] = $value;

}

if($type == Auth::ADMIN)

Application::redirect("adminpanel/index");

else if($type==Auth::USER)

Application::redirect("whitelistpanel/index");

Ovvero la parte di codice dove effettivamente vi è un cambiamento di pagina. Dopo aver eseguito tutti i controlli del caso vado a verificare di che tipo sia l’utente che sta eseguengo l’accesso e lo reindirizzo alla pagina che visualizzerà.

### Model Users

La classe Users, che fa parte del gruppo dei Model, in questo caso viene utilizza per controllare se un utente esiste, ha sbagliato password o non esiste proprio.

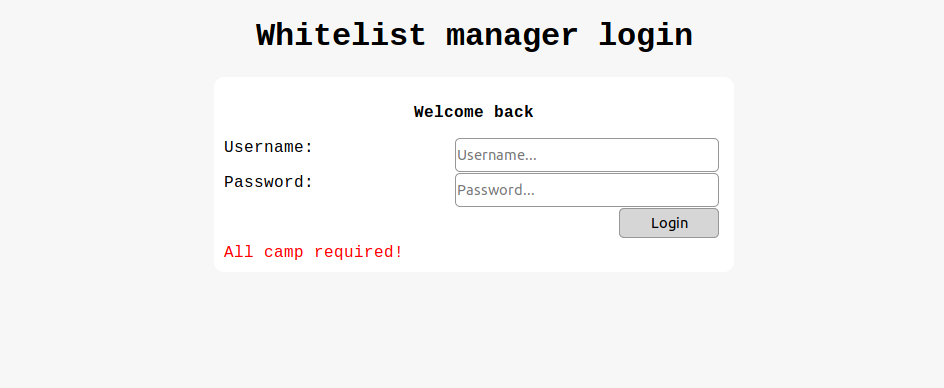
$login = Users::exist($username, $password);

if ($login == Users::LOGIN\_SUCCESS){ ... }

else if($login == Users::USERNAME\_NOT\_EXIST){ ... }

else if($login == Users::PASSWORD\_WRONG){ ... }

La pagina di login ha avuto molte modifiche siccome inizialmente l’avevo ideato con troppa grafica a confronto di quella richiesta. Modificando il tutto, il risultato finale è il seguente:



Come si vede la pagina presenta pochissimo css. Ho deciso di mettere il login al centro per una questione estetica mia.

La pagina è relizzata pricipalemente in html, ma presenta anche qualche parte in php, ad esempio, il messagio che cambia in base al tipo di errore che si presenta.

<span class="error">

<?php if (@isset($error)) if (is\_bool($error)) echo "All camp required!"; else echo $error ?>

</span>

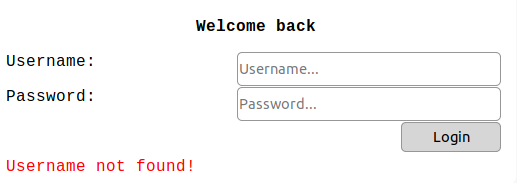
<span class="success">

<?php if (@isset($success)) echo $success; ?>

</span>

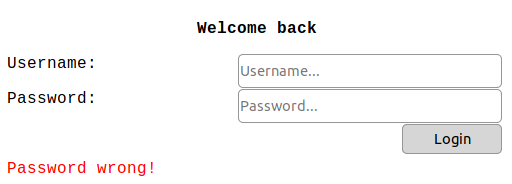
Il seguente codice come detto in precedenza serve per mostrare un messaggio di errore nel caso ci sia e per mostrare un messaggio di successo.

### Username not found



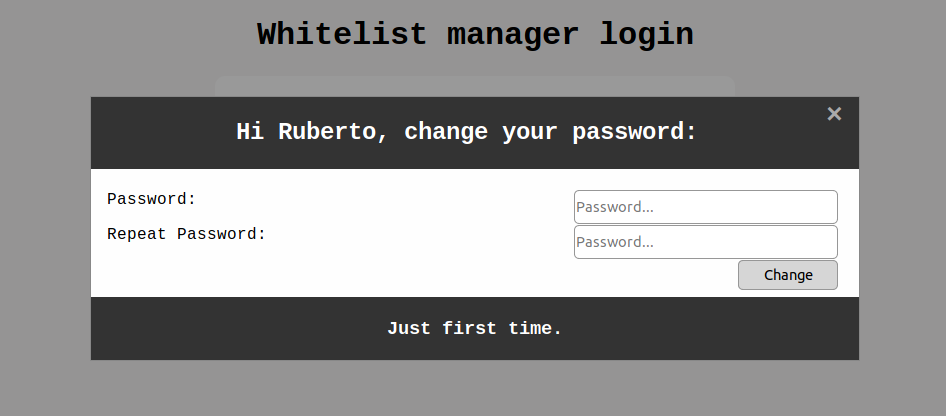
Messaggio mostrato solo nel caso che l’utente inserisca un username non valido oppure non trovare all’interno del database.

### Password wrong



Messaggio mostrato nel caso che l’utente inserisca un username corretto e presente all’interno ma con password diversa rispetto a quella del database.

### Change password



Quando un utente fa login per la prima volta gli viene mostrato il seguente modal dove gli viene imposto di cambiare la password. Per controllare che i dui campi siano uguali mi sono servito di un semplice controllo javascript, non ho pensato fosse necessario farlo anche php.

<div id="myModal" class="modal" <?php if (@isset($setpassword)) echo "style='display:block;'" ?>>

<div class="modal-content">

<div class="modal-header">

<span class="close">&times;</span>

<h2>

<?php if (@isset($surname))

echo "Hi $surname, change your password:"

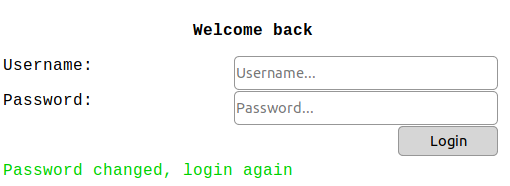
?>

</h2>

</div>

Nel caso la variabile setPassword fosse esistente allora vado a mostrare il modal.

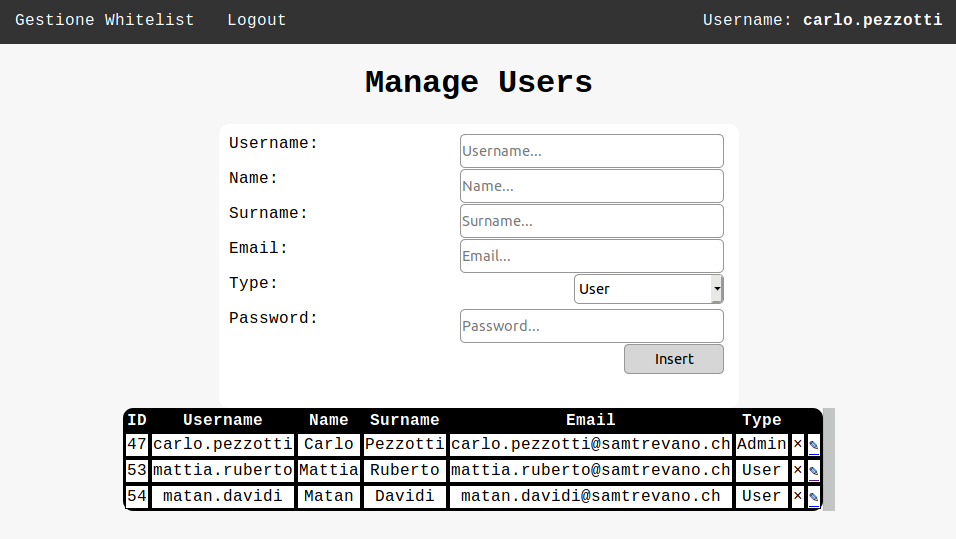
### Passoword changed



Alla fine di tutto vi è quest’ultimo stato della pagina dove viene mostrato un messagio per indicare all’utente che la password è stata modificata.

## Pagina amministrativa

La pagina amministrativa per gestire gli utenti che popolano il database. Ovvero vi è la possibilità di inserire, modificare oppure cancellare gli utenti nel database. Sotto richiesta del committente ogni nuovo utente inserito nel database de avere un password decisa dall’amministratore e modificata al primo login. La pagina si mostra nel seguente modo:



In alto a destra c’è l’username della persona che è collegata.



Non ha un vero e proprio senso, però a livello informativo era utile da mettere.

In alto a sinistra ci sono 2 tasti, uno che ti porta alla gestione della whitelist ed uno che ti permette di fare logout.



Questa nav bar si presenterà anche più avanti nella pagina di gestione della whitelist.

### Inserimento nuovo utente

L’inserimento di un nuovo utente si basa un form che di seguito manda una richiesta ad una pagina php tramite il protocollo POST.

if (!empty($\_POST["username"]) && !$error) {

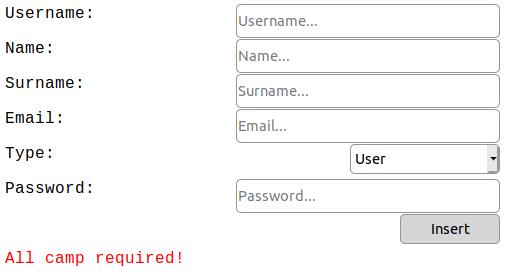
if (Validate::usernameValidate($\_POST["username"]) != Validate::VALIDATE\_FALIED) {

$username = $\_POST["username"];

}

} else $error = true;

Le seguenti righe servono per controllare che l’utente non abbia inserito dei dati non idonei al database. In caso quest’ultimi non dovessero rispettare i requisiti da me imposti segnalo il tutto mostrando un testo di errore nella pagina.



Dopo aver validato il tutto ed esser sicuri dei dati, li inserisco all’interno del database nel seguente modo:

$sql = "INSERT into user values(0,:username,:name,:surname,:email,:type,:password,1)";

$values = array(

'username' => $username,

'name' => $name,

'surname' => $surname,

'email' => $email,

'type' => $type,

'password' => hash('sha256', trim($password))

);

try {

Admin::doQuery($sql, $values);

}

### Tabella

Per stampare la tabella a schermo utilizzo un metodo del model Admin che mi ritorna una array indicizzato già pronto per essere stampato con tutto le maiuscole del caso.

use Models\Admin;

$result = Admin::getTable();

echo "<table class='tableR'>";

echo "<thead><tr>";

foreach ($result[0] as $key => $col) {

echo "<th>$key</th>";

}

### Aggiornamento utenti e rimozione

L’aggiornamento e la rimozione degli utenti si trova a destra di ogni riga.



echo "<td><a onclick='confirmDelete($ID,\"$username\")'><span class='operation'>&times;</span></a></td>";

echo "<td><a href='" . URL . "adminpanel/modify/$ID'><span class='operation'>&#9998;</span></a></td>";

La modifica è pressoché simile all’inserimento nel database, l’unica cosa che alla fine devo aggiungere una clausola *where.*

# Test

## Protocollo di test

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case:**  **Riferimento**: | TC-001  REQ-001 | **Nome:** |  |
| **Descrizione:** |  | | |
| **Prerequisiti:** |  | | |
| **Procedura:** |  | | |
| **Risultati attesi:** |  | | |

## Mancanze/limitazioni conosciute

# Consuntivo

# Conclusioni

## Sviluppi futuri

Si potrebbe aggiungere oltre alla whitelist anche un blacklist dei siti..

## Considerazioni personali

Mi è piaciuto molto come progetto perchè ho imparato a installare e configurare proxy squid, che a prima vista sembra complicato. Mi ha messo in molta difficoltà la parte di far eseguire a php un comando con i permessi di amministratore, ma grazie a un po’ di buona ricerca online e di logica ci sono riuscito. La stesura dei diari durante tutto il percorso del progetto mi è sembrata eccessiva anche se a volte mi sono servite tutte le informazione scritte nei diari in precedenza.

# Bibliografia

## Sitografia

[www.stackoverflow.com](http://www.stackoverflow.com)

[www.php.net](http://www.php.net)

(per altri siti vedi diari di lavoro)

# Allegati

Elenco degli allegati, esempio:

* Diari di lavoro
* Codici sorgente/documentazione macchine virtuali
* Istruzioni di installazione del prodotto (con credenziali di accesso)
* Eventuali guide utente / Manuali di utilizzo
* Mandato e/o Qdc
* Prodotto