CPT Meeting Room Scheduler

[1 Introduzione 4](#_Toc26898128)

[1.1 Informazioni sul progetto 4](#_Toc26898129)

[1.2 Abstract 4](#_Toc26898130)

[1.3 Scopo 4](#_Toc26898131)

[Analisi 5](#_Toc26898132)

[1.4 Analisi del dominio 5](#_Toc26898133)

[1.5 Analisi e specifica dei requisiti 5](#_Toc26898134)

[1.6 Use case 9](#_Toc26898135)

[1.7 Pianfiicazione 1](#_Toc26898136)

[1.7.1 Spiegazione diagramma Gantt 1](#_Toc26898137)

[1.8 Analisi dei mezzi 4](#_Toc26898138)

[1.8.1 Software 4](#_Toc26898139)

[1.8.2 Hardware 4](#_Toc26898140)

[2 Progettazione 5](#_Toc26898141)

[2.1 Design dell’architettura del sistema 5](#_Toc26898142)

[2.2 Design dei dati e database 6](#_Toc26898143)

[2.2.1 Schema logico 8](#_Toc26898144)

[2.2.2 Tipi di dati 8](#_Toc26898145)

[2.3 Design delle interfacce 11](#_Toc26898146)

[2.4 Design procedurale 13](#_Toc26898147)

[3 Implementazione 14](#_Toc26898148)

[3.1 Gestione permessi 14](#_Toc26898149)

[3.1.1 Classe PermissionModel e classe Permissions 14](#_Toc26898150)

[3.2 Pannello admin 15](#_Toc26898151)

[3.2.1 Gestione utenti 16](#_Toc26898152)

[3.2.2 Gestione prenotazioni 22](#_Toc26898153)

[3.3 Validazione dati 22](#_Toc26898154)

[3.4 Struttura API 22](#_Toc26898155)

[4 Test 22](#_Toc26898156)

[4.1 Protocollo di test 22](#_Toc26898157)

[4.2 Risultati test 23](#_Toc26898158)

[4.3 Mancanze/limitazioni conosciute 23](#_Toc26898159)

[5 Consuntivo 23](#_Toc26898160)

[6 Conclusioni 23](#_Toc26898161)

[6.1 Sviluppi futuri 23](#_Toc26898162)

[6.2 Considerazioni personali 23](#_Toc26898163)

[7 Bibliografia 23](#_Toc26898164)

[7.1 Bibliografia per articoli di riviste: 23](#_Toc26898165)

[7.2 Bibliografia per libri 23](#_Toc26898166)

[7.3 Sitografia 24](#_Toc26898167)

[8 Allegati 24](#_Toc26898168)

# Introduzione

## Informazioni sul progetto

Allievo coinvolto: Luca Di Bello

Classe: Informatica 4AC presso la Scuola di Arti e Mestieri a Trevano

Docenti responsabili: Fabrizio Valsangiacomo

Data inizio: 03 / 09 / 2019

Data fine: 20 / 12 / 2019

## Abstract

*Paper has been used for many centuries in the past and is still used for different purposes; with the passing of time we have realized that*

*it can become chaotic to the point of losing valuable information.*

*At the moment there are many meeting rooms that are reserved using an old piece of paper risking to blow plans*

*of months and months.*

*With this product you have a much cheaper, more ecological and safer substitute!*

*All the information is saved in a database and everything is accessible and modifiable through a*

*simple and intuitive web page, reachable from any device, whether mobile or not.*

## Scopo

Lo scopo di questo progetto è quello di implementare un sistema utile per gestire le prenotazioni dell’aula riunioni della sezione informatica. Il tutto sarà controllato da remoto tramite delle interfacce web. Verrà posto un monitor collegato ad un Raspberry Pi il quale permetterà di leggere quando la sala riunioni è occupata.

Questo progetto verrà anche utilizzato per prepararci all’esame di maturità che dovremo affrontare alla fine dell’anno.

## Analisi

## Analisi del dominio

Fino ad ora i professori utilizzavano un foglio di carta appeso alla porta con scritto chi, quando e per quanto tempo verrà utilizzata l’aula, dunque si cerca un modo per rendere tutto più chiaro e semplice utilizzando un sistema informatico.

## Analisi e specifica dei requisiti

|  |  |
| --- | --- |
| **ID: REQ-001** | |
| **Nome** | Creazione banca dati |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | - |
| **Sotto requisiti** | |
| **001** | Si necessita una tabella che tiene in memoria tutti gli utenti ed i permessi che hanno |
| **002** | Si necessita una tabella che tiene in memoria quali riservazioni sono state fatte |
| **003** | Il database dev’essere normalizzato fino al livello BCNF per semplificare modifiche future |
| **004** | Utente aggiuntivo con permessi limitati da utilizzare per leggere i dati dal sito web |

|  |  |
| --- | --- |
| **ID: REQ-002** | |
| **Nome** | Login con LDAP |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | Si necessitano i permessi per accedere al servizio di *active directory* di scuola |
| **Sotto requisiti** | |
| **001** | Pagina web con form che permette all’utente di inserire le proprie credenziali, quindi e-mail e password |
| **002** | Soltanto i professori possono accedere al servizio (…@edu.ti.ch) |
| **003** | Pagina responsive e funzionante su qualunque dispositivo (mobile e non) |

|  |  |
| --- | --- |
| **ID: REQ-003** | |
| **Nome** | Permessi degli utenti |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | - |
| **Sotto requisiti** | |
| **001** | Livello utente normale: può vedere ed inserire una riservazione. Inoltre, può eliminare delle riservazioni fittizie |
| **002** | Livello utente avanzato: ha gli stessi permessi di un utente normale ma può inserire e cancellare riservazioni di un altro utente |
| **003** | Livello utente amministratore: ha tutti i premessi disponibili |

|  |  |
| --- | --- |
| **ID: REQ-004** | |
| **Nome** | Pagina gestione utenti |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | - |
| **Sotto requisiti** | |
| **001** | Si necessita una pagina web dove un utente amministratore può creare, cancellare modificare agli utenti |
| **002** | L’amministratore deve poter creare degli utenti normali ed utenti avanzati (con permessi aggiuntivi) assegnando una password di default. |
| **003** | Al primo login di un utente gli deve essere chiesto di cambiare la password di default con una personale |
| **004** | L’amministratore deve poter cambiare i permessi di un utente in ogni momento. |
| **005** | Alla creazione di un nuovo utente la password generata verrà inviata per e-mail all’utente in questione |
| **006** | Ci deve sempre essere almeno un utente amministratore. |
| **006** | Non deve essere possibile la registrazione di un utente con un’e-mail esterna a quella scolastica (gmail, Sunrise, ..) |

|  |  |
| --- | --- |
| **ID: REQ-005** | |
| **Nome** | Gestione prenotazioni |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | - |
| **Sotto requisiti** | |
| **001** | Deve essere presente un form che permette di aggiungere una nuova prenotazione. I campi necessari sono: data, l’orario di inizio e di fine. Anche il nome, il cognome e la mail di chi fa la riservazione devono essere mostrati. Dev’essere presente un campo utile per aggiungere delle osservazioni. |
| **002** | Controllo sulla data ed ora: non devono essere passati e non dev’esserci già una prenotazione |
| **003** | L’orario della prenotazione dev’essere a blocchi di 15 minuti |
| **004** | Alla creazione e cancellazione di una riservazione verrà inviata un’e-mail di conferma al relativo utente |

|  |  |
| --- | --- |
| **ID: REQ-006** | |
| **Nome** | Schermo per visualizzazione prenotazioni |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | Verrà posto (tramite un supporto in alluminio) davanti all’aula riservata ai colloqui. |
| **Sotto requisiti** | |
| **001** | Non si dovrà vedere nessuna pagina web |
| **002** | Devono essere mostrate le date, l’orario di inizio e fine della riservazione, e le informazioni relative al docente che ha eseguito la riservazione (nome e cognome) |
| **003** | Deve mostrare le osservazioni se aggiunte dal docente durante la riservazione |
| **004** | Il sistema verrà controllato tramite un Raspberry Pi Model B fornito dalla scuola |

|  |  |
| --- | --- |
| **ID: REQ-007** | |
| **Nome** | Generazione di report |
| **Priorità** | 2 |
| **Versione** | 1.0 |
| **Note** | - |
| **Sotto requisiti** | |
| **001** | I report devono essere in formato PDF |
| **002** | I file di report devono essere generati automaticamente dalla pagina web |
| **003** | 3 tipi di report: giornalieri, settimanali e mensili. |

## Use case

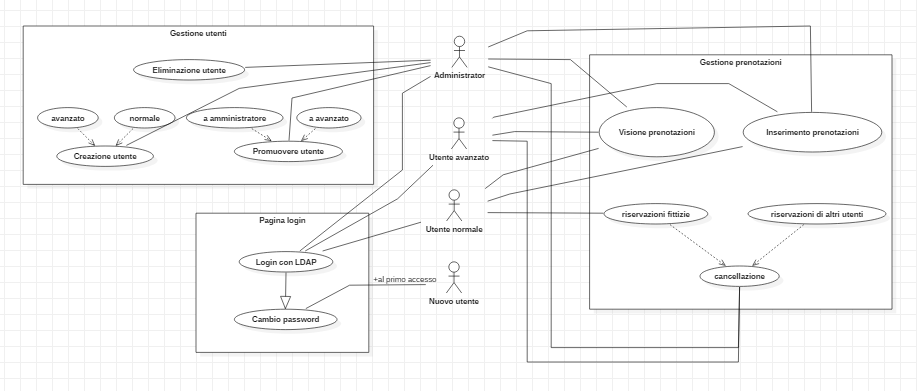


Figure 1 - Diagramma *Use Case*

Come si può vedere molto bene dall’immagine sovrastante ho deciso di dividere il diagramma in 3 macro-gruppi (gestione utenti, gestione prenotazioni e pagina login) per semplificare sia la lettura che la stesura del diagramma.

L’utente con i permessi di amministratore ha tutti i permessi sia nella gestione utenti sia nella gestione delle prenotazioni. L’utente avanzato invece ha tutti i permessi nella gestione delle prenotazioni però non può eliminare, creare o promuovere utenti.

L’utente normale invece può soltanto creare prenotazioni, visionare le prenotazioni fatte dagli altri professori ed eliminare le prenotazioni create da lui (non quelle di altri utenti).

Il “nuovo utente” invece come si può vedere, ha soltanto i permessi per cambiare la password. Questo perché al primo login di un utente il sistema richiede di cambiare la password di default con una password personale. Una volta cambiata la password l’utente viene promosso automaticamente ad “Utente normale”.

## Pianfiicazione

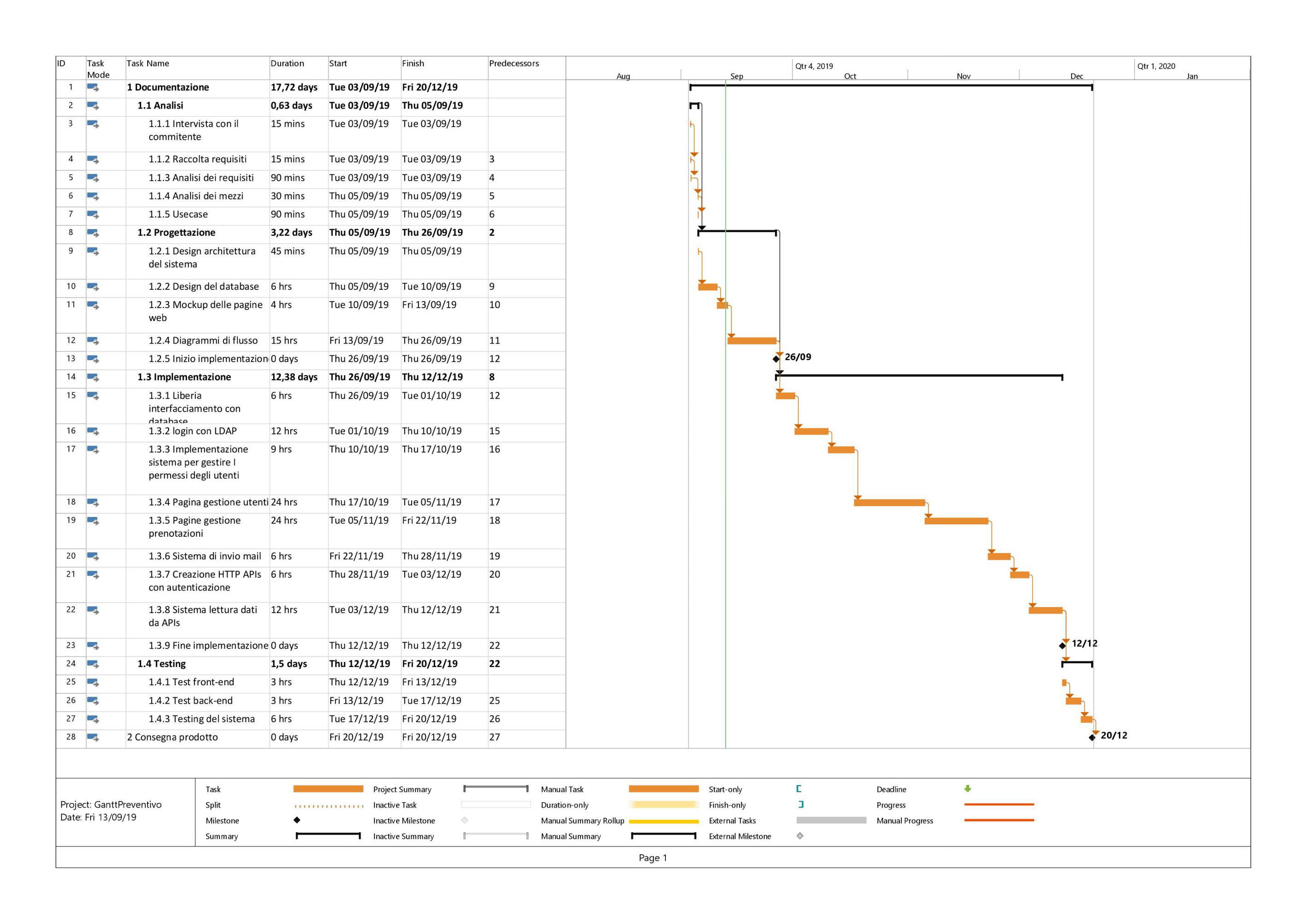


Figure 2 - Diagramma Gantt preventivo

### Spiegazione diagramma Gantt

Nei vari sotto capitoli spiegherò nello specifico cosa avverrà all’interno di ogni attività. Molte di queste sono state riprese dalla documentazione.

#### Analisi

Figure 3 - Gantt attività analisi

Ho deciso di suddividere il capitolo di analisi in 5 attività:

* Intervista con il committente
  + Intervista con il committente, utilizzata per chiarire i dubbi relativi alla consegna
* Raccolta dei requisiti
  + Raccolta dei requisiti tramite la lettura approfondita del QdC (Quaderno Dei Compiti)
* Analisi dei requisiti
  + Riflessione riguardo alle richieste raccolte, analizzandole e suddividendole in modo sensato
* Analisi dei mezzi
  + Riflessione riguardo ai mezzi da utilizzare per il progetto (sia Software che Hardware)
* Use case
  + Stesura del diagramma Use Case

#### Progettazione

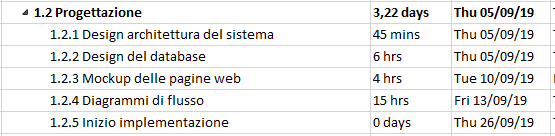


Figure 4 - Gantt attività progettazione

Ho deciso di suddividere il capitolo riguardante alla progettazione del sistema in altre 4 attività:

* Design architettura del sistema
  + Schema che mostra a livello grafico come i vari componenti del sistema interagiscono tra loro
* Design del database
  + Diagramma *Entity Relation* (ER) e schema logico che illustra la struttura del database
* Mockup delle pagine web
  + Sketch delle pagine web
* Diagrammi di flusso
  + Diagrammi di flusso che mostrano il funzionamento di ogni componente del sistema

#### Implementazione

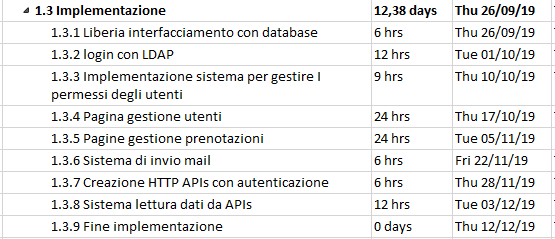


Figure 5 - Gantt attività implementazione

Ho deciso di suddividere il capitolo riguardante all’implementazione del sistema in 8 attività:

* Libreria interfacciamento con database
  + Implementazione di una libreria che permetterà la connessione e l’invio di query al database
* Login con LDAP
  + Sistema di login utilizzando LDAP.
* Implementazione sistema per gestire i permessi degli utenti
  + Implementazione di un sistema che controlla i permessi di un utente prima di eseguire qualsiasi operazione
* Pagina gestione utenti
  + Pagina che permette di gestire gli utenti come da requisiti
* Pagina gestione prenotazioni
  + Pagina che permette di gestire le prenotazioni come da requisiti
* Sistema di invio mail
  + Sistema che permette l’invio di mail di conferma ad un determinato utente
* Creazione http APIs con autenticazione
  + Creazione di Api http con autenticazione che permettono la lettura di dati del database. Esse verranno utilizzate dal Raspberry per mostrare i dati su schermo
* Sistema lettura dati da APIs
  + Sistema di lettura e formattazione dei dati utilizzando le API fornite dal sito web

#### Testing

Figure 6 - Gantt attività testing

Ho deciso di suddividere il capitolo riguardante al testing del sistema in 3 attività:

* Test front-end
  + Testing delle pagine web nella loro interezza
* Test back-end
  + Testing di tutto il back-end
* Testing del sistema
  + Testing approfondito dell’intero sistema

## Analisi dei mezzi

### Software

Per lo sviluppo di questo sistema ho utilizzato questi software:

* PHP 7.2.10: Linguaggio utilizzato per gestire la logica del sito da back-end
* Apache 2.4.27: Web server utilizzato per sviluppare il sito dalla mia macchina
* PhpMailer 6.0.7: Libreria PHP utilizzata per inviare mail agli utenti del sito.
* MDB 4.8.3: Acronimo di *Material Design for Bootstrap*. Esso è un framework front-end basato su *Boostrap 4* ma con una grafica che segue il concetto di casa Google chiamato *“Material Design”*.
* XDebug: Modulo aggiuntivo per PHP che fornisce di avere delle informazioni aggiuntive utili durante il debug
* PHPStorm
* MySQL Workbench 6.3: Applicativo che permette la gestione completa di uno o più database a livello grafico (tramite GUI)
* MeekroDB 2.3: Framework per database in PHP. Esso permette di semplificare l’iterazione tra applicazione e database MySQL

### Hardware

L’intero progetto è stato sviluppato sul mio laptop personale. Queste sono le specifiche:

* Nome: Asus ROG gl702vm
* Ram: 16GB
* CPU: Intel Core i7-7700HQ 2.80GHz
* Gpu: Nvidia GTX 1060 6GB
* Durante lo sviluppo di questo progetto ho utilizzato i seguenti sistemi operativi (OS):
  + Windows 10 Home versione 1809
  + PopOS! 18.04 LTS/19.10

Per controllare il monitor invece ho utilizzato un Raspberry Pi Model B, queste sono le sue specifiche:

* Nome: Raspberry Pi Model B
* Ram: 1GB LPDDR2 (900 MHz)
* CPU: 4× ARM Cortex-A53, 1.2GHz
* OS: Raspbian

Come monitor ho utilizzato un HP Compaq LA1051g fornito dalla scuola.

# Progettazione

## Design dell’architettura del sistema

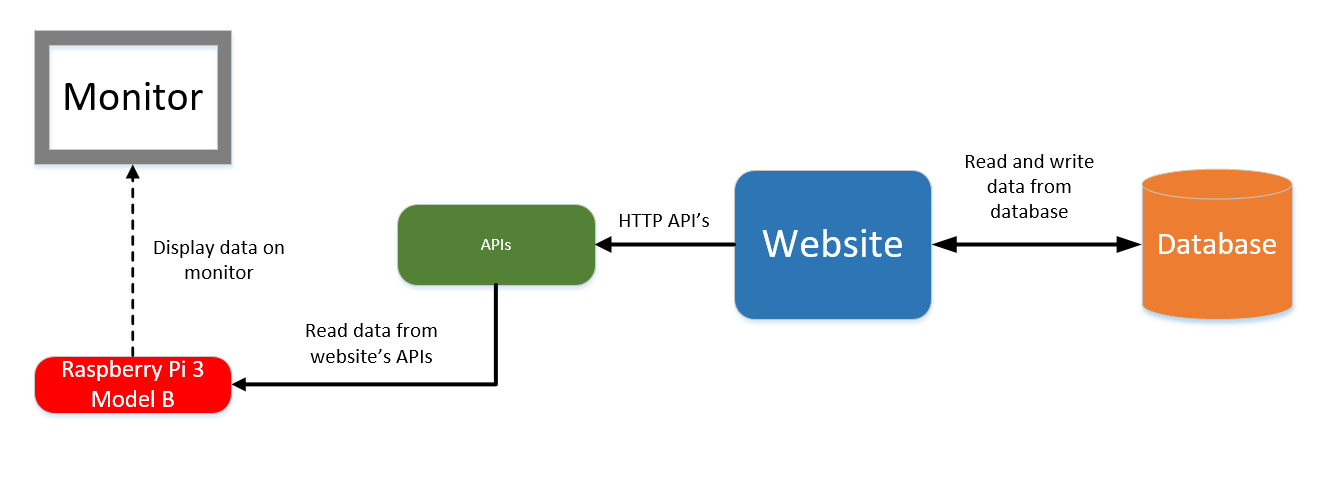


Figure 7 - Struttura del programma

Lo schema sovrastante è composto da 8 parti principali collegate tra loro in questo modo: Il sito web si interfaccerà con il database per la lettura e scrittura dei dati e fornirà delle API HTTP che verranno utilizzate dal Raspberry Pi per leggere i dati (sotto forma di JSON) i quali verranno parsati (formattati) e mostrati all’utente finale tramite un Monitor.

## Design dei dati e database

Questo è il diagramma *entity relation* del database della mia applicazione:

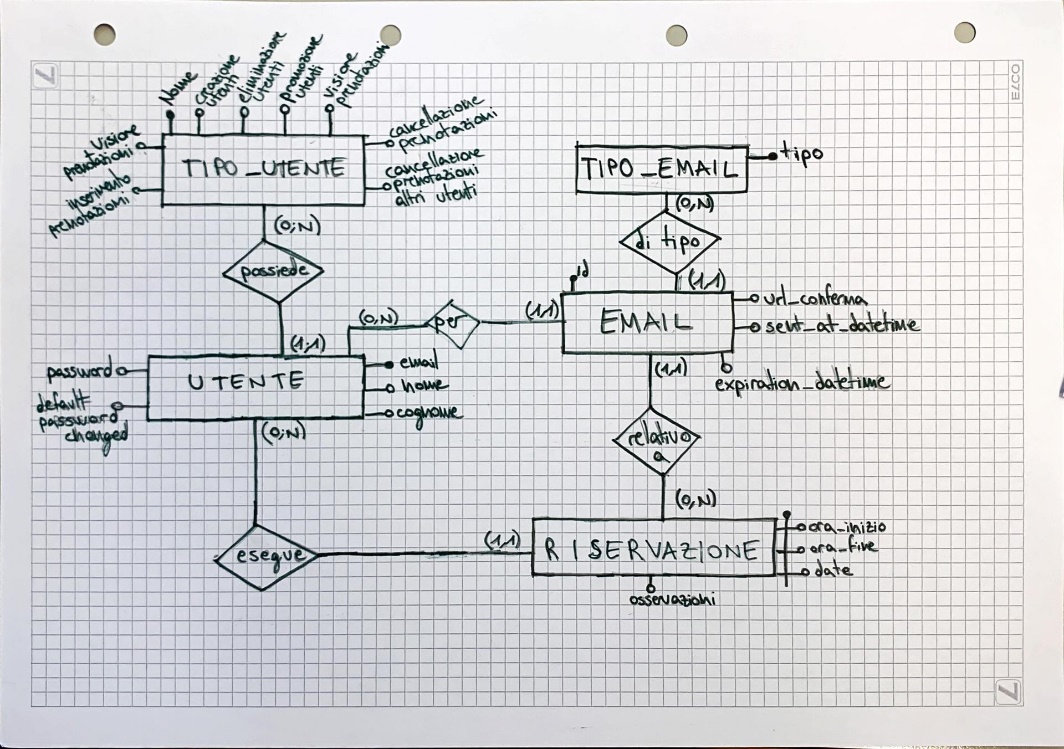


Figure 8 - Diagramma ER database

Lo schema è composto di 5 entità principali, di quali due non sono altro che generalizzazioni (*tipo\_utente* e *tipo\_email*).

L’entità *tipo\_utente* possiede molti attributi; ognuno di questi attributi è un permesso dell’utente. Ho deciso di farlo in questo modo per rendere l’applicazione “aggiornabile”: se in futuro si avrà la necessità di creare un nuovo tipo di utente in questa maniera non ci sarà la necessità di cambiare codice ma bisognerà soltanto aggiungere un record all’interno della tabella. Mi sono ispirato al funzionamento dei permessi di MySQL.

L’entità *utente* come si può intuire conterrà i dati dell’utente che utilizzerà l’applicazione tra questi anche il tipo di permessi che ha (indicato tramite l’associazione intitolata “*possiede”*). L’attributo *default\_password\_changed* servirà per tenere in memoria se l’utente ha cambiato la password al primo accesso o meno.

L’entità *riservazione* conterrà tutti i dati utili per l’identificazione della riservazione, quindi chi ha riservato e quando ha riservato.

L’entità *email* è collegata a ben 3 entità: utente, riservazione e *tipo\_email*. Questo perché un’e-mail è inviata ad un *utente* preciso nello specifico ed è riguardante una specifica *riservazione*. L’entità *tipo\_mail* è utilizzata per specificare se è un’e-mail di conferma di eliminazione o di creazione.

Gli attributi dell’entità *mail* servono per tenere in memoria l’url di conferma e il tempo di vita dell’url (24 ore solitamente).

### Schema logico

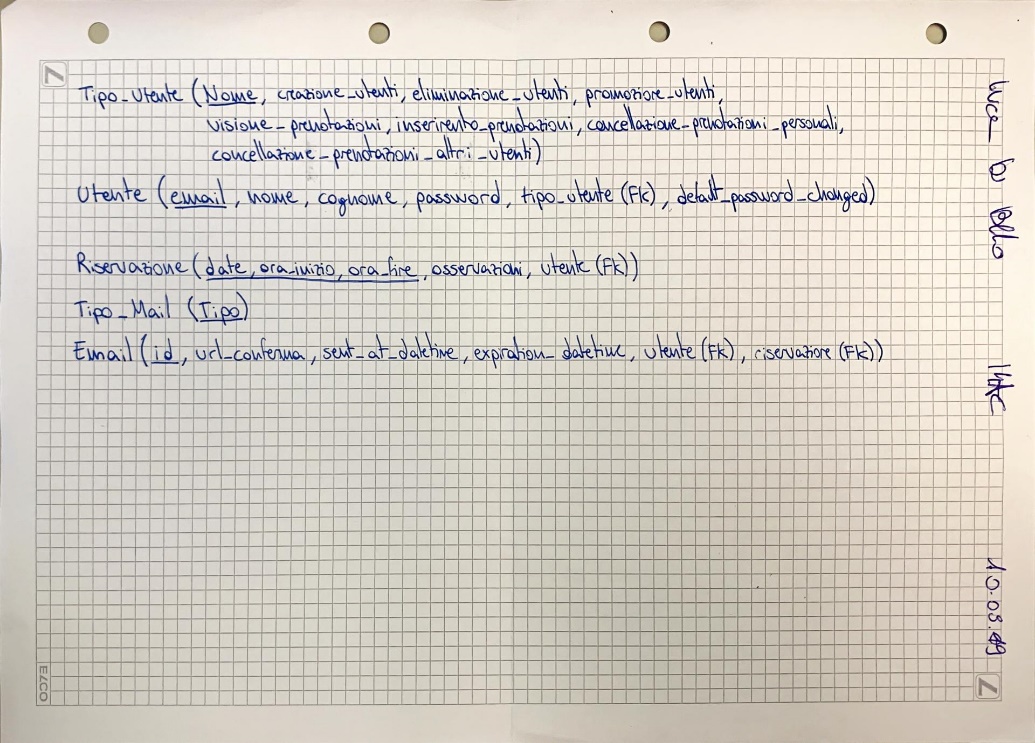
Questo è lo schema logico del database prodotto tramite il diagramma ER disegnato in precedenza:

Figure 9 - Schema logico database

### Tipi di dati

|  |  |
| --- | --- |
| Tipo\_Utente | |
| Nome attributo | Tipo attributo |
| Nome | *Primary Key, Varchar(50)* |
| Creazione\_utenti | *Bit, default 0* |
| Eliminazione\_utenti | *Bit, default 0* |
| Promozione\_utenti | *Bit, default 0* |
| Visione\_prenotazioni | *Bit, default 0* |
| Inserimento\_prenotazioni | *Bit, default 0* |
| Cancellazione\_prenotazioni\_personali | *Bit, default 0* |
| Cancellazione\_prenotazioni\_altri\_utenti | *Bit, default 0* |

|  |  |
| --- | --- |
| Utente | |
| Nome attributo | Tipo attributo |
| Email | *Primary Key, Varchar(255)* |
| Password | *Varchar(255)* |
| Nome | *Varchar(100)* |
| Cognome | *Varchar(100)* |
| Tipo\_utente | *Foreign Key, Varchar(50)* |
| Default\_password\_changed | *Bit, default 0* |

|  |  |
| --- | --- |
| Riservazione | |
| Nome attributo | Tipo attributo |
| Id | *Primary Key, Int(11), Auto Increment* |
| Date | *Primary Key, Datetime* |
| Ora\_inizio | *Time* |
| Ora\_fine | *Time* |
| Osservazioni | *Varchar(512)* |
| Utente | *Foreign Key, varchar(255)* |

|  |  |
| --- | --- |
| Tipo\_mail | |
| Nome attributo | Tipo attributo |
| Nome | *Primary Key, Varchar(50)* |

|  |  |
| --- | --- |
| Email | |
| Nome attributo | Tipo attributo |
| Id | *Primary Key, Int(11), Auto Increment* |
| Url\_conferma | *Varchar(512)* |
| Sent\_at\_datetime | *Datetime* |
| Exipiration\_datetime | *Datetime* |
| Utente | *Foreign Key, varchar(255)* |
| Riservazione | *Foreign Key, Int(11)* |

ERRORE: riservazione non ha un ID, dovrei esportare 3 foreign key 🡪 Mettere apposto lo schema logico ed ER

## Design delle interfacce

ho iniziato a fare i mockup delle pagine web utilizzando un sito web chiamato

Ho realizzato 4 mockup utilizzato un sito web chiamato *mockflow.com:* pagina di login, pagina gestione utenti, pagina gestione prenotazioni e pagina visualizzazione prenotazioni (che verrà creata dal raspberry Pi).

Questo è il mockup che raffigura la pagina di gestione degli utenti:

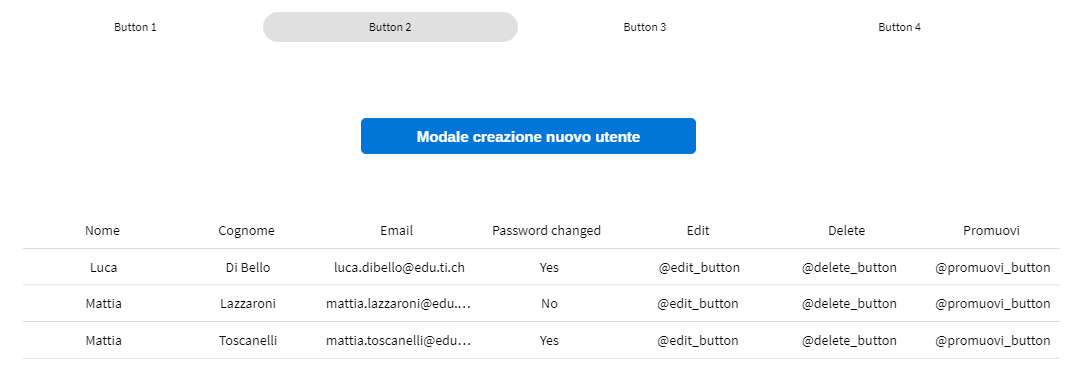


Figure 10 - Pagina gestione utenti

Questa è il mockup che raffigura la pagina di gestione delle prenotazioni:



Figure 11 - Pagina gestione prenotazioni

Questo è il mockup che raffigura la pagina di login, utilizzata per accedere al sito web:

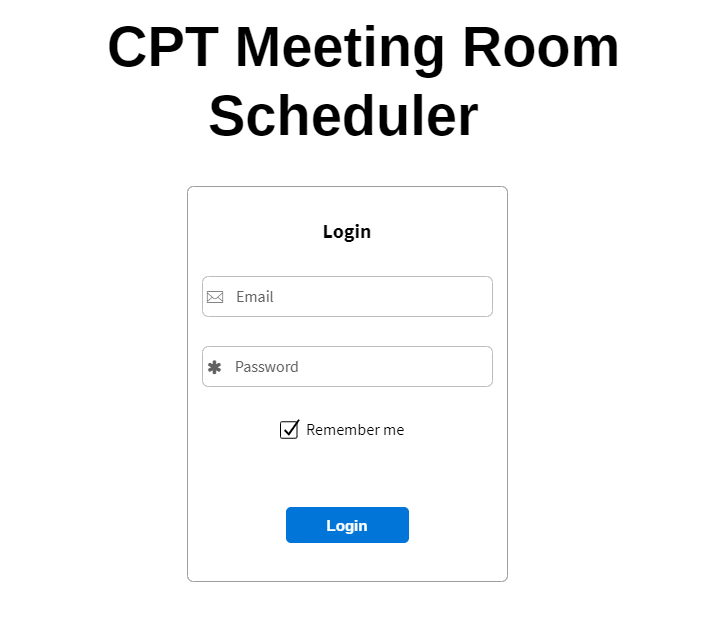


Figure 12 - Pagina di login

Ed infine questo è il mockup che raffigura la pagina di visualizzazione delle prenotazioni che verrà mostrata sul monitor:



Figure 13 - Visualizzazione prenotazioni su monitor

## Design procedurale

Descrive i concetti dettagliati dell’architettura/sviluppo utilizzando ad esempio:

* Diagrammi di flusso e Nassi.
* Tabelle.
* Classi e metodi.
* Tabelle di routing
* Diritti di accesso a condivisioni …

Questi documenti permetteranno di rappresentare i dettagli procedurali per la realizzazione del prodotto.

# Implementazione

## Gestione permessi

INSERIRE SCHEMA CHE MOSTRA TUTTI I POSSIBILI PERMESSI, SPIEGANDO COSA IMPLICA AVERLO ATTIVO O MENO

La gestione dei permessi è stata implementando utilizzando 2 classi principali ed una di supporto:

* Classe *PermissionModel*
* Classe *Permissions*
* Classe *PermissionManager* (classe di supporto)

Nei capitoli successivi sarà descritta ogni singola classe, spiegando sia la sua struttura (metodi e/o attributi) sia l’utilizzo di essa all’interno del sistema.

Questa è la struttura delle tre classi:

### Classe PermissionModel e classe Permissions

La classe PermissionModel contiene svariate funzioni utilizzate per la lettura dei permessi salvati all’interno del database. Queste classi sono fondamentali per la gestione dei permessi del sistema. Nei vari sottocapitoli successivi andrò a spiegare l’utilizzo ed il funzionamento di tutti i metodi della classe aggiungendo degli spezzoni di codice nelle parti critiche e/o di difficile comprensione.

#### getLocalPermission

Metodo wrapper che permette di caricare i permessi di un utente tramite la chiave del permesso inserite come parametro nel costruttore (esempio di chiave: *admin, user)*. Questo viene fatto leggendo i dati contenuti nella tabella *tipo\_utente* presente all’interno del database *cptmrs*. Tutti i dati letti vengono parsati ed inseriti all’interno di un oggetto *Permission* per rendere più semplice i controlli su cosa può o non può fare un determinato utente.

Questo metodo viene utilizzato dal sistema

Definisco questo metodo *wrapper* dato che non fa altro che richiamare il metodo *read\_data\_from\_database* utilizzando come parametro la chiave inserita nel costruttore e passare il risultato al metodo *parse\_data*. Quest’ultimo metodo non fa altro che fare un parsing dei dati letti dal database inserendoli in un oggetto di tipo *Permission*. Ecco il codice del metodo:

*// Permissions for local users*

**public** **function** getLocalPermissions(): Permissions{

$data = $this->read\_from\_database($this->PERM\_KEY);

return $this->parse\_data($data, $this->PERM\_KEY);

}

#### getLdapPermission

Metodo wrapper che permette di caricare i permessi di default di un utente che effettua il login utilizzando il protocollo *LDAP*. Questo viene fatto utilizzando sempre il metodo *read\_data\_from\_database* ma passando come chiave una definita nella classe *config.php.* Questa costante è chiamata ‘DEFAULT\_USER\_PERMISSION\_GROUP’ ed ha come valore di default *user*.

Questo metodo viene richiamato quando il sistema rileva un login utilizzando *LDAP*.

Proprio come per *getLocalPermission* anche questo metodo è definito *wrapper* proprio perché non fa altro che chiamare il metodo *read\_data\_from\_database* utilizzando la chiave salvata all’interno della costante *DEFAULT\_USER\_PERMISSION\_GROUP.* I dati ritornati dal metodo vengono passati nuovamente al metodo *parse\_data* che genera un oggetto *Permissions* utilizzandoli.

#### getUserPermissionGroup

Questo metodo è utilizzato per ricavare il nome del permesso assegnato ad un utente, il quale username è passato come parametro al metodo, se per qualche motivo non viene trovato nessun permesso il metodo ritorna il valore salvato all’interno della costante *DEFAULT\_USER\_PERMISSION\_GROUP* (che di default il valore “user”).

#### read\_data\_from\_database

Quest’ultimo metodo è sicuramente il fulcro di entrambe le classi. Esso è una funzione privata e viene utilizzato per leggere i dati relativi ad un permesso.

Si passa come parametro il nome del permesso (es: admin, user, ...) ed il metodo ritorna un array che contiene tutti i dati relativi ad esso.

Questo array si può passare al metodo ***parse\_data*** per convertire l’array in un oggetto *Permissions.*

### Controllo dei permessi

Per certe funzioni c’è la necessità di controllare i permessi dell’utente che esegue la richiesta. Questo si può fare utilizzando la classe *PermissionManager*, la quale non è altro che una classe *wrapper* che permette di leggere i permessi dell’utente. Una volta letti i permessi è possibile fare i vari controlli di cosa può e non può fare l’utente. Per esempio questo è un pezzo di codice che controlla se l’utente ha i permessi o meno di eliminarne un altro:

if (PermissionManager::getPermissions()->canEliminareUtenti()) {

$result = UserModel::delete($username);

if (is\_array($result)) {

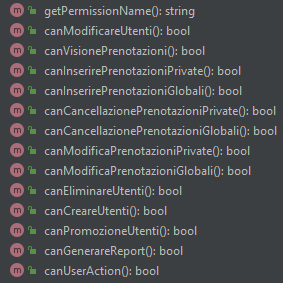
$GLOBALS["NOTIFIER"]->add\_all($result);

}

} else {

$GLOBALS["NOTIFIER"]->add("Non hai i permessi necessari per eliminare gli utenti");

}

Questi sono tutti i controlli disponibili:

## Pannello admin

L’applicazione presenta un pannello admin che permette la completa gestione degli utenti e delle prenotazioni. Esso è accessibile soltanto agli utenti con gli utenti amministratori (permessi di tipo “*admin”*).

Quando si accede al controller admin (quindi quando si digita l’url “..../admin”) il sistema controlla se i permessi dell’utente siano corretti. Se l’utente dispone dei permessi necessari il sistema carica la pagina utilizzando la libreria *ViewLoader*, altrimenti viene rimandato alla homepage.

Il pannello admin presenta una logica che permette di nascondere le funzionalità dove l’utente non ha permessi: Se un utente non ha nessun permesso per la modifica degli utenti, la pagina oltre a diventare inaccessibile, sparisce il link dalla barra di navigazione. Esegue la stessa cosa per la gestione delle prenotazioni.

Se invece un utente (per esempio) possiede i permessi per leggere e modificare gli utenti presenti nel sistema ma non ha quelli necessari per la creazione e l’eliminazione di essi i due tasti relativi a queste azioni vengono nascosti automaticamente.

Ho scelto di sviluppare questo sistema per rendere invisibili a determinati utenti determinate azioni e quindi limitare possibili problemi di sicurezza e *bugs,* i quali, se a livello di pannello admin, potrebbero risultare molto dannosi e rischiano di compromettere il corretto funzionamento dell’applicazione.

### Gestione utenti

Il pannello admin presenta una sezione dedicata per la gestione di ogni singolo utente registrato nel database locale dell’applicazione. All’interno di questa sezione si può eliminare, modificare, creare e promuovere utenti. Come ho detto in precedenza, questa pagina è dinamica. Essa cambia a seconda dei permessi dell’utente connesso in quel momento (vedi capitolo 3.2: *Pannello admin*).

Se l’utente entra nel pannello admin ma non dispone nessun permesso relativo alla gestione degli utenti il collegamento alla pagina, il quale solitamente è nella barra di navigazione in cima allo schermo, viene nascosto. Se l’utente prova ad accedere alla pagina il sistema mostra una pagina di errore che fa capire all’utente di non avere i permessi necessari per eseguire alcuna operazione sugli utenti.

Nei vari sottocapitoli successivi spiegherò ogni singola sezione della pagina, andando a spiegare sia il perché di certe scelte di design/strutturali che il come ogni componente opera con il sistema.

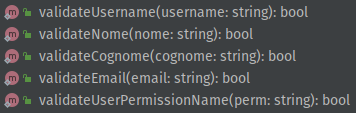
#### UserValidator

Ho deciso di descrivere per primo il sistema di validazione dati dato che viene utilizzato da le funzioni più importanti/critiche della pagina: aggiunta e modifica degli utenti.

Ritengo che questa classe sia la più importante tra quelle utilizzate a livello di gestione degli utenti. Essa permette e controlla che i dati siano sempre corretti e precisi, senza permettere nessun tipo di attacco su di essi: XSS Injection (<https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>) e SQL Injection (<https://www.owasp.org/index.php/SQL_Injection>).

Questi attacchi potrebbero, se utilizzati da un utente malintenzionato, compromettere il corretto funzionamento del sistema e mettere a rischio i dati degli utenti.

La classe *UserValidator* fornisce metodi statici che permettono di validare ogni dato relativo agli utenti*.* Questa classe infatti viene utilizzata per controllare i dati prima dell’inserimento/modifica di quelli presenti nel database. Questa è la sua struttura:

Spiegazione dettagliata del funzionamento di ogni metodo di validazione:

* *validateUsername*

Questo metodo riceve una stringa (username da validare) come parametro. Questa stringa viene controllata utilizzando la seguente *regular expression* (RegEx): /^[a-zA-Z0-9.]{1,255}$/

Questo RegEx controlla che la stringa sia alfanumerica (lettere e numeri) e permette di contenere i punti. Ho deciso di permettere i punti all’interno degli *username* per permettere la creazione di nomi utenti di questo tipo: *luca.dibello, pinco.pallino, ecc.*

Questa espressione regolare controlla anche che la lunghezza della stringa sia compresa tra 1 e 255 caratteri.

* *validateNome*

Questo metodo riceve una stringa (*nome* da validare) come parametro. Questa stringa viene controllata utilizzando il metodo *ctype\_apha* già presente in PHP. Questo è il codice che gestisce la validazione del nome:

setlocale(LC\_ALL, 'it\_CH.utf8');

return ctype\_alpha($nome);

La prima stringa di codice imposta il tipo di caratteri da validare. È molto importante impostare i caratteri per la gestione degli accenti: è, ö, é, ì, ecc.

La seconda stringa invece effettua la chiamata al metodo che va a fare la vera e propria validazione.

* *validateCognome*

Questo metodo riceve una stringa (cognome da validare) come parametro. Questa stringa viene controllata utilizzando lo stesso metodo della validazione del nome (*validateNome*) ma con una piccola modifica nel codice, la quale permette di avere dei cognomi composti (es: Di Bello, De Maria, ...)

Questa è la stringa che è stata modificata:

return ctype\_alpha(str\_replace(" " ,"", $cognome));

Come si può vedere dal codice il cognome viene passato nel metodo *str\_replace* il quale rimuoverà tutti gli spazi contenuti nel cognome. Quindi il cognome *Di Bello* verrà validato come se fosse il cognome *DiBello.*

Ho scelto di utilizzare questo modo per la gestione dei cognomi composti dato che il carattere spazio non può provocare nessun danno/problema di sicurezza nel sistema.

* *validateEmail*

Questo metodo riceve una stringa (email da validare) come parametro. Questa stringa viene poi controllata in due *step*:

1. Controllo formattazione email (controllo classico)
2. Controllo dominio

Il primo step controlla il formato della mail utilizzando un metodo già presente in PHP chiamato *filter\_var* il quale, se impostato con un flag, permette di controllare se un email è formattata correttamente o meno.

Il secondo step invece controlla se il dominio dell’email da validare (quindi la parte dopo la ‘@’: [luca.dibello@**samtrevano.ch**](mailto:luca.dibello@samtrevano.ch))) è uguale a quello impostato nella configurazione dell’applicativo (file di config: *config/config.php*).

Se entrambi i controlli vengono passati l’email è valida, altrimenti ritornerà *false*.

* *validateUserPermissionName*

Questo metodo riceve il nome di un gruppo di permessi (*es: admin, user, ...)* e controlla se effettivamente è un gruppo esistente all’interno del sistema o meno. Questo lo fa appoggiandosi al metodo *getUniquePermissionTypes* della classe *PermissionModel.*

Questo metodo (come dice anche il suo nome) ritorna una lista che contiene i nomi di tutti i gruppi presenti nel sistema.

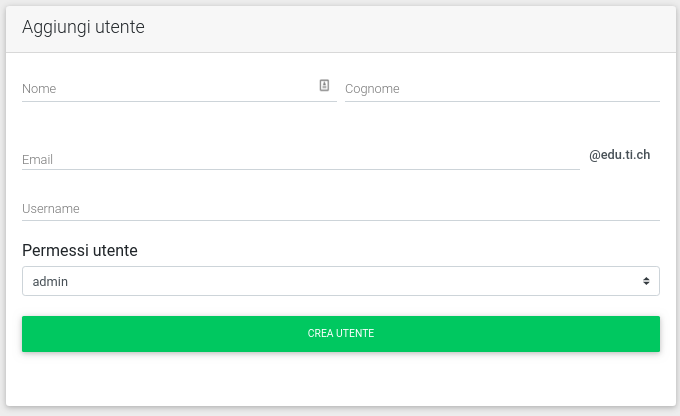
Una volta aver interrogato il database e ricevuta la lista, il metodo controlla se il nome passato come parametro è contenuto nella lista:

return in\_array($perm, PermissionModel::getUniquePermissionTypes());

Se il nome è contenuto significa che è un nome valido, altrimenti il metodo ritornerà *false.*

#### Aggiunta utenti

Questa sezione della pagina permette l’inserimento di un nuovo utente tramite un modulo (*form*). Questo è un piccolo screenshot che mostra com’è strutturato e che dati necessita:

Come si può vedere dall’immagine il *form* richiede all’amministratore di inserire il proprio nome, cognome la parte iniziale della mail (p.e: [**luca.dibello**@edu.ti.ch)](mailto:luca.dibello@edu.ti.ch)) ed i permessi che avrà l’utente all’interno dell’applicazione.

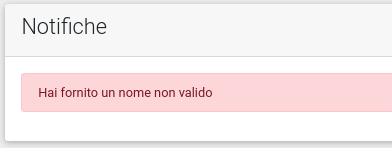
I permessi disponibili vengono caricati automaticamente andando a leggere i record presenti nella tabella *tipo\_utente* del database *cptmrs*. Questo viene fatto utilizzando la classe *PermissionModel* (vedi capitolo 3.1.1).

Se, per esempio, l’utente *luca.dibello* non dispone dei permessi necessari per la creazione di nuovi utenti l’intero form necessario per la creazione dell’utente viene nascosto.

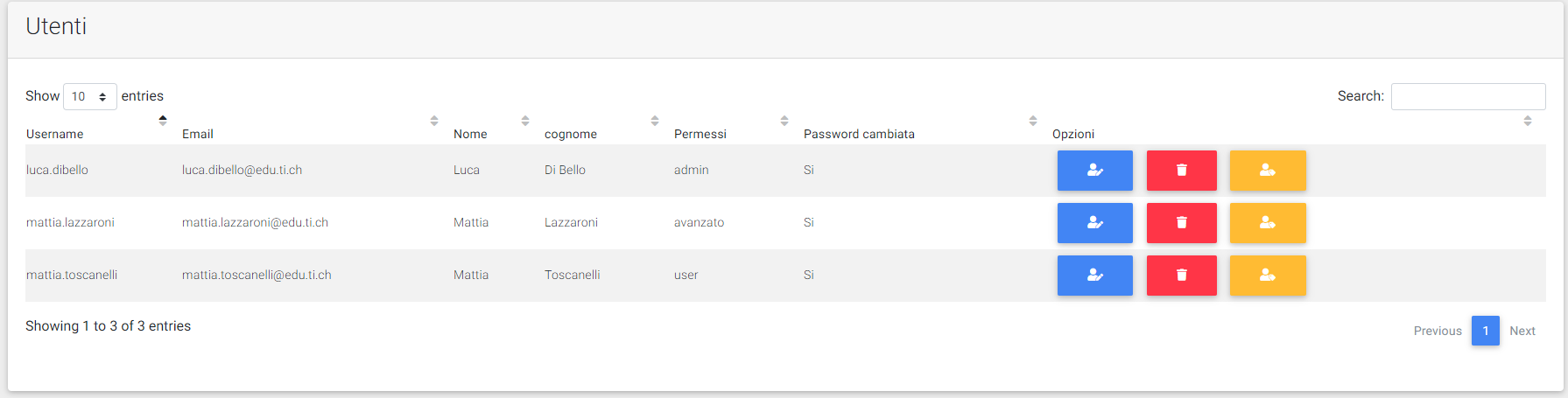
I dati vengono controllati da back-end tramite la classe *UserValidator,* la quale è stata creata ad hoc proprio per la gestione degli utenti.

Se i dati inseriti sono validi, essi vengono inseriti nel database e quindi l’utente viene creato correttamente. Se invece i dati risultano non validi l’operazione relativa alla creazione dell’utente viene annullata e viene mostrato un errore nella sezione *Notifiche* posta in cima alla pagina.

Questo è un esempio di notifica la quale è stata generata inserendo un nome non valido:

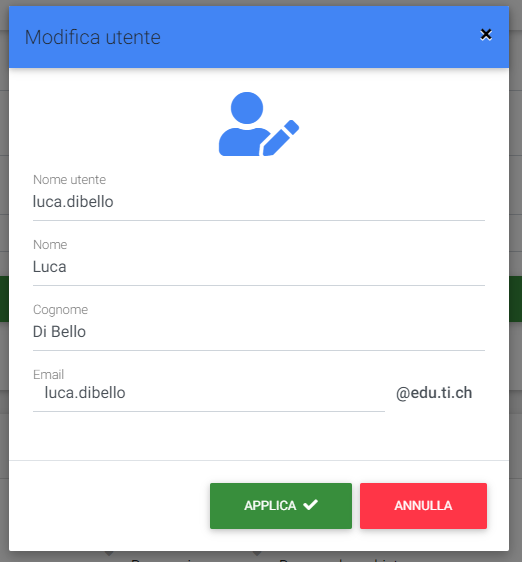


#### Visione utenti locali

In questa sezione si possono eseguire molteplici operazioni sugli utenti locali presenti nel sistema. Gli utenti vengono mostrati in una tabella generata dinamicamente da PHP, la quale (tramite la libreria JavaScript chiamata DataTables) viene resa una tabella “avanzata” che permette l’esecuzione di filtri ed ordinamento alfabetico. Ecco come vengono mostrati i dati al suo interno:

L’ultima colonna della tabelle non contiene dati sugli utenti ma dei pulsati per eseguire delle operazioni sull’utente desiderato, ecco le loro funzioni:

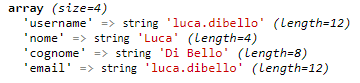
**Modifica utente**

Il primo pulsante (quello blu) serve per modificare i dati dell’utente (quindi username, email, nome o cognome). Se viene cliccato viene mostrato a schermo un modale che permette la modifica delle varie informazioni:

(Nota: I dati vengono inseriti automaticamente all’interno del form tramite un piccolo script JavaScript)

Cliccando sul pulsante“applica” presente nel modale viene eseguita una richiesta POST alle API specifiche per la modifica degli utenti, raggiungibili tramite questo link: *api/user/update/<username\_utente>*

Per esempio, se voglio modificare l’utente *luca.dibello* devo inviare i dati a questo link: <http://example.com/api/user/update/luca.dibello>

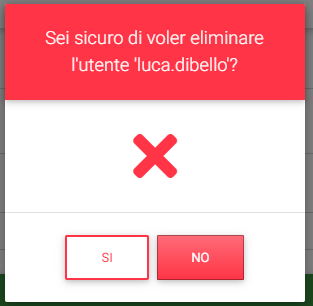
I dati vengono formattati ed inviati con questa semplice struttura:

Se le API riescono a validare correttamente i dati secondo i canoni prestabiliti (vedi capitolo *validazione dati* per ulteriori informazioni*)* e l’utente che esegue la richiesta ha i permessi necessari per la modifica degli utenti, i dati vengono passati alla classe *model* chiamata *UserModel* (metodo *update*) che si occupa di aggiornare i dati all’interno del database MySQL.

Cliccando sul pulsate “Annulla” il modale viene chiuso e vengono puliti tutti i campi.

**Eliminazione utente**

Cliccando il pulsante rosso viene aperto invece un modale di conferma di eliminazione. Esso si presenta così all’utente:



Ho deciso di inserire un modale di conferma di eliminazione dell’utente in quanto, se viene eliminato l’utente, anche tutte le sue prenotazioni vengono eliminate a cascata per non avere dati orfani all’interno del database.

Cliccando su *“*Si*”*, viene eseguita una richiesta alle API specifiche per l’eliminazione degli utenti, richiamabili tramite questo link: api/user/delete/<username\_utente>

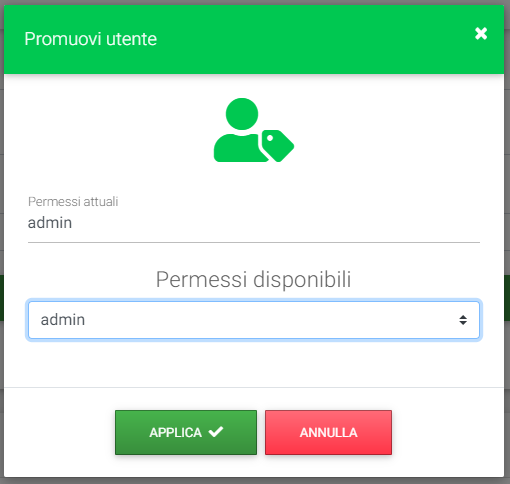
Per esempio se si vuole eliminare l’utente “luca.dibello” devo richiamare le API tramite questo link:  
<http://example.com/api/user/update/luca.dibello>

Se le API riescono a validare correttamente i dati secondo i canoni prestabiliti (vedi capitolo *validazione dati* per ulteriori informazioni*)* e l’utente che esegue la richiesta ha i permessi necessari per l’eliminazione degli utenti, i dati vengono passati alla classe *model* chiamata *UserModel* (metodo *delete*) che si occupa di aggiornare i dati all’interno del database MySQL.

Se invece si clicca sul pulsante “No” l’operazione viene annullata e quindi i dati non vengono eliminati.

**Promozione utente**

Cliccando l’ultimo pulsante (quello color ambra) viene aperto un modale che permette di cambiare i permessi ad un determinato utente. Ecco come si presenta l’interfaccia all’utente:



Esso mostra i permessi attuali dell’utente in un input di tipo testo *read-only* (non modificabile) ed un menù a tendina dove si può scegliere il nuovo permesso da applicare all’utente. I permessi all’interno di questo menù a tendina vengono caricati dinamicamente dal database tramite la classe *model* chiamata *PermissionModel*.

Cliccando sul tasto applica viene inviata una richiesta POST alle API che si occupano del cambio dei permessi di un utente. Esse sono raggiungibili tramite questo link: api/user/promote/<username>

Per esempio, se voglio cambiare i permessi all’utente *luca.dibello* devo inviare i dati a questo link:  
<http://example.com/api/user/promote/luca.dibello>

I dati inviati nella richiesta vanno formattati in questo modo (esempio cambio di permessi *admin*):

Se le API riescono a validare correttamente i dati secondo i canoni prestabiliti (vedi capitolo *validazione dati* per ulteriori informazioni*)* e l’utente che esegue la richiesta ha i permessi necessari per la promozione degli utenti, i dati vengono passati alla classe *model* chiamata *UserModel* (metodo *promote*) che si occupa di aggiornare i campo relativo al tipo di permessi assegnato all’utente all’interno del database MySQL.

Se invece si clicca su “*Annulla”* l’operazione viene annullata e nessun dato viene modificato.

(Nota: dato che i permessi vengono caricati al login dell’utente, se si vuole cambiare i permessi all’utente con cui sono loggato in quel momento è necessario eseguire un logout-login per applicare le modifiche)

Tutti e tre i pulsanti presentano dei *tool-tips* che vengono mostrati al passaggio del mouse. Essi servono per permettere all’utente di capire a cosa serve ogni singolo pulsante senza dover per forza cliccarlo e quindi aprire il modale. Ecco come vengono mostrati i *tool-tip* su ogni singolo pulsante:

|  |  |  |
| --- | --- | --- |
|  |  |  |

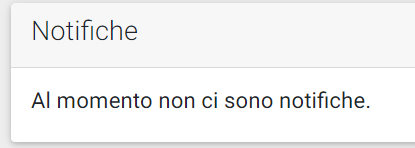
Questo è stato possibile grazie ad una libreria JavaScript chiamata *popper.js*, la quale è già presente nel framework front-end che ho utilizzato per lo sviluppo di questo progetto (MDBootstrap).

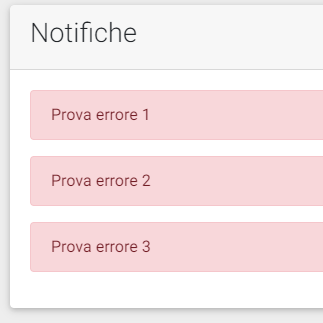
I pulsanti vengono nascosti automaticamente a seconda dei permessi dell’utente che visualizza la pagina: se per esempio l’utente *luca.dibello* non dispone dei permessi necessari per l’eliminazione degli utenti, il pulsante non viene nemmeno mostrato a schermo. Stessa cose per tutti e tre i pulsati.

Se invece l’utente non ha i permessi per la visualizzazione degli utenti presenti nel sistema l’intera tabella non verrà mostrata.

Se durante un azione di modifica, eliminazione, creazione o promozione di un utente vengono prodotti degli errori, sia di validazione che a livello di dabase o codice, viene mostrata una notifica nel blocco posto in cima alla pagina chiamato *“Notifiche”*.

Se non sono presenti notifiche esso si mostra all’utente in questa maniera:



Mentre, se sono generati degli errori, essi vengono mostrati uno sotto l’altro:

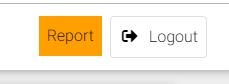
(Nota: per ulteriori informazioni sul sistema che si occupa della gestione delle notifiche vedi capitolo “*gestione notifiche*”)

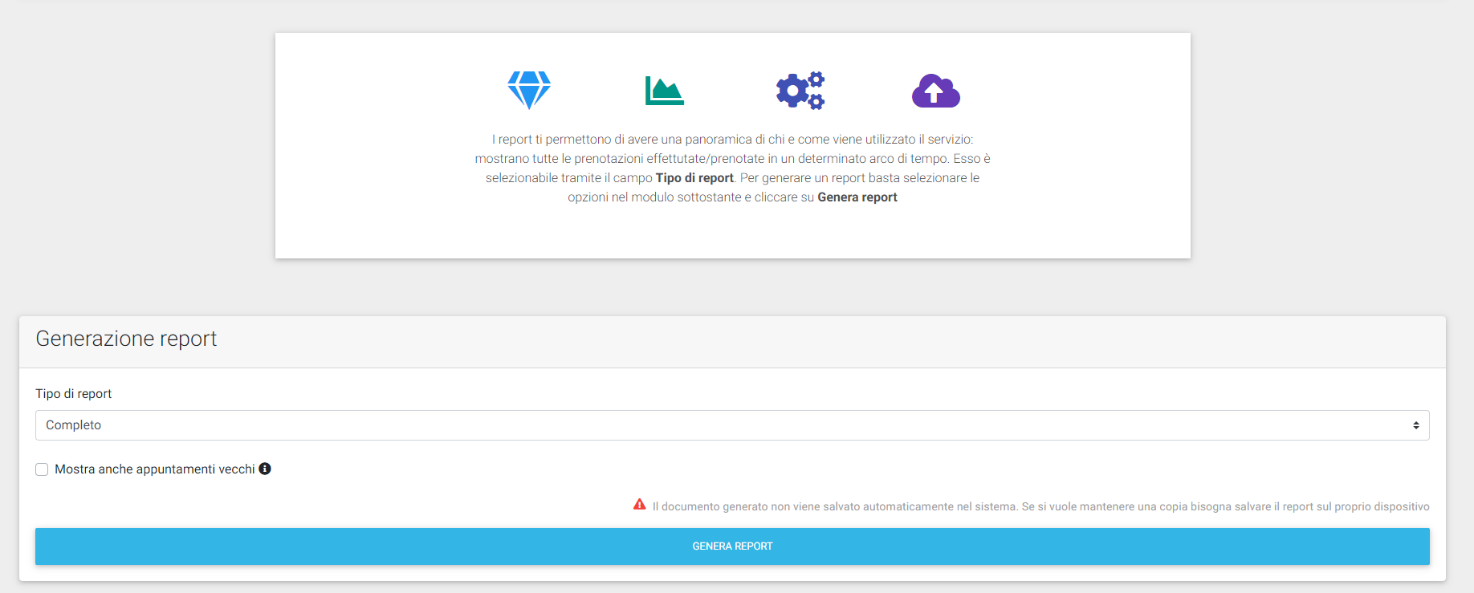
### Calendario

*CONTINUARE*

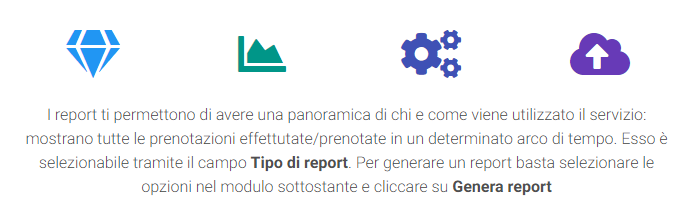
## Generazione di report

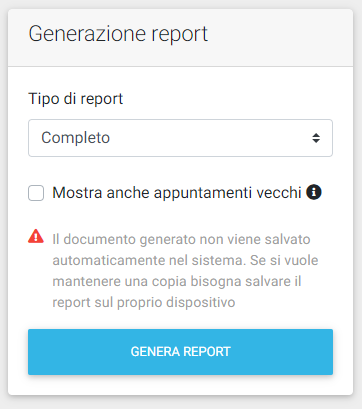
La generazione del pdf di report è resa possibile tramite la libreria FPDF, la quale è stata installata molto semplicemente tramite il package manager *Composer*.

Ho creato un controller chiamato *report*, il quale si occupa di controllare se l’utente ha i permessi necessari per la generazione di report e del caricamento delle pagine (solo si dispongono dei permessi necessari). Se l’applicazione rileva che l’utente loggato ha i permessi necessari per la generazione di questi report viene mostrato un nuovo collegamento con sfondo giallo nella barra di navigazione:

Se un utente che non dispone dei permessi necessari per accedere a questa sezione del sito, prova ad entrarci comunque il sistema mostra una pagina di errore che notifica all’utente il problema.  
Se invece l’utente ha i permessi necessari viene caricata la pagina correttamente. Questo è un screenshot della pagina:

Come si può ben vedere la pagina è strutturata da due blocchi:

Il primo blocco non è altro che una spiegazione di come generare un report. Ho deciso di aggiungere questo perchè questa funzione verrà utilizzata principalmente dalla segreteria e/o la direzione e quindi ho preferito rendere il suo funzionamento il più chiaro possibile tramite una piccola spiegazione. Ecco il messaggio mostrato

Il secondo blocco invece (intitolato *Generazione report*) non è altro che un modulo che permette di scegliere il tipo di report desiderato. Ecco come viene mostrato il menù all’utente:

Nel menù a tendina intitolato *Tipo di report* è possibile scegliere il tipo di report desiderato. Questi sono i vari tipi di report implementati nel sistema:

* Giornaliero
  + Mostra tutte le prenotazioni del giorno in cui viene generato
* Settimanale
  + Mostra tutte le prenotazioni della in cui viene generato
* Mensile
  + Mostra tutte le prenotazioni del mese in cui viene generato
* Annuale
  + Mostra tutte le prenotazioni dell’anno in cui viene generato
* Completo
  + Mostra tutte le prenotazioni presenti nel sistema

Di default i dati mostrati in questi report sono relativi alle prenotazioni future, e quindi non vengono scritte le prenotazioni che ormai sono passate.

Se invece si abilita la spunta intitolata “*Mostra anche appuntamenti vecchi”* nel report vengono mostrati anche gli appuntamenti vecchi. Per esempio se genero un report di tipo giornaliero con abilitata la spunta verranno mostrati non solo gli appuntamenti futuri ma tutti quelli dell’intera giornata.

Una volta selezionato il tipo di report desiderato e cliccato su “*Genera report”* vengono trasmesse le impostazioni all’azione *generate* del controller (quindi <http://example.com/report/generate>). Se il tipo di record scelto non viene riconosciuto dal sistema non fa altro che riportarti al controller *report,* facendo caricare nuovamente la pagina altrimenti viene creato un oggetto di tipo *ReportGenerator* al quale vengono passate le impostazioni scelte dal form:

$report = **new** ReportGenerator($type, $all\_rows\_flag);

(Nota. *$type* ha salvato al suo interno il tipo di report desiderato mentre *$all\_rows\_flag* è un *booleano*, se esso è *true* il report mostrerà tutte le prenotazioni, altrimenti solo quelle future)

E poi viene richiamato il metodo generate:

$report->generatePDF();

Questo metodo esegue diverse due principali operazioni in sequenza:

1. Generazione di una query utile per la selezione dei dati corretta, in base al tipo di report selezionato (tipo + flag dati vecchi)
2. Query sul database con tramite query generata e costruzione del PDF

La generazione della query viene fatta tramite un metodo chiamato *“\_generate\_query”*. Questo metodo non fa altro che leggere le impostazioni inviate dal form e, a seconda delle diverse combinazioni, genera una query. Per esempio questa è la query generata per selezionare i dati necessari per la costruzione di un report settimanale normale (che mostra solo i dati vecchi):

**SELECT** utente,

DATE\_FORMAT(**DATA**, '%d/%m/%Y') **AS** 'Data' ,

TIME\_FORMAT(ora\_inizio, '%H:%i') **AS** 'Ora inizio',

TIME\_FORMAT(ora\_fine, '%H:%i') **AS** 'Ora fine'

**FROM** riservazione

**WHERE** yearweek(**DATA**) = yearweek(**CURRENT\_DATE**())

**AND**

**TIMESTAMP**(CONCAT(**DATA**,' ', ora\_inizio)) > NOW()

**ORDER** **BY** **DATA**, ora\_inizio, ora\_fine **ASC**

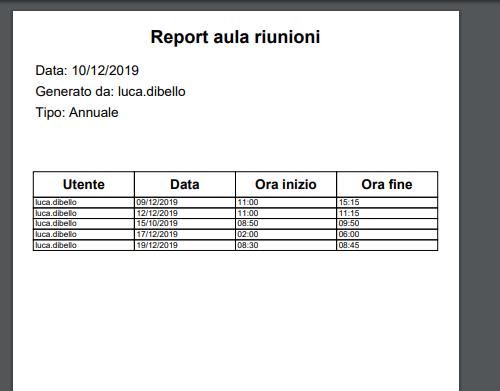
Come si può vedere in questa query vengono svolte diverse operazioni: nella prima parte di query i dati trovati vengono formattati in formato europeo, quindi data in formato “giorno/mese/anno” e gli orari in formato “ora/minuto”. Successivamente, nella seconda parte di query (dopo la *keywork* “*WHERE”*), avviene effettivamente fatto un “filtro” tra i dati presenti nella tabella. Il filtro viene fatto in due parti:

1. yearweek(**DATA**) = yearweek(**CURRENT\_DATE**())
   1. Questo primo filtro seleziona i campi con la data che appartiene alla stessa settimana della data di oggi
2. **TIMESTAMP**(CONCAT(**DATA**,' ', ora\_inizio)) > NOW()
   1. Dei dati trovati in precedenza, seleziona solamente quelli nel futuro, rispetto alla data e l’ora in cui si sta generando il report. Se si genera un report con la spunta *“Mostra anche appuntamenti vecchi”* attiva il sistema non fa altro che inviare la query senza questo filtro.

Dopo aver selezionato i dati corretti essi vengono ordinati in modo ascendente a seconda della loro data, ora di inizio ed ora di fine, quindi l’appuntamento più vicino sarà alla data ed ora di generazione, più la sua posizione all’interno della tabella sarà alta.

Dopo aver generato la query, vanno recuperati i dati e creata la tabella. Per far questo ho utilizzato un pezzo di codice trovato sulla guida ufficiale di FPDF che si occupa di far questo: esegue lui il *fetch* dei dati dal database, crea l’header della tabella, esegue i calcoli per la grandezza delle celle e scrive i dati al suo interno. Questo è il link dove ho trovato il codice <http://www.fpdf.org/en/script/script14.php>

A questo codice ho fatto qualche modifica per inserire delle informazioni aggiuntive come la data di generazione, il nome dell’utente che lo ha generato ed il tipo di report generato. Ho anche dovuto aggiungere un controllo nel codice di generazione della tabella che si occupa di controllare se effettivamente il sistema ha trovato delle prenotazioni che rispettano i canoni scelti in precendeza dal modulo (quello dove si sceglie il tipo di report da generare). Se trova i dati essi vengono scritti all’interno della tabella, se invece non viene trovato nessun dato viene mostrato un messaggio che informa l’utente che non sono stati trovati dei dati che soddisfano la richiesta.

Questo è un esempio di generazione di un report di tipo annuale:

Se invece, per esempio, genero un report giornaliero ma nella giornata non è pianificata nessuna prenotazione (e quindi la query generata non riesce a trovare i dati) la tabella viene generata con al suo interno il messaggio citato in precedenza:

Questo è il pezzo di codice che ho aggiunto per mostrare il messaggio di notifica (quello in grassetto):

*// Check if there is data or not*

if(mysqli\_num\_rows($res) > 0){

while ($row = mysqli\_fetch\_array($res))

$this->Row($row);

}

**else{**

**$this->CustomTextRow**

**(**

**"Non sono state trovate prenotazioni che soddisfano la richiesta.",**

**$prop["width"],**

**20**

**);**

**}**

Il metodo *CustomTextRow* non fa altro che creare una cella di grandezza (larghezza ed altezza) custom, con al suo interno (centrato verticalmente ed orizzontalmente) un testo custom.  
Nel mio caso genero una cella di larghezza uguale alla tabella e di altezza 20 con al suo interno il testo che notifica l’utente che non sono state trovate le prenotazioni richieste.

Questo è il codice del metodo, il quale non è altro che una piccola rivisitazione del metodo *Row* (già presente nello snippet che ho utilizzato):

**function** CustomTextRow($text,$w,$h=5){

$this->SetX($this->TableX);

$ci = $this->ColorIndex;

$fill = !empty($this->RowColors[$ci]);

if ($fill){

$this->SetFillColor(

$this->RowColors[$ci][0],

$this->RowColors[$ci][1],

$this->RowColors[$ci][2]

);

}

$this->Cell($w, $h, $text, 1, 0, 'C', $fill);

$this->Ln();

$this->ColorIndex = 1 - $ci;

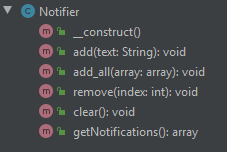
}

Se per qualche motivo lo script non riesce a collegarsi al database esso mostrerà un messaggio di errore, scrivendo delle informazioni utili per risolverlo (query che si ha provato ad eseguire ed informazioni sull’errore lette dal DBMS).

## Gestione notifiche

La gestione delle modifiche è stata fatta tramite una piccola libreria scritta ad-hoc per questo progetto chiamata *Notifier.*

Questa classe non fa altro che gestire una lista salvata all’interno della sessione per l’utente, essa infatti fornisce delle funzioni che permettono di inserire una stringa (*add*) una lista di stringhe (add\_all) e di pulire tutte le notifiche contenute nell’array di sessione (clear).

Questi sono tutte le operazioni disponibili:

Il metodo *add* ed *add\_all* (come detto in precedenza) non fanno altro che aggiungere dati alla lista. Il metodo *remove* invece permette di rimuovere una notifica specifica dall’array (in questo progetto non viene utilizzato).

Un metodo fondamentale di questa classe è *getNotifications*, il quale ritorna la lista completa delle notifiche salvate all’interno di quella di sessione. Questo metodo viene utilizzato per esempio nella schermata di login per mostrare gli errori di login oppure nel pannello admin, per mostrare gli errori di validazione dei dati.

Questa classe l’ho scritta io durante il mio stage in Irlanda. Ho deciso di riutilizzarla per questo progetto dato che, una volta implementata, è molto semplice e veloce gestire i messaggi di errore.

Questo è un esempio di come si possono leggere e scrivere all’interno di una pagina web tutte le notifiche salvate:

<!-- Check if there are notifications -->

**<?php** if(count($GLOBALS["NOTIFIER"]->getNotifications()) != 0): **?>**

<!-- Write notifications one by one -->

**<?php** foreach ($GLOBALS["NOTIFIER"]->getNotifications() as $notification): **?>**

<div class="alert alert-danger" role="alert">

**<?php** echo $notification **?>**

</div>

**<?php** endforeach; **?>**

<!-- Clear notifications -->

**<?php** $GLOBALS["NOTIFIER"]->clear(); **?>**

**<?php** endif; **?>**

In questo esempio di codice non faccio altro che, come prima cosa, controllare se ci sono delle notifiche salvate. Se ci sono le stampo una ad una a schermo e, solo dopo averle stampate a schermo, rimuovo tutte le notifiche dalla lista.

Le rimuovo per evitare di mostrarle nuovamente se l’utente ricarica la pagina.

## Validazione dati

In questo capitolo viene spiegato come ogni singolo dato viene validato, andando a spiegare le tecniche utilizzate e la natura delle mie scelte di validazione.

CONTINUARE

## Struttura API

Questo capitolo illustra tutte le API presenti nel sistema, spiegando la loro utilità all’interno del sistema ed il come richiamarle (spiegando anche i dati da inviare). Ho deciso di dedicare un intero capito a questa classe in quanto è grazie ad essa che sono possibili tutte le operazioni su utenti e prenotazioni, infatti ritengo questo controller la parte più importante del progetto: senza nessun componente del progetto andrebbe.

Questo è un piccolo schema che illustra tutte le API disponibili nel sistema:

* User
  + Delete
  + Add
  + Update
  + Promote
* Booking
  + Delete
  + Add
  + Update
* Calendar
  + Lettura di tutte le prenotazioni
  + Lettura di tutte le prenotazioni del giorno corrente

Le API *User* e *Booking* sono delle API interne, quindi delle API accessibili solamente dal sito stesso in quanto necessitano che l’utente che richiama queste funzioni dev’essere loggato e deve avere dei permessi adeguati.

Le API relative al calendario invece possono essere utilizzate anche da sistemi esterni. Per non permettere però a chiunque di leggere le prenotazioni salvate nel database ho implementato un sistema di “password” per l’utilizzo delle API del calendario. Infatti, ogni chiamata a queste API, deve essere fornita di un token inviato sotto forma di stringa. Il sistema quando rileva la richiesta controlla il codice.

Tutte le API di questo sistema (sia quelle interne che esterne) controllano anche la bontà del dato invitato (quindi esegue una validazione). Le API interne controllano anche che l’utente sia loggato e che abbia i permessi per eseguire la seguente operazione.

Ho deciso di fare questo controllo back-end dei permessi in quanto, se fossero fatti solo lato front-end, sarebbero facilmente aggirabili disattivanto JavaScript.

Le API del calendario (*Calendar*) è utilizzata dal Raspberry per leggere le prenotazioni del giorno corrente. I dati ricevuti dalle API sono sotto forma di JSON.

## Esempio di API

Questo è un piccolo pezzo di codice che mostra come ho fatto a creare un API. In questo *snippet* verrà mostrata l’API per l’eliminazione di un utente del sistema:

**public** **function** user($action = "", $username = **null**)

{

....

if (Auth::isAuthenticated()) {

$GLOBALS["NOTIFIER"]->clear();

*/\**

*\* API url: api/user/delete/<username>*

*\* Permission necessari: eliminazione\_utenti*

*\* Dati extra: nessuno*

*\*/*

if ($action == "delete" && !is\_null($username)) {

*// API creazione utenti*

*// Controllo i permessi*

if (PermissionManager::getPermissions()->canEliminareUtenti()) {$

*// L'utente ha i permessi per eseguire l'operazione*

*// Passo al model la richiesta di eliminazione dell'utente*

$result = UserModel::delete($username);

*// Controllo se l'eliminazione dell'utente ha prodotto errori*

if (is\_array($result)) {

*// Errori trovati, li aggiungo nelle notifiche*

$GLOBALS["NOTIFIER"]->add\_all($result);

}

} else {

*// Se l'utente non ha i permessi aggiungo una notifica*

$GLOBALS["NOTIFIER"]->add("Non hai i permessi necessari per eliminare gli utenti");

}

RedirectManager::redirect("admin/utenti");

}

}

....

}

Questo pattern viene ripetuto per ogni singola API del sistema.

### API User

Nei capitoli sottostanti andrò a descrivere esaustivamente come funziona ogni singola API degli utenti. Per avere delle informazioni aggiuntive sul funzionamento di esse si può andare a leggere i commenti PHPDoc nel controller *api.php* (*src/application/controller/api.php*

(Nota: Per usare queste API l’utente deve essere loggato nel sistema.)

#### Delete

Questa API viene utilizzata per eliminare un utente e tutte le sue prenotazioni dal database. Essa viene utilizzata infatti dal pannello di gestione utenti presente nel pannello di amminsitrazione del sistema.

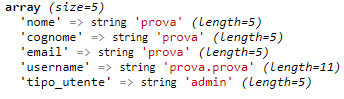
Questo è link per richiamarla: [*http://example.com/api/user/delete/****<nome\_utente>***](http://example.com/api/user/delete/%3cnome_utente%3e¨)

Una volta che il sistema rileverà la chiamata all’API, essa andrà a controllare se l’utente che ha eseguito questa richiesta ha effettivamente i permessi per farla. Per utilizzarla l’utente necessita del permesso di eliminazione utenti chiamato *eliminazione\_utenti*.

(Nota: questa API per è richiamabile tramite qualsiasi richiesta HTML (get,post,put,delete,...) in quanto non c’è nessun tipo di controllo a livello di codice.)

#### Add

Questa API viene utilizzata per creare un nuovo utente all’interno del sistema. Anch’essa viene utilizzata nel pannello di gestione utenti presente nel pannello admin.

Per richiamare questa API bisogna fornire i dati dell’utente in un array associativo inviato con *POST.* Questo è un esempio di dati inviati per la creazione di un utente:

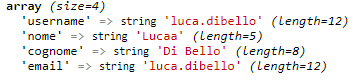
Questo è il link per richiamare l’API (ed anche dove vanno inviati i dati POST): <http://example.com/api/user/add>

Anche questa API, una volta ricevuta la richiesta, andrà a controllare se l’utente che esegue questa richiesta ha effettivamente i permessi per farlo. Per utilizzarla l’utente necessita di avere i permessi di creazione utenti chiamato *creazione\_utenti*.

Se l’utente ha i permessi per eseguire l’operazione il sistema esegue un *sanitize* dei dati POST, i quali vengono successivamente passati al metodo *add* della classe *UserModel* la quale si occuperà di validazione dei dati ed inserimento di essi nel database.

#### Update

Questa API viene utilizzata per modificare un utente già presente all’interno del sistema. Anche questa viene utilizzata dal pannello di gestione utenti presente nel pannello admin.

Per richiamare questa API bisogna fornire tutti i dati dell’utente tramite un array associativo inviato con *POST*. Questo è un esempio di dati inviati per la modifica di un utente:

Per poter richiamare correttamente l’API questi dati vanno inviati al seguente link: [http://example.com/api/update/**<nome\_utente**](http://example.com/api/update/%3cnome_utente)**>**.

Il sistema, una volta inviati i dati, controllerà se l’utente che ha eseguito la richiesta ha i permessi necessari per eseguire la seguente operazione. Per utilizzarla l’utente necessita di avere il permesso di modifica degli utenti chiamato *modifica\_utenti*.

Se l’utente ha i permessi il sistema esegue un *sanitize* dei dati inviati con POST, i quali vengono successivamente passati al metodo *update* della classe *UserModel*, la quale si occuperà della validazione dei dati e della modifica dell’utente all’interno del database.

#### Promote

Questa API viene utilizzata per modificare i permessi di un utente già presente all’interno del sistema. Anche questa viene utilizzata dal pannello di gestione utenti presente nel pannello admin.

Per richiamare questa API bisogna fornire all’api il nome del permesso da assegnare all’utente. Questa informazione viene fornita tramite l’invio di un array associativo tramite *POST*. Questo è un esempio di dati inviati per la modifica dei permessi di un utente:

Per poter richiamare correttamente l’API questi dati vanno inviati al seguente link: [http://example.com/api/promote/**<nome\_utente**](http://example.com/api/promote/%3cnome_utente)**>**.

Il sistema, una volta inviati i dati, controllerà se l’utente che ha eseguito la richiesta ha i permessi necessari per eseguire la seguente operazione. Per utilizzarla l’utente necessita di avere il permesso di promozione degli utenti chiamato *promozione\_utenti*.

Se l’utente ha i permessi il sistema esegue un *sanitize* dei dati inviati con POST, i quali vengono successivamente passati al metodo *promote* della classe *UserModel*, la quale si occuperà della validazione dei dati e della dei permessi dell’utente all’interno del database.

### API Booking

Nei capitoli sottostanti andrò a descrivere esaustivamente come funziona ogni singola API che eseguono operazioni sulle prenotazioni. Per avere delle informazioni aggiuntive sul funzionamento di esse si può andare a leggere i commenti PHPDoc nel controller *api.php* (*src/application/controller/api.php*).

Queste API vengono utilizzate dal calendario per la modifica/creazione/eliminazione degli eventi. Per ulteriori informazioni su come vengono utilizzate/richiamate queste API dal calendario leggere il capitolo chiamato *Calendario*.

Queste API, a differenza delle API degli utenti (*User*) , ritornano sempre dei dati in JSON. In caso la richiesta viene eseguita correttamente il sistema ritorna il seguente JSON:

{"success": true}

Se invece l’operazione produce degli erorri le API ritornano un JSON che contiene tutti gli errori generati. Questo è un esempio di JSON di errore:

{"success":false,"errors": ["Errore 1","Errore 2","Errore 3"]}

Ho deciso di utilizzare questo approccio in quanto queste API vengono richiamate sempre tramite JavaScript, utilizzando AJAX.

(Nota: Per usare queste API l’utente deve essere loggato nel sistema.)

#### Delete

Questa API viene utilizzata per eliminare una prenotazione presente all’interno del calendario. Questa viene utilizzata dalla pagina del calendario per elimare gli evento selezionato.

Per utilizzare l’API non è necessario inviare dati aggiuntivi, infatti basta solamente eseguire una qualsiasi richiesta HTTP (post,get,delete,put,..) a questo link: [http://example.com/api/booking/delete/**<booking\_id**](http://example.com/api/booking/delete/%3cbooking_id)**>**

Una volta che il sistema rileva la richiesta controlla se l’utente ha i permessi necessari per eseguirla. In questa API, a seconda dell’azione desiderata, si necessitano due tipi di permessi diversi:

Se l’utente vuole eliminare una propria prenotazione (creata da lui stesso), necessita del permesso per l’eliminazione delle prenotazioni personali chiamato *cancellazione\_prenotazioni*. Se invece l’utente vuole eliminare una prenotazione che non è stata creata da lui (quindi creata da un altro utente del sistema) necessita del permesso per eliminare le prenotazioni degli altri utenti chiamato: *cancellazione\_prenotazioni\_altri\_utenti*.

Se l’utente ha i permessi necessari per eseguire l’operazione il sistema richiama il metodo *delete* della classe *BookingModel* per eseguire l’operazione di eliminazione della prenotazione.

#### Add

Questa API viene utilizzata per creare una prenotazione all’interno del calendario.

Per utilizzare l’API è necessario inviare i dati che descrivono la prenotazione sotto forma di un array associativo inviato con *POST*. Questi sono i dati necessari per il funzionamento:

* data
  + Data della prenotazione (formato YYYY-MM-DD)
* ora\_inizio
  + Ora di inizio della prenotazione (formato HH:mm)
* ora\_fine
  + Ora di fine della prenotazione (formato HH:mm)
* osservazioni
  + Osservazioni aggiuntive da mostrare nella prenotazione. Se non si vuole aggiungere delle note aggiuntive basta inviare *NULL*.
* utente
  + Username dell’utente che crea la prenotazione (es: *luca.dibello*)

I dati vanno inviati al seguente URL: <http://example.com/booking/add>

Quando il sistema rileva la richiesta esso controlla anche se l’utente che richiede la creazione di una nuova prenotazione ha effettivamente i permessi per farlo. Per eseguire l’operazione l’utente deve avere i permessi per la creazione di prenotazioni chiamato *creazione\_prenotazioni*.

(Nota: È presente il permesso chiamato *creazione\_prenotazioni\_altri\_utenti* ma non è stato implementato nel sistema)

### API Calendar

Come detto in precedenza queste API sono *esterne*, quindi sono richiamabili anche da apparecchi esterni al sistema. Essendo esterne ho implementato un sistema di “accesso” tramite TOKEN. Ad ogni richiesta va inviato anche un token, se il token è corretto il sistema permette di eseguire l’operazione, altrimenti viene ritornato un errore. Il token è salvato all’interno del file di config (*config.php: src/application/config/config.php*).

Questo è il token di default utilizzato per l’autenticazione:

058c24b04169e44528ff2be1ac83f5dd787aa2109ad64fdcf142538f4d8617b5832e532a7c4a004398c3a3b4f12d1eac47423680fd71c02105d33c77cae12d5d

Queste API vengono utilizzate dal Raspberry per poter leggere le prenotazioni e quindi mostrarle sul monitor.

Questi sono i possibili errori che possono generare queste API:

* Richiesta con metodo diverso da POST
  + {“code”: 400, “message”: “Wrong request”}
* Token non inviato nella richiesta
  + {“code”: 400, “message”: “Missing API token”}
* Token errato/non riconosciuto
  + {“code”: 401, “message”: “Wrong API token”}

#### Tutti i dati

Per leggere tutte le prenotazioni presenti nel sistema basta fare una richiesta POST (la quale contiene il token) al seguente URL: [*http://example.com/api/calendar*](http://example.com/api/calendar)

Se il token viene riconosciuto, il sistema richiama il metodo *getBookingsAfterDateTime* della classe *CalendarModel* per ritornare un array di *Booking* che descrive tutte le prenotazioni future. Esse vengono successivamente convertite in dati json e ritornati dalle API.

Questo è un esempio di risposta dell’API:

[{"id":18,"title":"luca.dibello","start":"2019-12-17T12:45:00+01:00","end":"2019-12-17T13:00:00+01:00","note":null,"professor":"luca.dibello"}]

#### Prenotazioni giornaliere

Per leggere tutte le prenotazioni giornaliere basta fare una richiesta POST (inviando anche il token di autenticazione) all’URL: <http://example.com/api/calendar/day>

Se il token viene riconosciuto, il sistema richiama il metodo *getBookingsAfterDateTimeDay* della classe *CalendarModel* per ritornare un array di *Booking* che descrive tutte le prenotazioni future della giornata. Esse vengono successivamente convertite in dati json e ritornati dalle API.

# Test

## Protocollo di test

Definire in modo accurato tutti i test che devono essere realizzati per garantire l’adempimento delle richieste formulate nei requisiti. I test fungono da garanzia di qualità del prodotto. Ogni test deve essere ripetibile alle stesse condizioni.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case:**  **Riferimento**: | TC-001  REQ-012 | **Nome:** | Import a card with KIC, KID and KIK keys, but not shown with the GUI |
| **Descrizione:** | Import a card with KIC, KID and KIK keys with no obfuscation, but not shown with the GUI | | |
| **Prerequisiti:** | Store on local PC: Profile\_1.2.001.xml (appendix n\_n) and Cards\_1.2.001.txt (appendix n\_n).  PIN (OTA\_VIEW\_PIN\_PUK\_KEY) and ADM (OTA\_VIEW\_ADM\_KEY) user right not set. | | |
| **Procedura:** | 1. Go to “Cards manager” menu,  in main page click “Import Profiles” link, Select the “1.2.001.xml” file, Import the Profile 2. Go to “Cards manager” menu,  in main page click “Import Cards” link, Select the “1.2.001.txt” file, Delete the cards,  Select the “1.2.001.txt” file, Import the cards 3. Research the “41795924770” Card, Click the imsi card link Check the card details 4. Execute the SQL: SELECT imsi, dir, keyset, cntr, rawtohex(kickey), rawtohex(kidkey), rawtohex(kikkey), rawtohex(chv), rawtohex(dap)FROM otacardkey a where imsi='340041795924770' ORDER BY keyset; | | |
| **Risultati attesi:** | Keys visible in the DB (OtaCardKey) but not visible in the GUI (Card details) | | |

## Risultati test

Tabella riassuntiva in cui si inseriscono i test riusciti e non del prodotto finale. Se un test non riesce e viene corretto l’errore, questo dovrà risultare nel documento finale come riuscito (la procedura della correzione apparirà nel diario), altrimenti dovrà essere descritto l’errore con eventuali ipotesi di correzione.

## Mancanze/limitazioni conosciute

Descrizione con motivazione di eventuali elementi mancanti o non completamente implementati, al di fuori dei test case. Non devono essere riportati gli errori e i problemi riscontrati e poi risolti durante il progetto.

# Consuntivo

Consuntivo del tempo di lavoro effettivo e considerazioni riguardo le differenze rispetto alla pianificazione (cap 1.7) (ad esempio Gannt consuntivo).

# Conclusioni

Quali sono le implicazioni della mia soluzione? Che impatto avrà? Cambierà il mondo? È un successo importante? È solo un’aggiunta marginale o è semplicemente servita per scoprire che questo percorso è stato una perdita di tempo? I risultati ottenuti sono generali, facilmente generalizzabili o sono specifici di un caso particolare? ecc

## Sviluppi futuri

Migliorie o estensioni che possono essere sviluppate sul prodotto.

* Aggiunta/modifica di permessi tramite pannello admin
* Migliore utillizzo del calendario da dispositivi mobili
* Gestione prenotazioni dal pannello admin (sotto forma di tabella)
* Generazione di report più di bell’aspetto

## Considerazioni personali

Con questo progetto ho imparato come creare e gestire un calendario completamente utilizzanto funzioni front-end, quindi tramite JavaScript ed AJAX. Ho imparato come gestire i permessi degli utenti sia a livello di banca dati sia a livello di applicativo (controlli, assegnazione permessi, lettura permessi).

# Bibliografia

## Sitografia

1. URL del sito (se troppo lungo solo dominio, evt completo nel diario),
2. Eventuale titolo della pagina (in italico),
3. Data di consultazione (GG-MM-AAAA).

**Esempio:**

* http://standards.ieee.org/guides/style/section7.html, *IEEE Standards Style Manual*, 07-06-2008.

# Allegati

Elenco degli allegati, esempio:

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