LE VISTE (VIEW)

Una vista è una ***tabella virtuale***, cioè che **non** viene memorizzata nel database. La **vista** è utilizzabile a tutti gli effetti come una tabella reale, QUINDI E' POSSIBILE ESEGUIRE QUERY SU DI ESSA. La view è una tabella che viene costruita nel momento in cui viene eseguita la query che la genera. Serve per *mostrare* ad un utente un particolare insieme di dati evitando di mostrare i dati che non sono di suo interesse (poi vedremo meglio tutti i vantaggi).

Quando si crea una vista, cosa viene memorizzato sulla memoria di massa? Viene memorizzata la query che genera la vista con il relativo nome, ma non la tabella ottenuta eseguendo la vista, quindi una vista non occupa ulteriore memoria nel database.

Sintassi per creare una view:

CREATE VIEW nome_vista AS

SELECT (elenco_campi)

FROM (elenco_tabelle)

[WHERE condizione]

In pratica, la vista è il risultato della query alla quale viene dato un nome e che è utilizzabile per ulteriori query

Esempio, creiamo una view che mostra solo il personale del dipartimento2:

CREATE VIEW personale_d2 AS

SELECT matricola,nominativo, data_nascita,stipendio, qualifica

FROM personale

WHERE id_dip='d2'

Una volta eseguita questa query, nel database è presente una vista personale_d2 utilizzabile in altre query come se fosse una tabella vera e propria

Operazione sulla vista

SELECT *

FROM personale_d2

WHERE stipendio>2500

Una vista può essere anche molto complessa, contenere subquery, funzioni di aggregazione, calcoli, ecc. e può o meno essere **aggiornabile** (**ossia consentire modifiche ai dati mostrati dalla vista, che si riflettono nelle tabelle sottostanti).** Una **vista è aggiornabile** solo se esiste una corrispondenza UNO a UNO fra le sue righe e le righe delle tabelle sottostanti. Una vista quindi non è aggiornabile se, ad esempio, contiene funzioni di aggregazione.

Esempio1: view non aggiornabile perché contiene funzioni di aggregazione

CREATE VIEW Stipendi_totali_dipartimenti AS

SELECT id_dip, SUM(stipendio) as totale_stipendi

FROM personale

GROUP BY id_dip;

Se si prova a modificare il valore dell' id_dip da D3 a D2 con la seguente query si ottiene un errore:

UPDATE Stipendi_totali_dipartimenti SET id_dip="D3" WHERE id_dip="D2"

Un aggiornamento sulla view precedente (personale_d2) andrebbe a buon fine poiché non sono presenti in tale query funzioni di aggregazione:

UPDATE personale_d2 SET id_dip="D3" WHERE id_dip="D2"

Inoltre si possono apportare modifiche (update o delete) alla view solo se tali modifiche interessano i campi di **UNA SOLA TABELLA** sottostante. Quindi se una vista è ottenuta dal JOIN fra due tabelle T1 e T2 e si tenta di eseguire una query per modificare i dati della vista, accade questo:

- Se **la modifica** riguarda dati che si trovano in **UNA SOLA** tabella sottostante, la modifica viene eseguita.
- Se **la modifica** riguarda **contemporaneamente dati che si trovano nella tabella T1 e dati che si trovano nella tabella T2** la modifica non avviene

Esempio 2: view non aggiornabile perché coinvolgente tabelle distinte

Creo la query che seleziona il personale con qualifica q5 e i dati dei relativi dipartimenti

CREATE VIEW personale_q5_dipartimenti AS

SELECT (*)

FROM personale INNER JOIN dipartimenti USING (id_dip)

WHERE qualifica=q5

Da questa view è possibile aggiornare i dati delle singole persone, **che si riflettono nella tabella sottostante**, oppure i dati dei singoli dipartimenti, che si riflettono nella tabella sottostante

Non è possibile però modificare **CONTEMPORANEAMENTE** i dati di un dipendente e i dati di un dipartimento, in quanto tali dati appartengono a due tabelle diverse.

La seguente query infatti genera un errore:

UPDATE `personale_q5_dipartimenti` SET `nominativo` = 'ROSSI MAURO', localita="via Marconi" WHERE `personale_q5_dipartimenti`.`matricola` = '00013';

I vantaggi dell'utilizzo delle viste sono:

- possono essere utilizzate nelle query come se fossero vere e proprie tabelle
- favoriscono il riutilizzo del codice: ad esempio il risultato di una query complessa può essere salvato in una VIEW e messo a disposizione di tutti coloro che stanno lavorando sullo stesso database.

Esempio: costruire una view che mostra, per ogni prodotto, l'id_prod e il suo costo (ottenuto a partire dal costo unitario dei suoi componenti e dalla quantità di tali componenti utilizzata nel prodotto):

```
create view costo_prodotti AS

SELECT composizione.id_prod, sum(componenti.costo_unitario*composizione.unita_comp) as costo_prodotto

FROM composizione, componenti

WHERE composizione.id_comp=componenti.id_comp

group by composizione.id_prod
```

Questa view ha il vantaggio, rispetto ad una tabella, che se cambia il costo unitario di un componente, il costo del prodotto rimane sempre aggiornato poiché è calcolato ogni volta che la view viene eseguita.

- consente agli utenti del database di vedere solo la parte "di loro interesse" di un DB molto complesso (ad esempio con centinaia di tabelle) e quindi consente loro di lavorare pur senza dover conoscere tutta la struttura complessa del DB. Ad esempio Il DB Administrator potrebbe predisporre delle viste per gli sviluppatori che si occupano di costruire in PHP parti di applicazione che devono accedere solo ad alcuni dati del database. Grazie a questo aspetto le view consentono di realizzare il livello utente indicato nel modello ANSI/SPARC delle applicazioni con DBMS visto all'inizio dell'anno.
- Consentire accessi selettivi: ad esempio, considerando alcune tabelle con dati sensibili, le view possono essere utilizzate per consentire, a determinati utenti, l'accesso solo a determinati campi. Questo aspetto garantisce la riservatezza dei dati.
- **Indipendenza della vista dalla struttura logica delle tabelle**. Questo è il vantaggio maggiore! Consente di realizzare l' **indipendenza logica dei dati** che avevamo visto all'inizio dell'anno!

  Esempio: se ad un certo punto si decidesse di apportare modifiche alla struttura dei dati, ad esempio di modificarei campi di una tabella, alcune delle query scritte (utilizzate nel codice PHP di una web application) andrebbero aggiornate per poter funzionare.
  Se invece le query della web application venissero svolte su una View, basterebbe modificare la View per rendere ancora valide le query.
  Grazie alle viste dunque il DB administrator potrebbe apportare modifiche alla struttura dati senza la necessità, da parte dello sviluppatore web, di modificare il codice dell'applicazione!

  ESEMPIO
  Nell'applicazione web che gestisce l'azienda è presente una pagina che mostra il totale degli stipendi ciascun dipartimento.

Supponiamo che nella tabella “personale”, per ragioni fiscali, debba essere modificato l’attributo stipendio. Anziché un attributo “stipendio” è necessario avere due attributi: “stipendio_netto” e “contributi”.

A. **Cosa succede senza le view.**

I dati da mostrare sono ottenuti dal database eseguendo la seguente query:

SELECT id_dip, sum(stipendio) as totale_stipendio
FORM personale
GROUP BY id_dip

```
<head>
    <title>Stipendi senza viste</title>
</head>
<body>
    <?php
        $server="localhost";
        $username="root";
        $password="";
        $dbname="azienda2";

        $connection=new mysqli($server,$username,$password,$dbname);
        if($connection->connect_errno)
            die("errore di connessione ". $connection->connect_errno);
        else
            echo "Connessione ok<br>";

        echo"<h1>TOTALE STIPENDI PER DIPARTIMENTO</h1><br><br>";

        $query="select id_dip, sum(stipendio) as totale_stipendio from personale
group by id_dip";
        $result=$connection->query($query);
        if($result->num_rows==0)
            echo"Nessun risultato";
        else
        {
            echo"DIP   &nbspStipendio<br><br>";
            while($row=$result->fetch_assoc())
            {
                echo"$row[id_dip]    $row[totale_stipendio]<br><br>
";
            }
            $result->free();
        }
        $connection->close();
    ?>
```

```
</body>
</html>
```

Risultato:

Se il DB Administrator modifica la tabella Personale nel seguente modo:

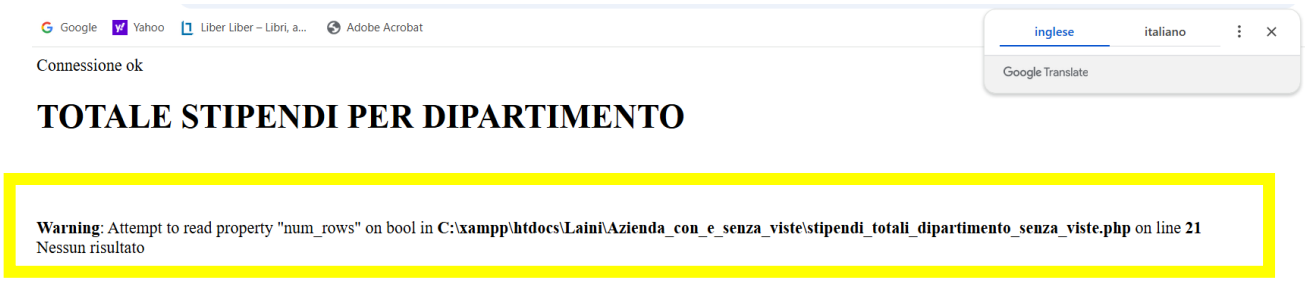
(copia e incolla)

```
ALTER TABLE personale ADD contributi float;        /*aggiunge colonna "contributi"*/

UPDATE personale                                   /*modifica dati nella colonna stipendio*/
SET contributi = stipendio * 0.3,                  /*e nella colonna contributi*/
   stipendio = stipendio - (stipendio * 0.3);

ALTER TABLE personale                              /*modifica il nome di "stipendio"*/
CHANGE COLUMN stipendio stipendio_netto float;     /*in "stipendio_netto"*/
```

Dopo aver opportunamente modificato la base di dati, la query in PHP non funziona più, quindi la pagina web restituisce un errore:




```
SELECT id_dip, sum(stipendio_netto+contributi) as totale_stipendio
FROM personale
GROUP BY id_dip
```

B. **Cosa succede con le view.**

Se, per creare le query iniziali da usare in PHP, si fosse utilizzata una View “vista_personale”, che, prima della modifica del database, coincide con la tabella personale:

CREATE VIEW vista_personale AS
SELECT *
FROM personale

La query sarebbe stata, inizialmente, le seguente:

SELECT id_dip, sum(stipendio) as totale_stipendio
FORM personale
GROUP BY id_dip

La pagina php sarebbe stata la seguente:

```
<body>
    <?php
        $server="localhost";
        $username="root";
        $password="";
        $dbname="azienda_con_viste";

        $connection=new mysqli($server,$username,$password,$dbname);
        if($connection->connect_errno)
            die("errore di connessione ". $connection->connect_errno);
        else
            echo "Connessione ok<br>";

        echo"<h1>TOTALE STIPENDI PER DIPARTIMENTO</h1><br><br>";

        $query="select id_dip, sum(stipendio) as totale_stipendio from
vista_personale group by id_dip";
        $result=$connection->query($query);
        if($result->num_rows==0)
            echo"Nessun risultato";
        else
        {
            echo"DIP   &nbspStipendio<br><br>";
            while($row=$result->fetch_assoc())
            {
                echo"$row[id_dip]    $row[totale_stipendio]<br><br>
";
            }
            $result->free();
        }
        $connection->close();
```

```
    ?>
</body>
</html>
```

In seguito alla stessa modifica del database del caso precedente, non sarebbe stato necessario modificare le due query “embeddate” nel PHP ma sarebbe stato sufficiente cancellare la view “vista_personale” e ricrearla nel seguente modo:

CREATE VIEW vista_personale AS

SELECT matricola, nominativo, data_nascita, qualifica, stipendio_netto+contributi as stipendio, id_dip

FROM personale

Modificando la vista, le pagine PHP continuano a funzionare senza la necessità di alcuna modifica nel codice. Tutte le pagine web dell’applicazione che utilizzavano la vista “vista_personale” non richiedono alcuna modifica. Il vantaggio è duplice:

1. La modifica riguarda solamente UNA vista del Databzse anziché le (magari molte) pagine php
2. La modifica viene eseguita direttamente dal DB Administrator senza che gli addetti alla gestione delle pagine web debbano intervenire.

I TRIGGER

Un trigger è un meccanismo che consente di intercettare un evento di aggiornamento dei dati (INSERT, UPDATE, DELETE) ed indicare delle azioni SQL da eseguire **prima o dopo tale aggiornamento**.

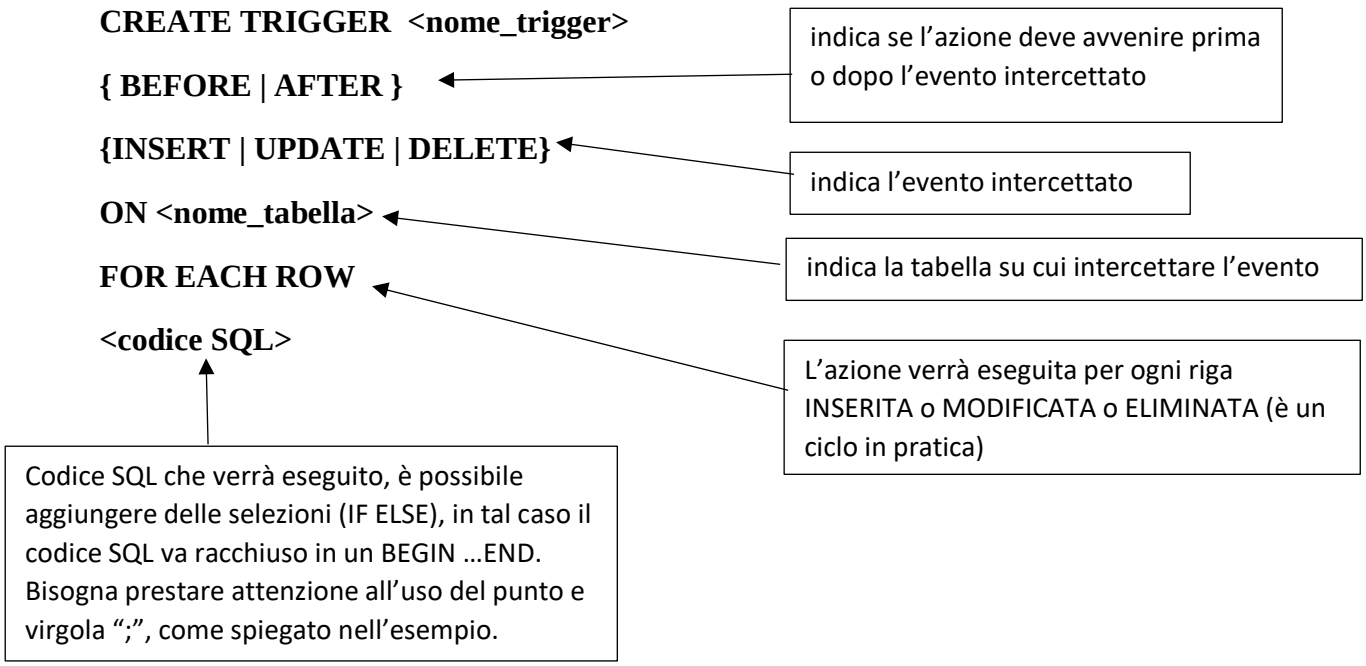
Quando si usano?

- Per aggiungere vincoli e quindi mantenere l'integrità dei dati
  Per esempio si può controllare con un trigger che un dato, prima di essere inserito, rispetti il dominio (come il vincolo CHECK)

- Per aggiornare dei dati "a cascata" in seguito all'aggiornamento di altri dati, come vedremo con un esempio.

Un Trigger è un **oggetto** legato alla tabella sulla quale si vuole intercettare l'evento.

La sintassi per un nuovo trigger è la seguente:



Vediamo alcuni esempi di Trigger e spieghiamo il codice.

ESEMPIO 1: trigger che impedisce l'inserimento, nella tabella prodotti, di un prodotto con prezzo negativo:

```
DELIMITER $$          // Questa istruzione cambia il delimitatore (delimiter). Il delimitatore è il simbolo utilizzato da sql per indicare la fine di un'istruzione. Di default il delimitatore in SQL è ";". Poiché nei trigger spesso è necessario utilizzare più istruzioni che terminano con ";" è necessario modificare il delimitatore per evitare che l'interprete SQL interpreti come fine dell'istruzione il primo ";" rilevato, ignorando il codice successivo. Con questa istruzione il delimiter viene posto temporaneamente a "$$"

CREATE TRIGGER controllo_prezzo

BEFORE INSERT ON prodotti

FOR EACH ROW

BEGIN

IF NEW.prezzo < 0 THEN        //Dopo FOR EACH ROW si può far riferimento al valore assunto da ciascuna riga interessata precedentemente alla modifica e successivamente alla modifica, rispettivamente con OLD e NEW. Quindi NEW indica la nuova tupla che si sta tentando di aggiungere alla tabella

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Il prezzo del prodotto non può essere negativo.';
                          // Questa istruzione genera un errore con un codice di errore generico 45000. Questo è un valore convenzionale per indicare un errore appositamente generato dall'utente (non predefinito da SQL). SET MESSAGE_TEXT genera un messaggio di errore.

  END IF;

END$$                 // fine del blocco di istruzioni utilizzando il delimitatore $$

DELIMITER ;           // ripristino del delimitatore di default ";"
```

Il trigger intercetta l'inserimento di un prodotto con prezzo negativo e solleva l'errore evitandone l'inserimento.

La seguente query:

INSERT INTO prodotti(id_prod, nome_prodotto, id_dip, prezzo) values ('P123','prodotto x', 'D1', -12)

genera l'errore.

ESEMPIO 2: supponiamo di aggiungere, per comodità, un campo "costo" alla tabella prodotti, il valore assunto da tale campo deve essere dato dalla somma, per ogni componente utilizzato, del costo del componente per la quantità utilizzata. (anche se nel capitolo dedicato alle Views abbiamo già creato una view che fa questo che è una soluzione migliore)

Aggiungiamo ora un trigger che consente l'inserimento del costo di un prodotto solamente se il valore inserito è corretto, ossia se coincide con la somma dei prodotti di ogni componente per la quantità utilizzata.

Per aggiungere il campo costo:

```
ALTER TABLE prodotti ADD costo float
```

Il trigger da aggiungere è il seguente. **Attenzione**: è stato necessario utilizzare la funzione ROUND per evitare che il trigger non aggiornasse correttamente per problemi di arrotondamento nei calcoli.

```
DELIMITER $$

CREATE TRIGGER controllo_costo

BEFORE UPDATE ON prodotti

FOR EACH ROW

BEGIN

IF ROUND(NEW.costo,2) != ROUND( (SELECT sum(unita_comp*costo_unitario) from
composizione,componenti WHERE composizione.id_comp=componenti.id_comp

        AND composizione.id_prod=NEW.id_prod) ,2)THEN

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Il costo del prodotto non è corretto in base
al costo dei componenti.';

  END IF;

END$$

DELIMITER ;
```

IL VALORE DI COSTO INSERITO

IL COSTO CALCOLATO CON UNA QUERY

Poiché il costo del prodotto P001 con i dati dell'esempio è di 1.6 €, questo trigger consente l'inserimento del seguente costo:

```
UPDATE prodotti SET costo=1.6 WHERE id_prod='p001'
```

Ma non consente l'inserimento del seguente costo:

```
UPDATE prodotti SET costo=1.5 WHERE id_prod='p001'
```

ESEMPIO 3: Aggiungiamo un trigger che, al cambiare del costo unitario di un componente, aggiorna il costo dei prodotti che contengono tale componente

```
DELIMITER $$

CREATE TRIGGER aggiornamento_prezzo_prodotto
AFTER UPDATE ON Componenti
FOR EACH ROW
BEGIN
   -- Dichiarazione di una variabile per il costo totale
   DECLARE costo_totale DECIMAL(10,2);

   -- Aggiorna il prezzo solo per i prodotti che usano il componente modificato
   UPDATE Prodotti p
   SET p.costo = (
      SELECT SUM(c.unita_comp * cmp.costo_unitario)
      FROM Composizione c
      JOIN Componenti cmp ON c.id_comp = cmp.id_comp
      WHERE c.id_prod = p.id_prod
   )
   WHERE EXISTS (
      SELECT 1
      FROM Composizione c
      WHERE c.id_prod = p.id_prod
      AND c.id_comp = NEW.id_comp
   );
END$$

DELIMITER ;
```

Grazie a questo trigger, ogni volta che viene modificato il costo unitario di un componente, automaticamente viene aggiornato il costo di ogni prodotto che utilizza quel componente.

GESTIONE DEGLI UTENTI

Per poter visualizzare la pagina che chiede la password e quindi accedere con login:

Lo username di default è : root

La password di default è la stringa vuota: ''

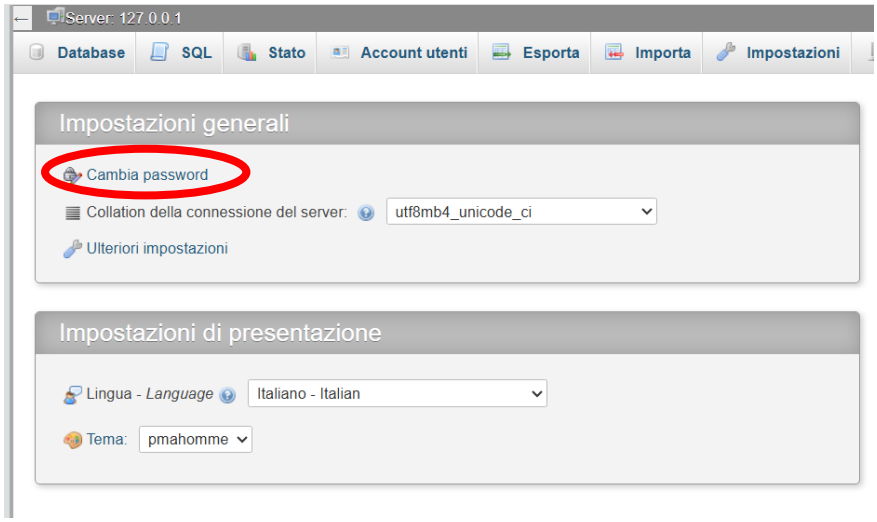
Per modificare tali credenziali di accesso:

1. Apri il file C:\xampp\phpMyAdmin\ config.inc.php
2. apporta le seguenti modifiche:

```
$cfg['Servers'][$i]['auth_type'] = 'cookie';                    //era 'config' //per mostrare la
                                                                //maschera di inserimento pw
```

3. Salva il file.

Se vuoi cambiare la password, accedi al phpMyAdmin/server e clicca su cambia password:



Modifica la pasword di root

Da ora quando si accede a localhost\phpMyAdmin appare una schermata di login che consente l'accesso con username e password.

Quando installiamo MySQL o MariaDB viene creato di default un *"super utente"* amministratore di tutti i database. Tale super utente ha come username ***root*** e ***non ha alcuna password***. Naturalmente questa soluzione è comoda ma è da utilizzare solo in fase di test. Nella realizzazione di un server per DB reali, ai fini della sicurezza, è sempre opportuno che l'utente principale, abbia un propria password.

L'utente *root* può svolgere qualunque operazione su qualunque DB creato, inoltre può creare altri utenti per ogni DB, assegnare a ciascun nuovo utente una password e assegnare o revocare dei **privilegi** a ciascun nuovo utente.

Quando si crea un nuovo utente è possibile specificare lo username, la password e l'host dal quale questo utente può accedere. Per indicare l'host Si possono utilizzare i caratteri "jolly" quindi con "%" si indica qualsiasi host. Con "localhost" si intende lo stesso computer su cui è in esecuzione il DBMS.

Cosa sono questi **privilegi o permessi**?

Sono le possibili operazioni che un utente può svolgere sul Database. Tali privilegi possono essere assegnati a livello GLOBALE, quindi su tutti i database del serverr (si indica con *.*), a livello di singolo DATABASE (si indica con nome_database.*) e anche a livello di singola tabella (si indica con nome_database.nome_tabelle)

I privilegi vengono assegnati con le istruzioni GRANT (per assegnare un privilegio) e REVOKE (per revocare un permesso).

A seconda dei privilegi assegnati ad un utente, esso potrà o meno accedere al DB:

- solo in lettura (operazioni consentite solo SELECT),
- creare/modificare/eliminare i dati (INSERT, ecc..),
- creare/modificare/eliminare tabelle (CREATE TABLE,ALTER TABLE, DROP TABLE)
- creare/eliminare/modificare i permessi di altri utenti.

La tabella dei privilegi viene di seguito riportata.

Permesso	Istruzioni
ALL	tutte esclusa GRANT
ALTER	ALTER TABLE
CREATE	CREATE TABLE
CREATE TEMPORARY TABLES	CREATE TEMPORARY TABLE
CREATE VIEW	CREATE VIEW
DELETE	DELETE
DROP	DROP TABLE
INDEX	CREATE INDEX, DROP INDEX
INSERT	INSERT
LOCK TABLES	LOCK TABLES
SELECT	SELECT
SHOW VIEW	SHOW CREATE VIEW
UPDATE	UPDATE
USAGE	nessuna
GRANT OPTION	GRANT, REVOKE

Tabelle che esistono soltanto in una sessione del database. Alla chiusura della connessione verrà automaticamente eliminata. Serve per query complesse ecc.

E' il privilegio che ha un utente senza alcun privilegio. A volte è utile creare un utente senza privilegi (non può fare nulla) e poi assegnare i privilegi

Quando si crea un nuovo utente per il quale viene definita una password, la password di accesso al database viene cifrata con una funzione di Hash (che dipende dalla versione di MariaDB) e memorizzata. I dati degli utenti con i rispettivi privilegi di accesso al database sono memorizzate in una apposita vista "*users*" (oppure, nella tabella mysql.global_priv_table dipende dalle versioni) del DB "*mysql*" presente nel server al momento dell'installazione del DMBS.

Vediamo un esempio in cui creiamo un utente assegnando ad esso solo i privilegi di select sul DB Azienda.

CREATE USER "pierone"@"localhost" IDENTIFIED BY "password_pierone"

Viene creato l'utente, la password viene cifrata e memorizzata nella tabelle mysql.user

L’utente creato non ha alcun privilegio (ha il privilegio ‘usage’)

DROP USER "pierone"@"localhost"

L’utente viene eliminato.

Poiché gli algoritmi di Hash di MariaDB potrebbero essere non particolarmente sicuri e aggiornati nel tempo, può essere utile creare utenti fornendo la password già “Hashata”, utilizzando il seguente comando:

CREATE USER "pierone"@"localhost" IDENTIFIED BY PASSWORD "hash_string"

Viene creato l’utente la cui password viene fornita già cifrata, tale hash string verrà memorizzato come hash_string della password (può essere che il DBMS richieda un Hash di lunghezza specifica)

Dopo aver creato nuovamente l’utente pierone, assegno all’utente pierone i privilegi INSERT e SELECT sul database Azienda3

```
GRANT SELECT, INSERT
ON Azienda3.*
TO "pierone"@"localhost"
```

l’utente pierone, solo se accede al DB dal localhost, potrà inserire nuove righe nella tabella ed effettuare query.
Provare ad accedere al database Azienda3 da phpMyadmin con le credenziali di Pierone e verificare:

- si potrà svolgere query
- si potrà aggiungere un dipendente
- non si potrà eliminare il dipendente aggiunto.

Dopo aver fatto il login come utente root, andiamo a rimuovere il privilegio INSERT per l’utente pierone@localhost

```
REVOKE INSERT
ON Azienda3.*
FROM "pierone"@"localhost";
```

(funziona anche se sottolinea in rosso REVOKE)

Provare ad accedere al database Azienda3 da phpMyadmin con le credenziali di Pierone e verificare che ora non è più consentito a tale utente di aggiungere dati ma solamente svolgere query

Chiarimento necessario sulla differenza fra utenti di un sito/applicazione ed utenti di un database memorizzati nel DBMS

Quando si realizza un'applicazione web in cui è richiesto agli utenti di effettuare un login, gli userId e le password dell'applicazione (o sito) vengono memorizzati in un'apposita tabella "utenti" del database.

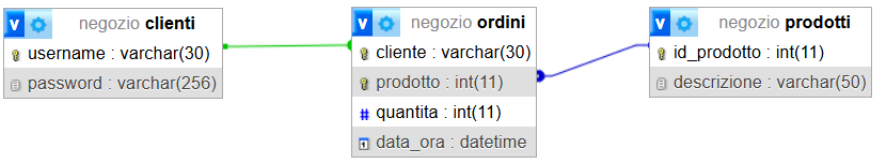
Nel momento in cui un utente dell'applicazione tenta di accedere all'applicazione web attraverso un form di login, l'applicazione **è già connessa al database con delle credenziali** di un "utente del database" memorizzato nella vista "users" di MySQL. Quando l'utente del sito invierà il proprio username e pw, verrà eseguita una query sulla tabella "utenti" del database dell'applicazione, e ciò determinerà la possibilità o meno di accedere al sito da parte dell'utente in base alla presenza o meno delle credenziali nella tabella "utenti". La query generata dal form verrà eseguita da un "utente del database" (i cui username e pw del database sono quelli con il quale è stata effettuata la connessione) e questa è una figura totalmente diversa da quello dell'utente dell'applicazione, e che gode di determinati privilegi. E' opportuno che i privilegi dell'utente del database con cui si effettua la connessione siano opportunamente limitati in modo da garantire la sicurezza del database.

Per garantire la sicurezza dei dati **(CIA: confidentiality, integrity, aviability,** che significano riservatezza, integrità, disponibilità dei dati**)** è utile creare diverse tipologie di utenti nell'accesso ai dati di un database, ciascuno con privilegi diversi.

Ciascuna tipologia di utente si connette al database con l'username e password assegnata dall'amministratore del database. E' possibile dunque creare diversi file php di connessione al database in base alla tipologia di utente che si deve connettere, che saranno utilizzati per la connessione al database nelle diverse pagine php.

ESEMPIO:

Ipotizziamo il seguente semplice database di un sito di un negozio per la vendita di prodotti



Vi saranno le seguenti tipologie di utenti del database:

- **DB administrator**: può svolgere qualsiasi operazione su qualsiasi tabella e creare nuovi utenti, può connettersi solo dall'host locale in cui è installato il DBMS
- **titolare**: può svolgere operazioni CRUD (in SQL sono: insert, select, update, delete) in tutte le tre tabelle, può connettersi solo dall'host locale in cui è installato il DBMS
- **cliente**:
    - nella tabella "prodotti"
        - può leggere (per visualizzare i prodotti)
        - non può creare, modificare, eliminare prodotti

- nella tabella “clienti”
  - può creare nuovi dati (quando si registra)
  - può leggere (quando effettua il login)
  - non può eliminare e modificare i dati (per cancellare /modificare i propri dati deve contattare il titolare)
- nella tabella “ordini”:
  - può scrivere nuovi dati (quando effettua un ordine)
  - può leggere (quando vuole visualizzare i propri ordini)
  - non può eliminare e modificare i dati (per cancellare /modificare un ordine deve contattare il titolare)

Il super utente del server (ROOT) crea il DB Administrator del db negozio assegnandogli tutti i privilegi COMPRESI I PRIVILEGI GRANT

```
-- Crea l'utente
CREATE USER 'admin_negozio'@'localhost' IDENTIFIED BY 'admin';

-- Concede tutti i privilegi sul database "negozio"
GRANT ALL PRIVILEGES ON negozio.* TO 'admin_negozio'@'localhost'
WITH GRANT OPTION;

-- Applica i cambiamenti
FLUSH PRIVILEGES;
```

Il super utente (root) e crea anche i due profili “titolare” e “cliente” (in MariaDB non si può assegnare il privilegio CREATE USER ad utente per uno specifico database, il superutente può assegnarlo ad un altro utente ma solo a livello di server, quindi per tutti i database presenti):

```
-- può accedere solo da localhost --
create user "titolare"@"localhost" identified by "titol";

-- può accedere da qualsiasi host --
create user "cliente"@"%" identified by "clie";

-- Applica i cambiamenti
FLUSH PRIVILEGES;
```

Il DB Administrator (admin_negozio) accede al database (ad esempio con phpMyAdmin) e crea le tabelle del database.

```
create table clienti
(username varchar(30) primary key not null,
 password varchar(256) not null);

create table prodotti
(id_prodotto integer auto_increment primary key not null,
 descrizione varchar(50) not null);

create table ordini
```

```
(id_prodotto integer not null,
 cliente varchar(30) not null,
 quantita integer not null,
data_ora datetime not null,
constraint pk PRIMARY key (id_prodotto,cliente),
constraint fk_ordini_prodotti FOREIGN KEY (id_prodotto)
REFERENCES prodotti(id_prodotto),
constraint fk_ordini_clienti FOREIGN KEY (cliente) REFERENCES
clienti(username)
 );
```

L'utente "admin_negozio" assegna i privilegi stabiliti all'utente di tipo "titolare"

```
grant insert, update, delete, select
on negozio.*
to titolare@'localhost';
```

L'utente "admin_negozio" assegna i privilegi stabiliti all'utente di tipo "cliente"

```
grant select
on negozio.prodotti
to 'cliente'@'%';

grant insert,select
on negozio.clienti
to 'cliente'@'%';

grant insert,select
on negozio.ordini
to 'cliente'@'%';
```

Per testare i profili 'titolare' e 'cliente' connettiamoci al database come titolare e vediamo che è possibile aggiungere prodotti alla tabella 'prodotti'

Titolare:

```
insert into prodotti(descrizione)
values ('prodotto1'),
 ('prodotto2'),
('prodotto3')
```

Connettiamoci come 'cliente' e vediamo che è possibile aggiungere un cliente e un ordine ma non un prodotto

Cliente:

```
insert into clienti(username,password)
values ('luciano','123'),
 ('pierone','789');


insert into ordini (id_prodotto,cliente, quantita,data_ora)
values (1,'luciano',1,now())

insert into prodotti(descrizione)   -- non può --
values ('prodotto4');
```

Nel database ora creiamo una tabella “titolari” che contiene username e password del titolare (o dei titolari)

```
create table titolari
(username varchar(30) primary key not null,
 password varchar(256) not null);
```

E aggiungiamo un titolare

```
insert into titolari(username,password)
values ('titol1','123')
```

Questa tabella non è associata ad altre tabelle del database poiché non è richiesto di memorizzare informazioni sulle operazioni svolte dai titolari.
Questa tabella andrebbe collegata al database solo se si volesse, ad esempio, sapere quale titolare ha aggiunto un nuovo prodotto.
Questa tabella serve esclusivamente per permettere il login al sito web al titolare con un proprio username e password.
Questa tabelle può essere creata solo dal dbAdministrator (o dall’utente root).

Nelle pagine php per l’accesso al sito internet, quando l’accesso verrà effettuato da un cliente, la connessione avverrà con i seguenti parametri:

```
// Parametri di connessione al database
$server = "localhost";
$username = "cliente";
$password = "clie";
$dbname = "negozio";
```

Nelle pagine php per l'accesso al sito internet, quando l'accesso verrà effettuato da un titolare, la connessione avverrà invece con i seguenti parametri:

```
// Parametri di connessione al database
$server = "localhost";
$username = "titolare";
$password = "titol";
$dbname = "negozio";
```

Questo contribuirà a garantire una maggior protezione dei dati poiché se un malintenzionato potesse accedere ai dati del database attraverso un form, ad esempio con un attacco sqlinjection nel from di login del cliente, esso potrebbe svolgere sul database solo le operazioni consentite al profilo di cliente, quindi non potrebbe compiere operazioni di eliminazione dei dati.

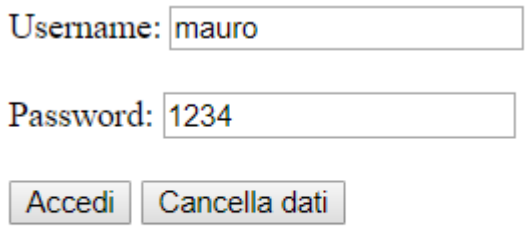
SQL INJECTION

E' una tipologia di attacco informatico nei confronti una base di dati che sfrutta la vulnerabilità di sicurezza del codice di una applicazione.

Consiste nell'iniettare (injection) del codice SQL malevolo all'interno di una query SQL presente nella pagina web dinamica attraverso un FORM.

Per chiarire i principi che stanno alla base di questa tipologia di attacchi facciamo un due esempi:

Consideriamo un form per il login ad un sito



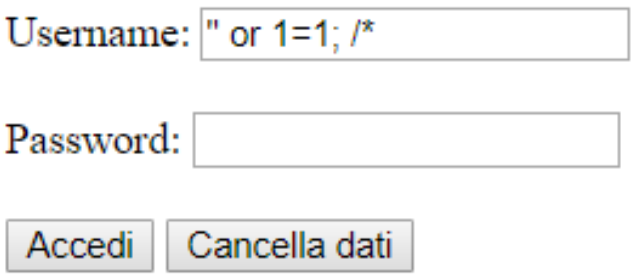
Ipotizziamo che nel database del sito vi sia una tabella utenti nella quale siano memorizzati Username e password degli utenti e che l'accesso al sito sia consentito in base al risultato della seguente query, nella quale sono stati riportati i valori inseriti dall'utente nel form

SELECT * FROM utenti WHERE username="mauro" AND password="1234"

Se la clausola WHERE fornisce risultato VERO l'utente ha accesso al sito altrimenti no.

La tecnica di SQL INJECTION consiste nell'inserire opportune stringhe nei campi del form in modo da "alterare" la query e far eseguire quindi del codice malevolo.

Ad esempio inserendo i seguenti valori nel form:



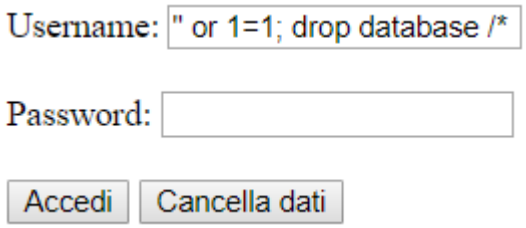
La query diventa:

SELECT * FROM utenti WHERE username=”” or 1=1; /* ” AND password=””

Questa parte diventa un commento SQL a causa di */ quindi viene ignorato dal DBMS

La clausola WHERE diventa VERA e concede l’accesso al sito web!

Esempio 2



la query diventa

SELECT * FROM utenti WHERE username=”” or 1=1; drop database; /* “AND password=””

In questo caso le query eseguite diventano due e la seconda elimina tutto il database (se la tipologia di utente ne ha il privilegio)!

(OSSERVAZIONE: utilizzando la libreria PHP mysqli quest’ultimo attacco non funziona poiché il comando $connection->query($query) non consente di inviare query multiple. Per inviare query multiple si utilizza il metodo multi_query() anzichè query).

Quelli visti sono gli esempi più semplici, in realtà esistono molte altre stringhe che consentono di ottenere numerose informazioni dal database, inoltre la sintassi delle stringhe che consentono l’ ”iniezione” può cambiare leggermente a seconda del DBMS utilizzato, comunque gli esempi chiariscono bene il funzionamento della tecnica della SQL INJIECTION.

In questo articolo sono riportate alcune stringhe che consentono di ottenere varie informazioni sul database: https://www.html.it/articoli/tecniche-sql-injection/

COME DIFENDERSI?

Non c'è un'unica soluzione, ci sono diverse "barriere" ciascuna delle quali contribuisce a ridurre la possibilità di query injection:

- Validare l'input negli script sia lato client sia lato server non consentendo l'inserimento di caratteri speciali (apice, doppio apice, asterisco ecc..)

- Gestire opportunamente i privilegi degli utenti (se un utente può eseguire solamente SELECT, chiunque esegua il login non potrà modificare il database)

- Utilizzare le **stored procedure SQL** per realizzare query parametriche