Hacer app web

ROMPER DISCO Y PODER RESTAURAR INFO

APRENDER A HACER BACKUP EN POSTGRES

Escenarios a probar

1)

Backup con pg\_dump de la base

Borrado de la carpeta $PGDATA, salida de servicio de postgres

2)

Backup con pg\_start\_backup tar of $PGDATA pg\_stop\_backup

Restauramos recuperando el backup def con tar xf y aplicando WALs

3)

pg\_start\_backup

tar cf $PGDATA

pg\_stop\_backup

Inscripciones web

Borrado de datos mediante SQL por parte de un usuario DELETE FROM sga\_alumnos;

Esto es un PITR (Point in Time Recovery), similar a un snapshot. Esto tambien sirve si no podemos hacer rollback de una transacción.

PG\_DUMP

* a nivel cluster: pg\_dumpall
* a nivel base, tabla, estructura,,,: pg\_dump
* Tipo texto / -Fc (comprimido) se restaura con pg\_restore

FILE SYSTEM LEVEL BACKUP (Bajar servicios de postgres, NO ES PRACTICO)

* Es un backup a nivel del file system
* Postgres debe estar bajo
* Es una foto , no es posible recuperar nada mas

BACKUP CONTINUO (USAR EN EL TP - OPTIMO)

* Permite hacer backup con los procesos andando
* Podemos hacer restore en cualquer momento

TAR (comando para copias de carpetas)

PG\_DUMP:

Puede backupear solo Datos (-a), solo Estructuras (-s) o Estructuras y datos

Formatos: -Fp (plano, es el default) es un .sql PESA MAS

Formato -Fc (custom, es comprimido y requiere del pg\_restore) No puedo verlo, pesa menos

pg\_dump ejemplos

Cuando se instala postgres, se crea un cluster de postgres.

Postrgres crea procesos, ocupa una parte de la ram (80% para el buffer pool aprox)

Los procesos interactuan con el buffer pool y el disco.

La primer bd es **postgres** (default)

pg\_dump trabaja a nivel disco.

pg\_dumpall (archivo sql con toda la info del cluster)

Backup Inicial

Segmentos Wal (Servidor remoto)

Ultimo WAL incompleto (AD)

SECURE COPY para pasar a un servidor remoto

**El profesor va a cambiar la clave de un usuario del sistema**

**Luego va a hacer un RM a la carpeta PGDATA (Borra toda la base de datos) desde root.**

**Las claves deben estar encriptadas.**

**ROOT DIRECTORY /home/demo/home -> archivo /etc/apache2/sites-enabled/000-default**

ENCRIPTAR PASSWORD

<https://alias.io/2010/01/store-passwords-safely-with-php-and-mysql/>

### What you should do

* Use a cryptographically strong hashing function like [bcrypt](https://en.wikipedia.org/wiki/Bcrypt) (see PHP's [crypt()](http://php.net/manual/en/function.crypt.php) function).
* Use a random salt for each password.
* Use a slow hashing algorithm to make brute force attacks practically impossible.
* For bonus points, regenerate the hash every time a users logs in.

$username **=** 'Admin';  
$password **=** 'gf45\_gdf#4hg';  
  
// A higher "cost" is more secure but consumes more processing power  
$cost **=** 10;  
  
// Create a random salt  
$salt **=** strtr(base64\_encode(mcrypt\_create\_iv(16, MCRYPT\_DEV\_URANDOM)), '+', '.');  
  
// Prefix information about the hash so PHP knows how to verify it later.  
// "$2a$" Means we're using the Blowfish algorithm. The following two digits are the cost parameter.  
$salt **=** sprintf("$2a$%02d$", $cost) **.** $salt;  
  
// Value:  
// $2a$10$eImiTXuWVxfM37uY4JANjQ==  
  
// Hash the password with the salt  
$hash **=** crypt($password, $salt);  
  
// Value:  
// $2a$10$eImiTXuWVxfM37uY4JANjOL.oTxqp7WylW7FCzx2Lc7VLmdJIddZq

In the above example we turned a reasonably strong password into a hash that we can safely store in a database. The next time the user logs in we can validate the password as follows:

$username **=** 'Admin';  
$password **=** 'gf45\_gdf#4hg';  
  
// For brevity, code to establish a database connection has been left out  
  
$sth **=** $dbh**->**prepare('  
 SELECT  
 hash  
 FROM users  
 WHERE  
 username = :username  
 LIMIT 1  
 ');  
  
$sth**->**bindParam(':username', $username);  
  
$sth**->**execute();  
  
$user **=** $sth**->**fetch(**PDO::**FETCH\_OBJ);  
  
// Hashing the password with its hash as the salt returns the same hash  
**if** ( hash\_equals($user**->**hash, crypt($password, $user**->**hash)) ) {  
 // Ok!  
}

A few additional tips to prevent user accounts from being hacked:

* Limit the number of failed login attempts.
* Require strong passwords.
* Do not limit passwords to a certain length (remember, you're only storing a hash so length doesn't matter).
* Allow special characters in passwords, there is no reason not to.

INYECCION SQL - <http://bobby-tables.com/>

<https://paragonie.com/blog/2015/05/preventing-sql-injection-in-php-applications-easy-and-definitive-guide>

<?php

// Connect to the database

$dbconn = pg\_connect('dbname=foo');

// Read in a text file (containing apostrophes and backslashes)

$data = file\_get\_contents('letter.txt');

// Escape the text data

$escaped = pg\_escape\_string($data);

// Insert it into the database

pg\_query("INSERT INTO correspondence (name, data) VALUES ('My letter', '{$escaped}')");

?>

<?php

$my\_data = pg\_escape\_string(utf8\_encode($\_POST['my\_data']));

?>