# BASE DE DATOS

Debido a la gran cantidad de datos pertenecientes al aeropuerto, será necesaria la creación de una base de datos que contenga esta información.

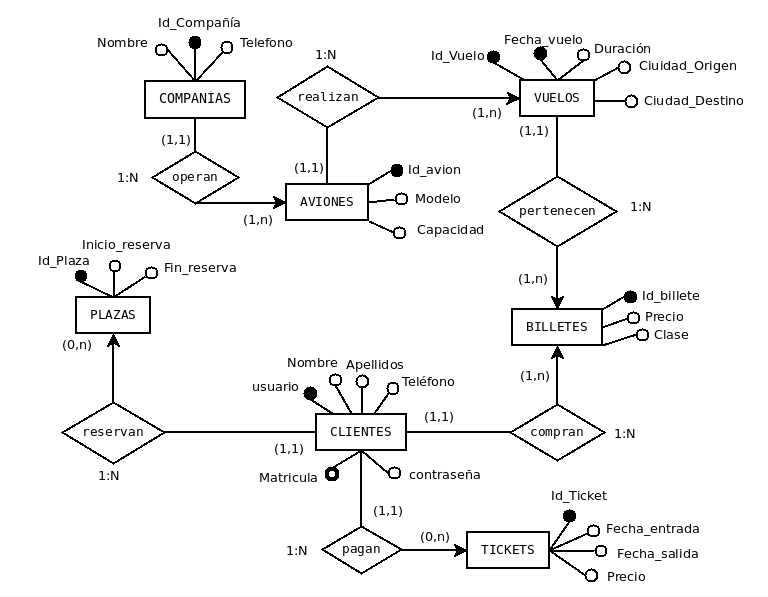
La base de datos será accesible desde la pagina web del aeropuerto y guardará los registros de los usuarios registrados en la pagina web, las compañias que operan en el aeropuerto, los vuelos diarios, entre otros.

Para poder gestionar la base de datos será necesario instalar en un servidor un sistema gestor de base de datos adecuado.

Se plantearon varias alternativas como Oracle, Microsoft Sql Server y MySql.

Al principio, la solución que más prometía era el SGBD MySql ,pero al final se optó por MariaDB. Este último, además de ser un software libre, ofrece un mejor rendimiento que MySql, mayor codificación de datos, así como herramientas de código abierto que sustituyen a aquellas propias de MySql.

## Modelo entidad relación



## Modelo relacional

compañias (id\_compañia, nombre, teléfono)

aviones (id\_avion, modelo, capacidad, id\_compañia)

vuelos (id\_vuelo, fecha\_vuelo, duración, ciudad\_origen,

ciudad\_destino, id\_avion)

billetes (id\_billete, precio, clase, id\_cliente)

clientes (id\_cliente, email, contraseña, nombre, apellido, matricula)

tickets (id\_ticket, matricula, fecha\_entrada, fecha\_salida, precio)

plazas (id\_plaza, matricula, inicio\_reserva, fin\_reserva)

## Instalación y configuración inicial

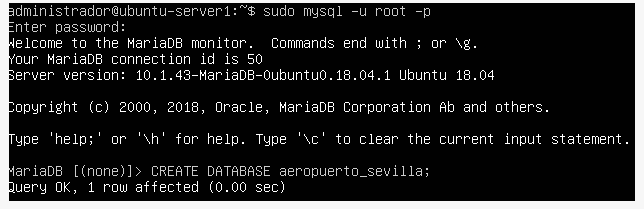
Instalamos el servidor y cliente de mariadb.



Una vez instalado el servidor, ejecutamos el siguiente comando para establecer una contraseña para el usuario root



Después, iniciamos sesión con el usuario root y creamos la base de datos del aeropuerto.



Una vez creada la base de datos creamos las tablas que la van a formar.

CREATE TABLE `aviones` (

`id\_avion` int(11) NOT NULL,

`modelo` varchar(20) DEFAULT NULL,

`capacidad` int(11) DEFAULT NULL,

`id\_compañia` int(11) DEFAULT NULL,

PRIMARY KEY (`id\_avion`),

KEY `fk\_aviones\_compañia` (`id\_compañia`),

CONSTRAINT `fk\_aviones\_compañia` FOREIGN KEY (`id\_compañia`) REFERENCES `compañias` (`id\_compañia`) ON DELETE NO ACTION ON UPDATE NO ACTION

);

CREATE TABLE `billetes` (

`id\_billete` varchar(30) NOT NULL,

`precio` decimal(5,2) DEFAULT NULL,

`clase` varchar(10) DEFAULT NULL,

`id\_cliente` int(11) DEFAULT NULL,

`id\_vuelo` int(11) DEFAULT NULL,

PRIMARY KEY (`id\_billete`),

KEY `fk\_billetes\_clientes` (`id\_cliente`),

CONSTRAINT `fk\_billetes\_clientes` FOREIGN KEY (`id\_cliente`) REFERENCES `clientes` (`id\_cliente`) ON DELETE NO ACTION ON UPDATE NO ACTION

);

CREATE TABLE `vuelos` (

`id\_vuelo` int(11) NOT NULL AUTO\_INCREMENT,

`fecha\_vuelo` date DEFAULT NULL,

`duracion` time DEFAULT NULL,

`ciudad\_origen` varchar(15) DEFAULT NULL,

`ciudad\_destino` varchar(15) DEFAULT NULL,

`id\_avion` int(11) DEFAULT NULL,

PRIMARY KEY (`id\_vuelo`),

KEY `id\_avion` (`id\_avion`),

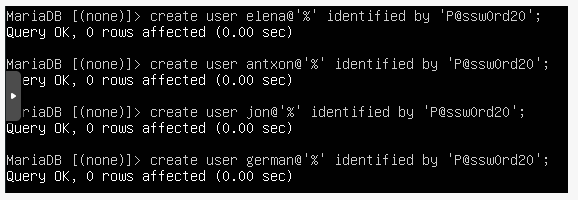
CONSTRAINT `vuelos\_ibfk\_1` FOREIGN KEY (`id\_avion`) REFERENCES `aviones` (`id\_avion`)

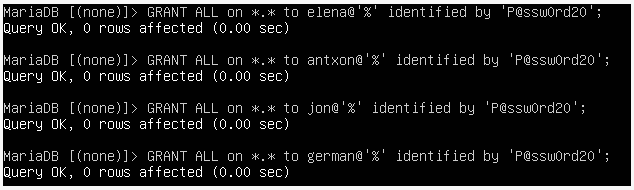
);

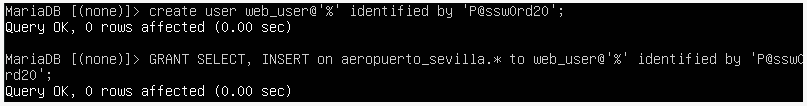
## 

## Creación de usuarios y asignar permisos

Vamos a crear diferentes usuarios que se encarguen de gestionar la base de datos del aeropuerto. Estos usuarios se dividen en administradores y usuarios web. Los primeros tendrán control total sobre la base de datos, mientras que los últimos únicamente podrán acceder a la base de datos para ingresar o modificar registros.



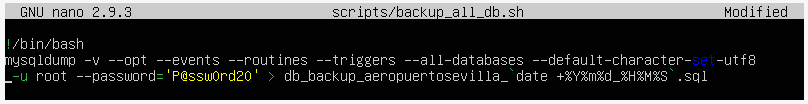




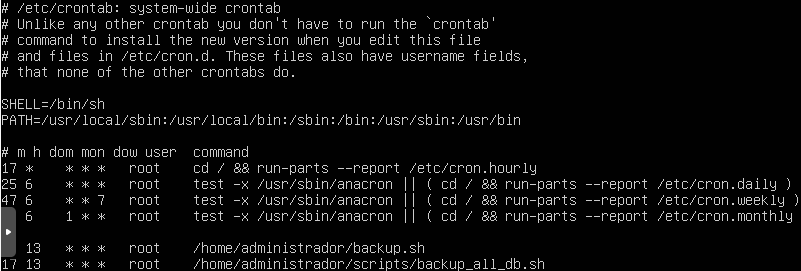
## Backups de la base de datos

Como medida de seguridad ante un posible error en el servidor y la consecuente pérdida de datos, se hará una copia diaria de la base de datos, tanto de las tablas como de otros elementos que la componen, como disparadores, índices, etc.

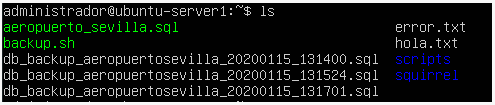
Para ello, se creará un script que realice dicha copia y la guarde en un archivo sql., cuyo nombre contendrá



Por último, se programará la ejecución del programa mediante crontab.



\*\*Comprobamos que se ha hecho el backup de la base de datos



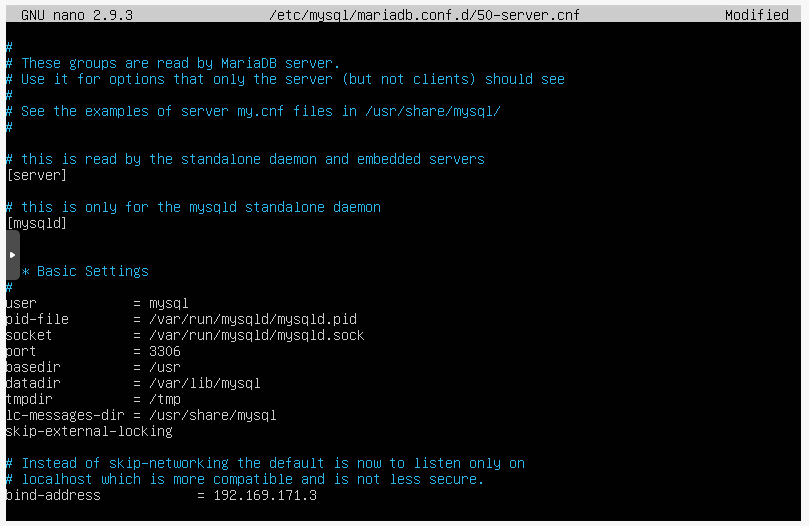
## Replicación

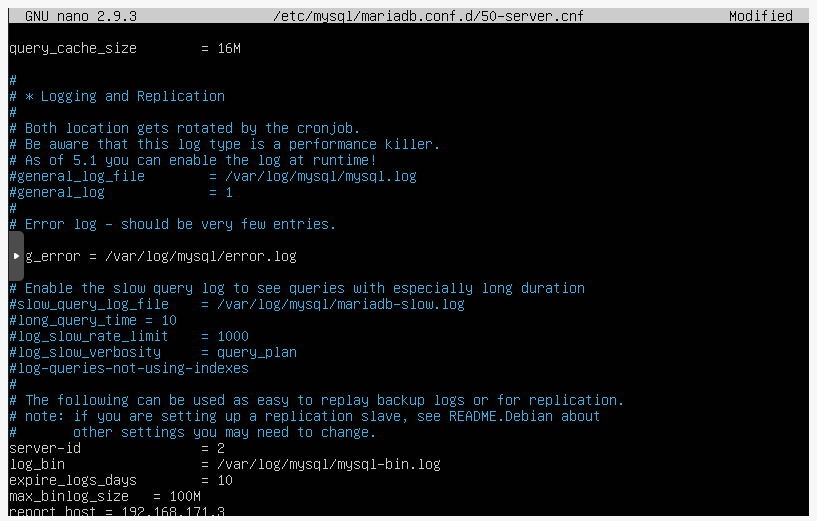
Dado que la base de datos va tener que estar siempre disponible, es necesario garantizar esa disponibilidad mediante la replicación del servidor de base de datos.

Con esto conseguiremos acceder a la base de datos incluso cuando un servidor deje de funcionar.

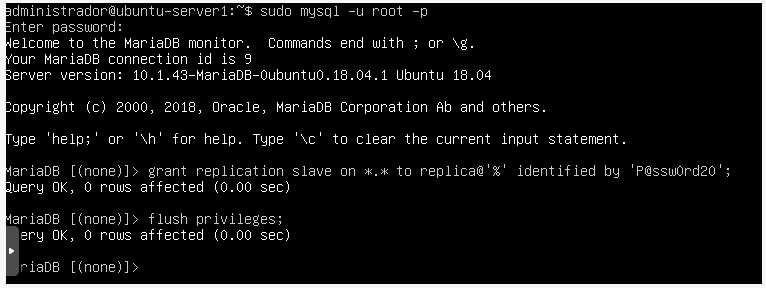
La base de datos podrá ser accesible por los usuarios y las modificaciones que se puedan realizar mientras el servidor principal no funcione se verán reflejadas en este cuando vuelva a la normalidad.

Primero, se debe modificar el fichero de configuración de mariadb en cada uno de los servidores, con el fin de prepararlos para la replicación. En este fichero se indicará el id de cada servidor y su IP.





Guardamos los cambios, iniciamos sesión en el cliente y creamos un usuario que tendrá permisos de replicación.

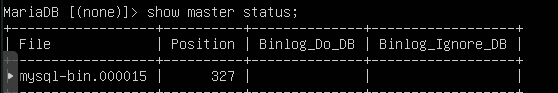


Una vez creado el usuario en los dos servidores, vamos a indicarle a los servidores cual es su maestro. Para ello les indicaremos el nombre del los log binarios\*\* del otro servidor y la posición de estos.

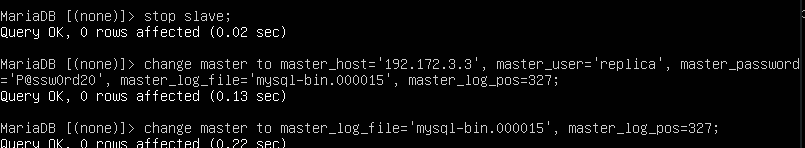
Estos logs contienen los eventos que representan las modificaciones del contenido de la base de datos.

Al indicarle a un servidor qué log de su maestro tiene que tener como referencia haŕa que sepa cuando se ha producido una modificación de la base de datos y que se guarden esos cambios en el propio serrvidor\*\*

Para sabe la ubicación de este log, ejecutamos el siguiente comando

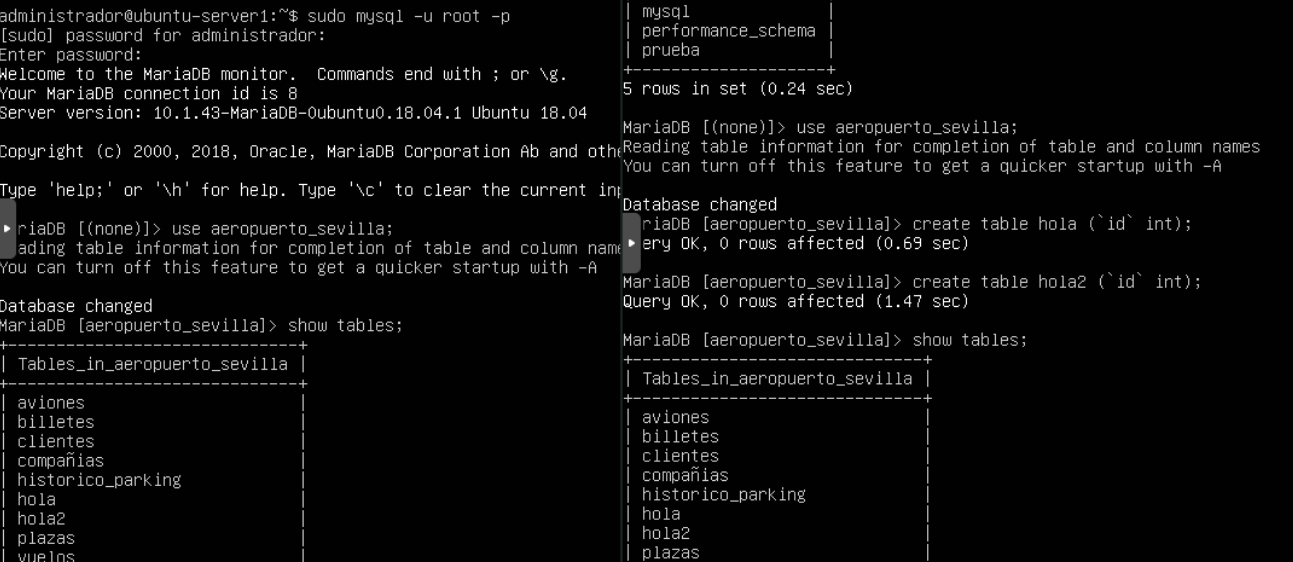


Y en el otro servidor realizamos la replicación, indicando la IP del servidor maestro, el usuario y contraseña de este y el fichero log y su posición.

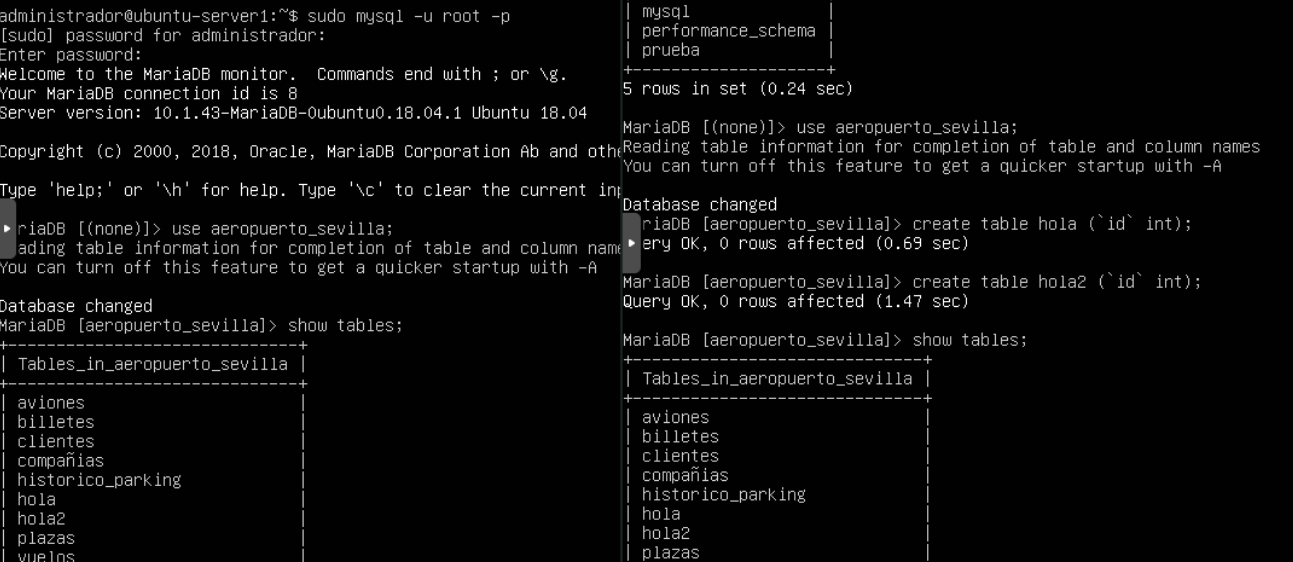


Ejecutamos los mismos comandos en el otro servidor y ya hemos completado la replicación de los dos servidores.

Para probar que la replicación funciona correctamente, creamos una tabla de prueba en uno de los servidores mientras el otro está apagado.



Al encender el otro servidor comprobamos que se han guardado los cambios realizados desde su maestro.



## TRAZABILIDAD DE DATOS

Debido al número de plazas del parking y la cantidad de reservas que se realizan cada día, es recomendable llevar un seguimiento de estas reservas. Hacer un seguimiento de estos datos no solo facilitará la resolución de problemas en los que estén invueltos, sino que ayudará a llevar un control de los registros.

Para ello, se creará una tabla histórico, que almacenará la información de cada reserva, así como información adicional que ayudará a resolver posibles errores.

CREATE TABLE `historico\_parking` (

`id\_historico` int(11) NOT NULL,

`id\_plaza` int(11) DEFAULT NULL,

`matricula` varchar(130) DEFAULT NULL,

`inicio\_reserva` date DEFAULT NULL,

`fin\_reserva` date DEFAULT NULL,

`date\_update` date DEFAULT NULL,

`cur\_user` varchar(15) DEFAULT NULL,

`cur\_host` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id\_historico`));

La información de las reservas se guardará automáticamente en el histórico mediante un trigger.

CREATE DEFINER=`elena`@`%` TRIGGER parking\_historico\_trg AFTER UPDATE ON plazas FOR EACH ROW

BEGIN

DECLARE max\_id int;

DECLARE vDate date;

DECLARE vUser varchar(15);

DECLARE vHost varchar(20);

SELECT max(id\_historico) INTO max\_id FROM historico\_parking;

SELECT CURDATE() INTO vDate;

SELECT SUBSTRING\_INDEX(user(), '@', 1) into vUser;

SELECT SUBSTRING\_INDEX(user(), '@', 2) into vHost;

IF max\_id IS NULL THEN SET max\_id = 0;

END IF;

IF old.matricula IS NOT NULL THEN

SET max\_id = max\_id+1;

INSERT INTO historico\_parking (id\_historico, id\_plaza, sector, matricula, inicio\_reserva, fin\_reserva, date\_update,

cur\_user, cur\_host)

VALUES (max\_id, old.id\_plaza, old.sector, old.matricula, old.inicio\_reserva, old.fin\_reserva, vDate, vUser, vHost);

END IF;

IF new.matricula IS NOT NULL THEN

SET max\_id = max\_id+1;

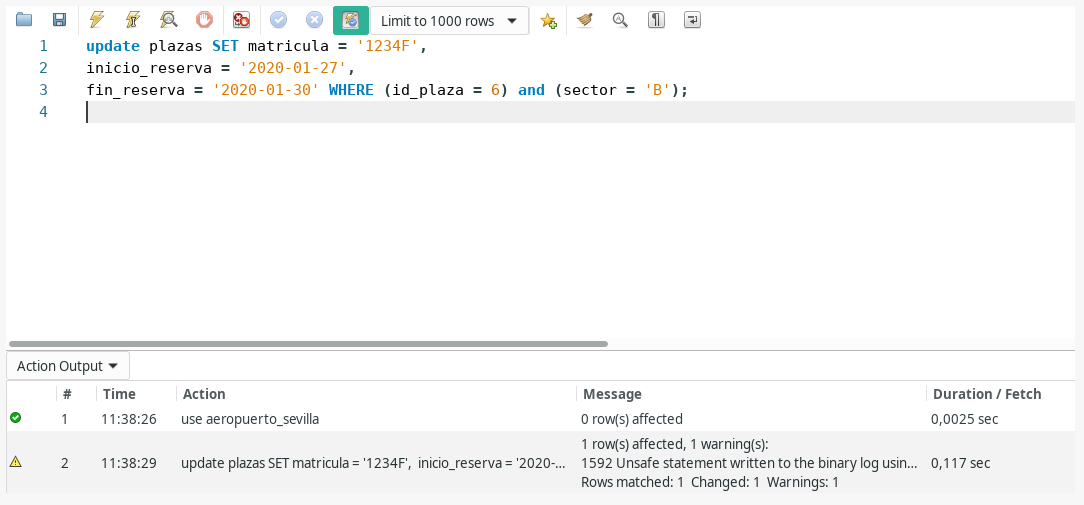
INSERT INTO historico\_parking (id\_historico, id\_plaza, sector, matricula, inicio\_reserva, fin\_reserva, date\_update,

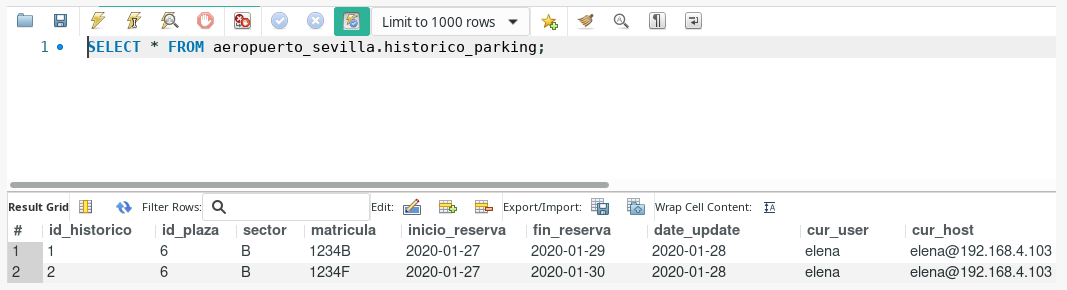
cur\_user, cur\_host)

VALUES (max\_id, new.id\_plaza, new.sector, new.matricula, new.inicio\_reserva, new.fin\_reserva, vDate, vUser, vHost);

END IF;

END





Por otra parte, es conveniente almacenar la información sobre los tiquets del parking que ya han expirado, lo que evitará posibles inconvenientes o errores a la hora del pago de estos tickets.

Para ello se creará un trigger que almacene en la siguiente tabla los registros de tickets expirados, indicando además la fecha en la que este expiró.

CREATE TABLE `historico\_tickets` (

`id\_historico` int(11) NOT NULL,

`id\_ticket` int(11) DEFAULT NULL,

`matricula` varchar(130) DEFAULT NULL,

`fecha\_entrada` datetime DEFAULT NULL,

`fecha\_salida` datetime DEFAULT NULL,

`precio` decimal(5,2) DEFAULT NULL,

`fecha\_expiracion` date DEFAULT NULL,

PRIMARY KEY (`id\_historico`)

);

CREATE DEFINER=`elena`@`%` TRIGGER `aeropuerto\_sevilla`.`tickets\_AFTER\_DELETE` AFTER DELETE ON `tickets` FOR EACH ROW

BEGIN

DECLARE max\_id int;

DECLARE vDate date;

SELECT max(id\_historico) INTO max\_id FROM historico\_tickets;

SELECT CURDATE() INTO vDate;

IF max\_id IS NULL THEN SET max\_id = 0;

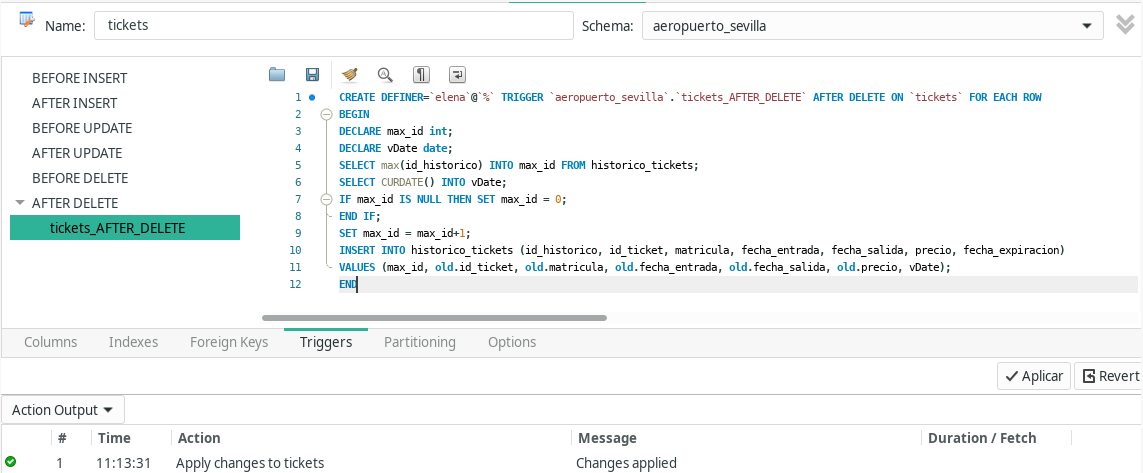
END IF;

SET max\_id = max\_id+1;

INSERT INTO historico\_tickets (id\_historico, id\_ticket, matricula, fecha\_entrada, fecha\_salida, precio, fecha\_expiracion)

VALUES (max\_id, old.id\_ticket, old.matricula, old.fecha\_entrada, old.fecha\_salida, old.precio, vDate);

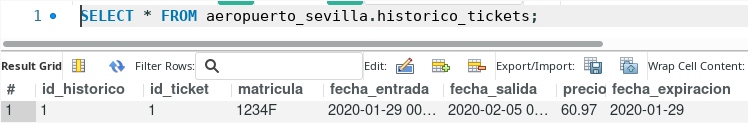
END



Para probar el buen funcionamiento del trigger, vamos a borrar el registro de un ticket ya expirado



Y comprobamos que se ha guardado dicho registro en el historico.



Por otra parte, en cuanto al aforo del parking, interesa controlar que no haya más de 400 reservas de plazas, ya que este es cupo máximo del parking.

Para garantizar que solo puedan realizarse tantas reservas como número de plazas, se ha creado un disparador que muestre un mensaje cuando el aforo del parking esté lleno.

