# Sistemas informáticos

# Práctica 2:

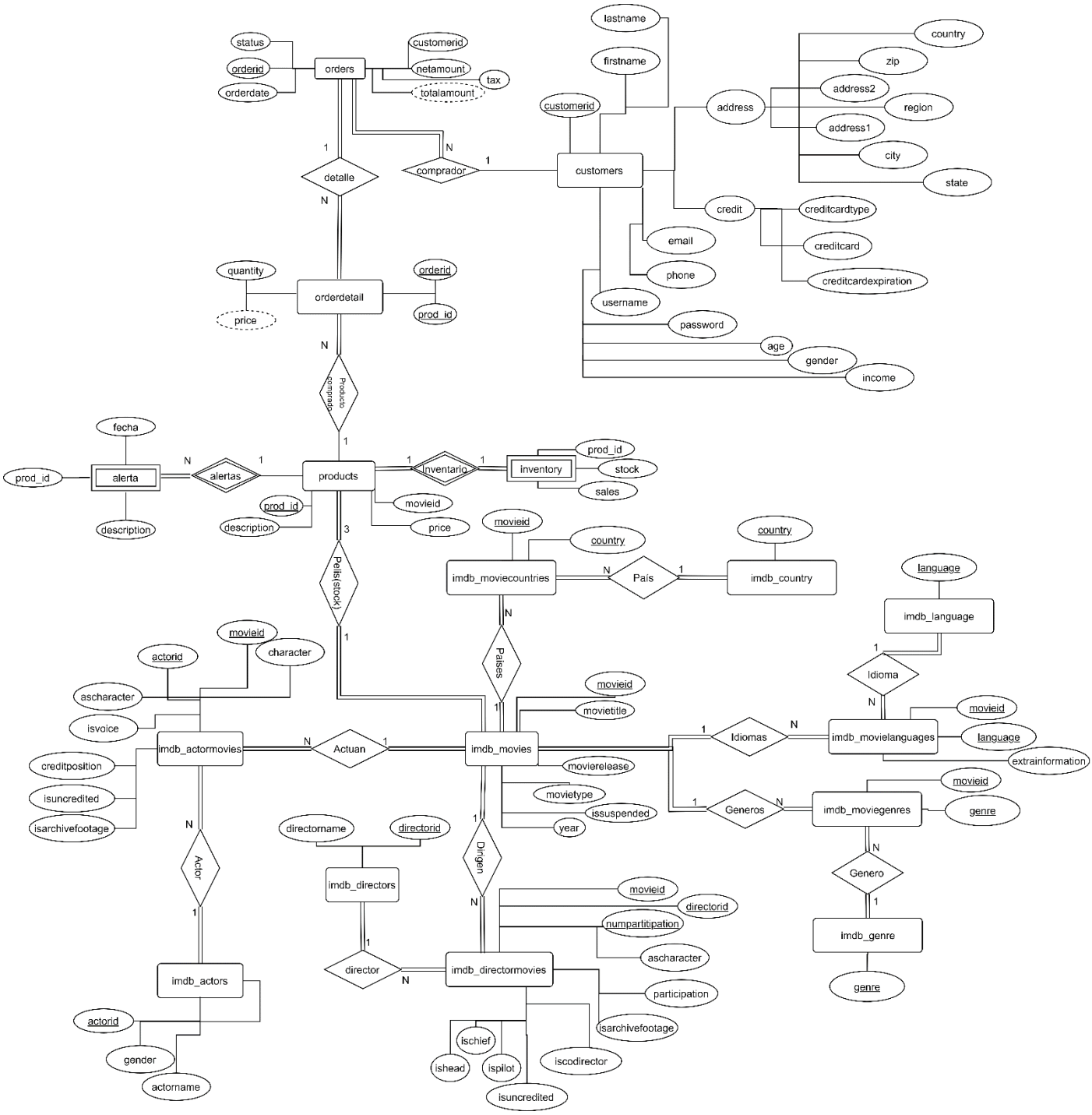
## Programación Web y Bases de Datos.

Autores:

Marcos Bernuy — Pareja 4
Kevin de la Coba Malam — Grupo 1391

# 1. Diseño de la BD

a. Actualiza.sql y diagrama entidad relación.




Este el diagrama entidad-relación de la base de datos. Si comenzamos con el customer podemos ver que tenemos varias columnas y una primary key, el id. Cada customer tiene N ordenes, las cuales en las cuales se definen las “compras”, tenemos un precio total neto y un precio total que depende del tax y del precio neto. Cada orden tiene N detalles asociados (productos). Las orderdetails están asociadas a su vez a un producto. Cada producto tiene una tabla inventario y una tabla alertas, el inventario muestra el stock de cada producto y las alertas se crean cuando se en un producto no hay stock (el trigger para esto no está implementado). Ahora tenemos imdb_movies, básicamente películas, la diferencia con la tabla products es que la tabla products corresponde a los productos que se pueden vender. Ahora cada movie tiene varios idiomas, países, géneros, directores y actores.

Las principales modificaciones de la base de datos fueron la creación de la tabla alertas, la tabla género y la tabla idioma. La tabla alertas tiene como primary key el id del producto al que se refiere y las tablas género e idioma tienen como primary key el propio género e idioma respectivamente, no son necesarias

más columnas en estas tablas. La creación de las tablas género e idioma periten tener en la base de datos únicamente un género e idioma único, no tendría mucho sentido tener el idioma repetido 500000 veces en la base de datos, simplemente con tenerla una vez y tener una tabla que nos relacione la movie con el género, lo mismo sucedería con los idiomas.

En esta primera versión de actualiza.sql simplemente debemos fijarnos en aspectos básicos como pueden ser las claves primarias y secundarias de cada tabla para la correcta comunicación entre ellas y que permitan la elaboración de consultas. Al comprobar que todas las claves secundarias estén de forma correcta también deberemos comprobar que las "constraints" o cambios de cascada estén en aquellas claves externas que deban tenerlo tanto para cuando se actualice la clave primaria como para cuando se elimine. Un ejemplo sería que al eliminar un producto de id="2" mantener la película que corresponda a un producto desconocido (ya se ha eliminado el producto) no tendría sentido y por tanto deberíamos establecer la clave secundaria como "on delete cascade".

Durante este proceso para una tabla denominada "orderdetail" nos damos cuenta que no podemos establecer correctamente las claves primarias, ya que están duplicadas, y las claves primarias deben ser siempre únicas. Para solucionar este problema, primero debemos deshacernos de las duplicaciones con la siguiente consulta:

```
DELETE FROM orderdetail WHERE orderid IN(
    SELECT orderid
    FROM(
        SELECT orderid, ROW_NUMBER() OVER (PARTITION BY orderid, prod_id ORDER BY
orderid) AS row_num
        FROM orderdetail
    ) AS t
    WHERE t.row_num > 1
);
```

Tras esta eliminación de tuplas duplicadas, y configuración correcta de tanto claves primarias como secundarias, solo nos falta cambiar el valor de una fecha, para que no de problemas al ejecutar "setPrice.sql". Para ello usamos la siguiente consulta:

```
UPDATE imdb_movies SET year = '1999' WHERE year = '1998-1999';
```

Por último recordar que al configurar las claves primarias y secundarias, también se deberá comprobar que no haya "constraints" que no hagan falta.

b. Consultas, procedimientos almacenados, triggers y funciones.

- **setPrice.sql**

```
UPDATE orderdetail
SET price = a.price/POWER(1.02, 2020-CAST(b.year AS INT) )
FROM products AS a, imdb_movies AS b
WHERE orderdetail.prod_id = a.prod_id AND b.movieid = a.movieid

-- Si en 2020 una pelicula vale 30€ ¿Cuanto valía en los 2000 si el precio ha ido
aumentando un 2%?
-- 2020 = 30,00 | 2000 = ¿x? |-> 30 = x * 1,02²⁰²⁰⁻²⁰⁰⁰ -> x = 30/1,02²⁰²⁰⁻²⁰⁰⁰
```

Para hacer esta consulta lo primero que debemos hacer es una regla de 3 (no exactamente). Nos dicen que los precios actuales son el resultado de un incremento anual del 2%, por lo tanto, si sabemos el año en el que esa película se hizo podemos calcular el precio original:

$$2020 = 30€, 2000 = ¿x?.$$

Sabiendo que el incremento es de un 2% cada año, calculamos la diferencia entre el año de la película y el actual, y elevamos 1,02 (el incremento) a esa diferencia.

$$30 = x * 1{,}02^{2020-2000}, x = \frac{30}{1{,}02^{2020-2000}}$$

Esta última operación es la que se ejecuta en la consulta. Hacemos update de los orderdetails donde haya productos que comparten una misma movieid.

- **setOrderAmount.sql**

```
CREATE OR REPLACE PROCEDURE setOrderAmount()

LANGUAGE 'plpgsql'

AS $$

BEGIN
        UPDATE   orders as a
        SET      netamount = t.sumprice,
                 totalamount = t.sumprice+t.sumprice*(tax/100)
        FROM     (
                         --Consulta para obtener el precio total de cada venta
                         SELECT sum(price_by_quantity.prc_of_each_detail) AS sumprice,
price_by_quantity.order_id_per_detail AS ord_id
                         FROM orders AS a, (
                                                  --Consulta para obtener el precio*quantity de
cada producto
                                                  SELECT price*quantity as prc_of_each_detail,
orderid as order_id_per_detail
                                                  FROM orderdetail
                                         ) as price_by_quantity
                         WHERE price_by_quantity.order_id_per_detail = a.orderid
                         GROUP BY a.orderid, price_by_quantity.order_id_per_detail
                ) AS t
        WHERE    t.ord_id = a.orderid AND (totalamount IS NULL AND netamount IS NULL);

END;
$$;

CALL setOrderAmount();
```

En este apartado creamos un procedimiento almacenado en la base de datos. Este procedimiento debe actualizar los campos *netamount* y *totalamount* de la tabla *orders*. El primer paso de la consulta es el de obtener el precio de cada orderdetail. Cada orderdetail tiene un producto, pero puede haberse comprado varias veces, por lo que el precio de cada orderdetail es la cantidad de productos multiplicada por el precio unitario del producto. Una vez tenemos el precio del orderdetail, tenemos que sumar todos los precios de los orderdetail que pertenezcan a un mismo order. La suma de estos orderdetails es el *netamount* de orderdetail (precio neto, sin impuestos), el *totalamount* es el *netamount* + *netamount*tax* (* el impuesto de cada orden). El *tax* esta en formato % por lo que debemos dividirlo entre 100.

- **getTopVentas.sql**

```
DROP FUNCTION gettopventas(integer,integer);

CREATE OR REPLACE FUNCTION getTopVentas (year_1 INTEGER, year_2 INTEGER)
RETURNS TABLE (
ANO              DOUBLE PRECISION,
PELICULA VARCHAR,
VENTAS           NUMERIC
)
AS $$

BEGIN

        RETURN QUERY
                -- Una vez numeradas las filas solo cogemos la fila numero 1
                SELECT t2.Oano2 AS ANO, b.movietitle AS TITULO, t2.sumadecantidades2 AS VENTAS

                FROM     (
                         -- Enumeramos cada fila pero agrupandolas por año
                         SELECT t1.sumadecantidades1 AS sumadecantidades2, t1.Oano1 AS Oano2,
t1.IMDB_Mid1 AS IMDB_Mid2,
                         ROW_NUMBER() OVER(PARTITION BY t1.Oano1 ORDER BY t1.sumadecantidades1
DESC) AS rk
                         FROM     (
```

```
                                -- Sumamos las versiones de las peliculas
                                SELECT SUM(t0.sumadecantidades) AS sumadecantidades1, t0.Oano0
as Oano1, t0.IMDB_Mid0 AS IMDB_Mid1
                                FROM     (
                                        -- Sumanmos las cantidades de los orderdetails y las
separamos por año
                                        SELECT SUM(t.cantidad) AS sumadecantidades, t.ODp_id
AS ODp_id2, t.Oano as Oano0, t.IMDB_Mid AS IMDB_Mid0
                                        FROM     (
                                                -- Seleccionamos todos los orderdetails y los
orders con los años
                                                SELECT d.movieid AS IMDB_Mid, c.prod_id AS
Pid, b.orderid AS ODoid, b.prod_id AS ODp_id, b.quantity AS cantidad, EXTRACT(YEAR FROM
a.orderdate) AS Oano
                                                FROM orders AS a, orderdetail AS b, products
AS c, imdb_movies as d
                                                WHERE a.orderid = b.orderid AND (
CAST(EXTRACT(YEAR FROM a.orderdate) AS INTEGER) BETWEEN year_1 AND year_2 )
                                                        AND d.movieid = c.movieid AND
c.prod_id = b.prod_id
                                                ) AS t
                                        GROUP BY ODp_id2, Oano0, IMDB_Mid0
                                        ORDER BY sumadecantidades DESC
                                        ) AS t0
                                GROUP BY Oano1, IMDB_Mid1
                                ) AS t1, imdb_movies AS c
                        GROUP BY Oano2, IMDB_Mid2, sumadecantidades2
                        ) AS t2, imdb_movies AS b
                -- Seleccionamos solo la primera
                WHERE rk = 1 AND b.movieid = t2.IMDB_Mid2
                ORDER BY VENTAS DESC;

END; $$

LANGUAGE 'plpgsql';
```

Aquí se nos pide crear una función para obtener las películas más vendidas entre dos años.
Recomendamos ver el código de la función en el archivo getTopVentas.sql, ya que aquí no se puede ver bien.
Lo primero para resolver esta consulta es tener en cuenta que buscamos las *imdb_movies* no los *products*, este fue un error que tuvimos hasta que nos dimos cuenta de que una película puede tener de 1 a 3 productos asociados dependiendo las versiones (standard, gold y ultra).
Teniendo esto claro, debemos obtener las ordenes que se hallan hecho entre los años especificados, nosotros decidimos obtener también en este paso la id de las películas, de los productos y de las ordenes para poder ir agrupándolos paso a paso, y por último la cantidad de productos por orderdetail.
En el siguiente paso agrupamos las orderids de orderdetail que tengan un mismo prod_id.
Repetimos el proceso y agrupamos los productos que compartan el movieid.
Ahora ya tenemos cada movieid con la cantidad de productos vendidos entre los años establecidos, pero se nos pide la más vendida de cada año ordenadas de mayor a menor. Para esto enumeramos por grupos las películas en orden descendente (las películas de 2017 serán numeradas de 1 a x dependiendo de cuantas películas se hayan vendido, lo mismo para las de 2018…). Una vez numeradas las películas solo cogemos aquellas que tengan el número 1 en cada año.

- **getTopMonths.sql**

```
CREATE OR REPLACE FUNCTION getTopMonths (num_products_umbral INTEGER, importe_umbral INTEGER)
RETURNS TABLE (
IMPORTE          NUMERIC,
MES              DOUBLE PRECISION,
ANO              DOUBLE PRECISION,
PRODUCTOS        NUMERIC
)
AS $$

BEGIN

RETURN QUERY
        -- Sumamos los precios y la cantidad de productos agrupandola en meses
        SELECT SUM(t.importe_consulta), t.mes_consulta as mes, t.ano_consulta as ano,
SUM(t.productos_consulta)
```

```
        FROM     (
                 -- Agrupamos los orderdetails haciendo un join con ordeid por año y mes.
Obtenemos los precios y la cantidad de productos en cada order
                 SELECT sum(a.totalamount) AS importe_consulta, EXTRACT(MONTH FROM a.orderdate)
AS mes_consulta, EXTRACT(YEAR FROM a.orderdate) AS ano_consulta, COUNT(b.orderid)*quantity as
productos_consulta
                 FROM orders as a, orderdetail as b
                 WHERE b.orderid = a.orderid
                 GROUP BY EXTRACT(YEAR FROM orderdate), orderdate, quantity
                 ) AS t
        GROUP BY mes, ano
        HAVING SUM(t.importe_consulta) > importe_umbral OR SUM(t.productos_consulta) >
num_products_umbral;

END; $$

LANGUAGE 'plpgsql';

SELECT * FROM getTopMonths(19000, 320000)
```

Aquí tuvimos que crear otra función para obtener la cantidad de productos y el importe total de los meses donde se superen ciertos umbrales.

El primer paso fue obtener el precio total por cada mes. En este paso no se obtiene el numero real de productos por mes (si se ejecuta únicamente este paso se pueden ver repeticiones en los meses y años), para obtener el resultado real, es necesario agrupar el primer resultado por mes y año, sumando los importes y los productos de la anterior.

Añadimos un *having* para filtrar las filas que no cumplan nuestras restricciones.

# 2. Integridad de los datos.

a. Actualiza.sql.

En esta segunda parte de la elaboración de “actualiza.sql” debemos crear nuevas tablas, en concreto las tablas “imdb_language”, “imdb_genre”, “imdb_country” y “alerta”. En el caso de las tablas “imdb_”, la creación de estas tiene el objetivo de eliminar elementos multivaluados. La implementación y creación seguirá aquella que hay entre “imdb_movies” y “imdb_actors” o “imdb_directors” ya que ambas relaciones tienen una tabla auxiliar que permite la eliminación de estos valores multivaluados. Para ello realizaremos la siguiente consulta, que puede ser aplicada a cada tabla:

```
CREATE TABLE imdb_language(
    language varchar primary key
);

INSERT INTO imdb_language(
    SELECT DISTINCT language
    FROM imdb_movielanguages
);

ALTER TABLE imdb_movielanguages ADD FOREIGN KEY (language) REFERENCES
imdb_language(language) ON DELETE CASCADE ON UPDATE CASCADE;
```

Una vez se ha realizado la creación y la inserción de las tablas restantes, debemos crear una nueva tabla alerta que nos permitirá identificar cuando un producto en concreto está vacío en stock. Para ello usaremos la siquiente consulta:

```
CREATE TABLE alerta(
    prod_id int REFERENCES products(prod_id) ON DELETE CASCADE ON UPDATE CASCADE,
    description text NOT NULL,
    fecha date NOT NULL
);
```

b. Triggers.

- **updOrders.sql**

```
DROP TRIGGER IF EXISTS updOrders ON orderdetail;

CREATE OR REPLACE FUNCTION updOrders()
RETURNS TRIGGER AS
$$
        BEGIN
                IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE') THEN
                        UPDATE orders
                        -- Actualizamos el valor con el precio correcto
                        SET netamount = t.precio, totalamount = t.precio+t.precio*((tax/100))
                        FROM
                                (
                                -- Calculamos el total de la orden
                                SELECT SUM(t0.total) AS precio
                                FROM
                                        (
                                                -- Calculamos precio total de producto*cantidad
                                                SELECT quantity*price AS total
                                                FROM orderdetail
                                                WHERE orderid = NEW.orderid
                                        ) as t0
                                ) AS t
                        WHERE NEW.orderid = orderid;

                        RETURN NEW;
                        -- En el caso de que borremos tenemos que no tener en cuenta el id del product borrado
                ELSIF (TG_OP = 'DELETE') THEN
                        UPDATE orders
                        SET netamount = t.precio, totalamount = t.precio+t.precio*((tax/100))
                        FROM
                                (
                                --
                                SELECT SUM(t0.total) AS precio
                                FROM
```

```
                                                (
                                                        -- Calculamos precio total de producto*cantidad
                                                        SELECT quantity*price AS total
                                                        FROM orderdetail
                                                        -- Excluimos el producto borrado
                                                        WHERE orderid = NEW.orderid AND prod_id != NEW.prod_id
                                                ) as t0
                                        ) AS t
                                WHERE NEW.orderid = orderid;

                                RETURN NEW;
                        ELSE
                                RETURN NULL;
                        END IF;
                END
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER updOrders
AFTER UPDATE OR INSERT OR DELETE ON orderdetail
FOR EACH ROW
EXECUTE PROCEDURE updOrders();
```

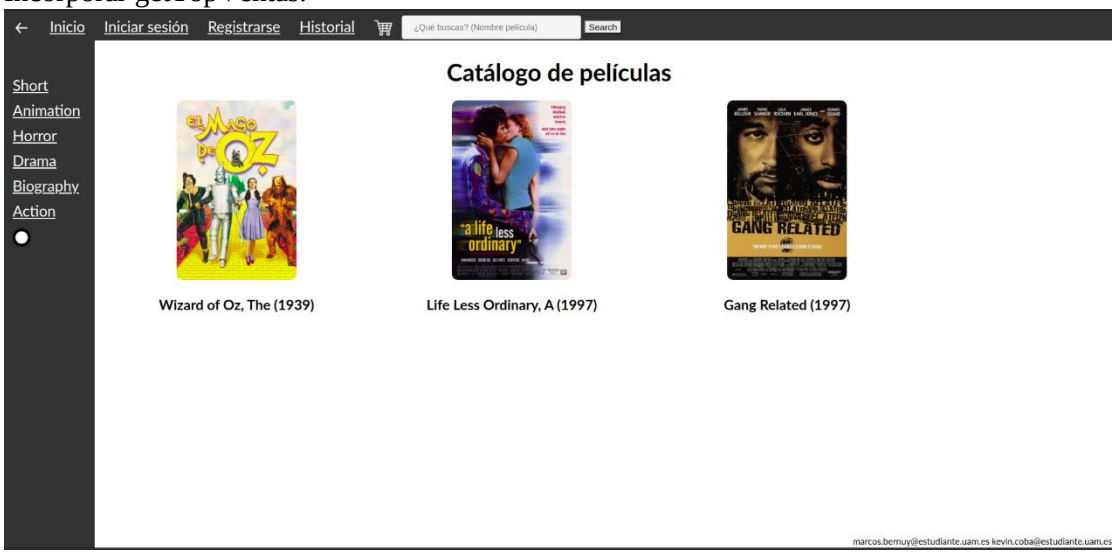
En este apartado se nos pide hacer un trigger que se active cuando se cree, actualice o borre un orderdetail y cambie los valores de *netamount* y *totalamount*. Un error que cometimos fue el de agrupar en el mismo if las tres condiciones mencionadas (actualizar, crear y borrar), esto es un error ya que siempre estábamos consultando el NEW, y al borrar, NEW tiene sus valores a NULL. Para solucionar esto dividimos el código en 2, actualizar y crear, y borrar. En la parte de actualizar y crear, cogemos NEW y sus campos con los valores correctos del orderdetail y actualizamos la order con el precio total de todos los orderdetail que lo componen, en la parte de borrar, debemos excluir de la suma el producto borrado para obtener el precio correcto.

- **updInventory.sql**

No realizado.

# 3. Integración del portal

a. Incorporar getTopVentas.

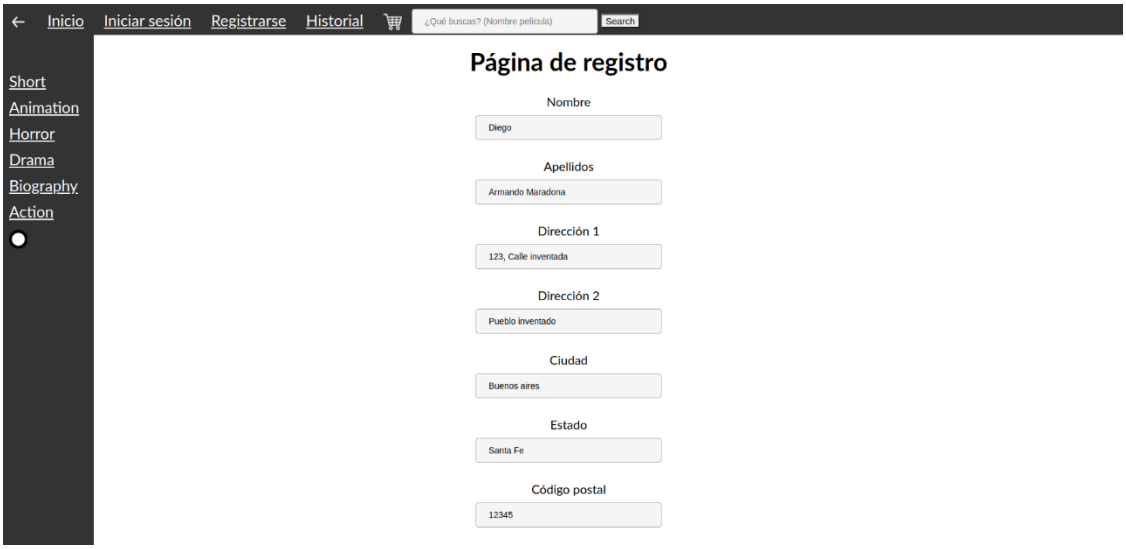


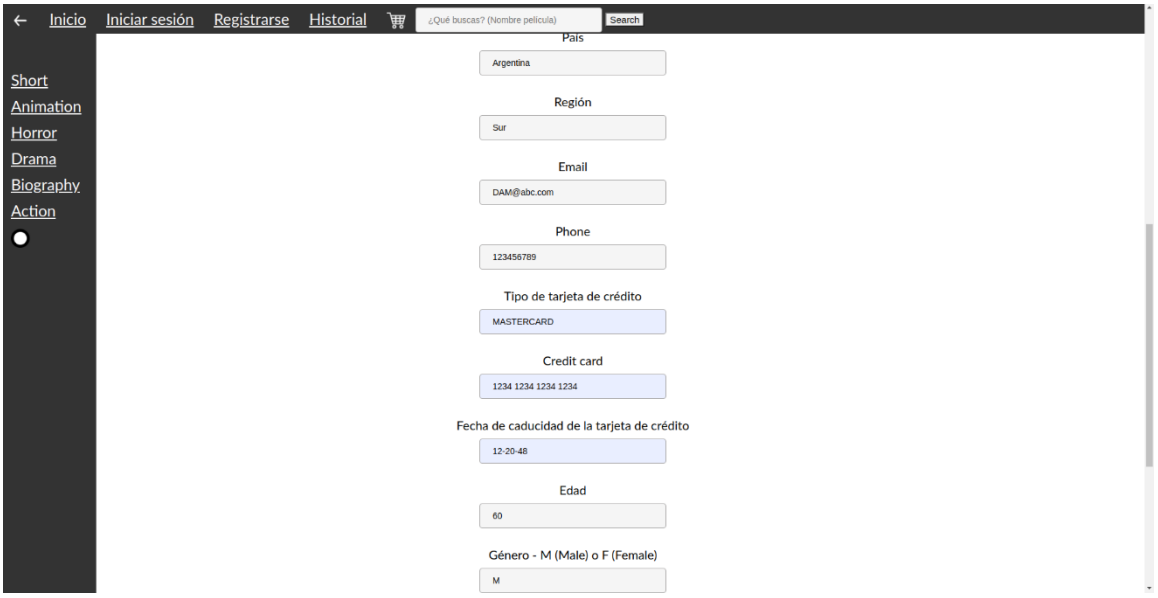

Usando sqlAlchemy ejecutamos la función para obtener las películas más vendidas, pero nos dimos cuenta que tardaba demasiado, por lo que simplemente tenemos una lista con los resultados de la consulta y eso es lo que se muestra.

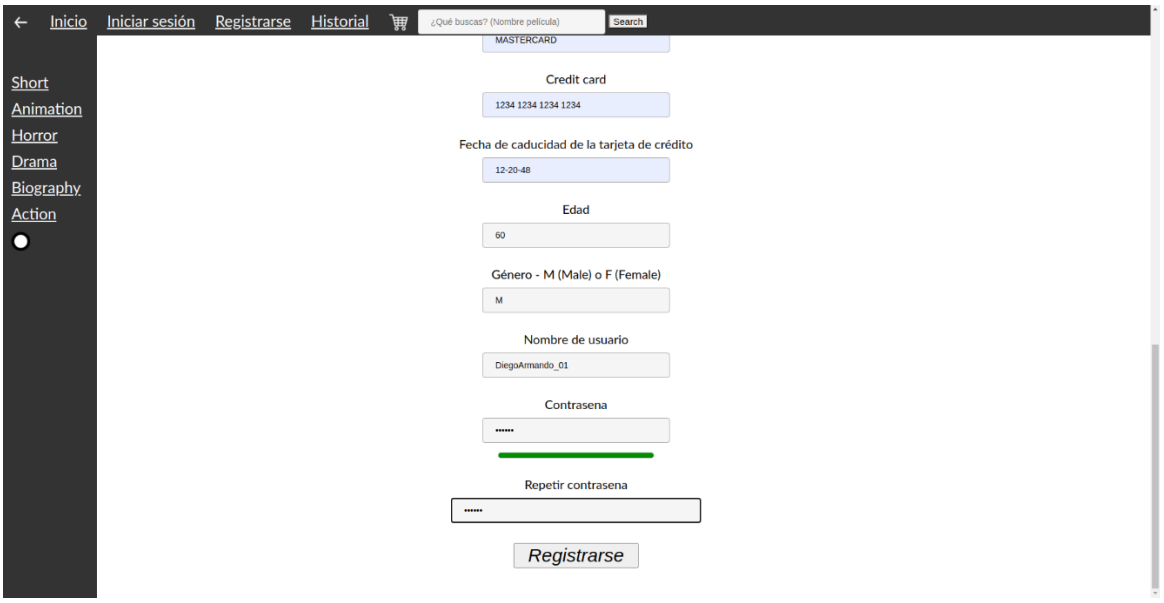
```
@app.route('/')
@app.route('/index')
def index():
    top_films = [[442893, 'Wizard of Oz, The (1939)', '', 0, '1939', 0],
                 [229764, 'Life Less Ordinary, A (1997)', '', 0, '1997', 0],
                 [149475, 'Gang Related (1997)', '', 0, '1997', 0]]
    #top_films = database.db_top_films()
    if top_films == False:
        return
    stack_push(request.url)
    return render_template('index.html', movies=top_films, logged=logged())
```

La función comentada es completamente funcional, se puede usar pero cada vez que se quiera acceder a la página de inicio tardara 15 segundos.

b. Registro y login utilizando la base de datos.
En las próximas imágenes se puede ver el proceso de registro y el resultado en la base de datos.






Al pulsar el botón registrar se ejecuta el código donde se inserta en la tabla customers, el customer que acabamos de crear.

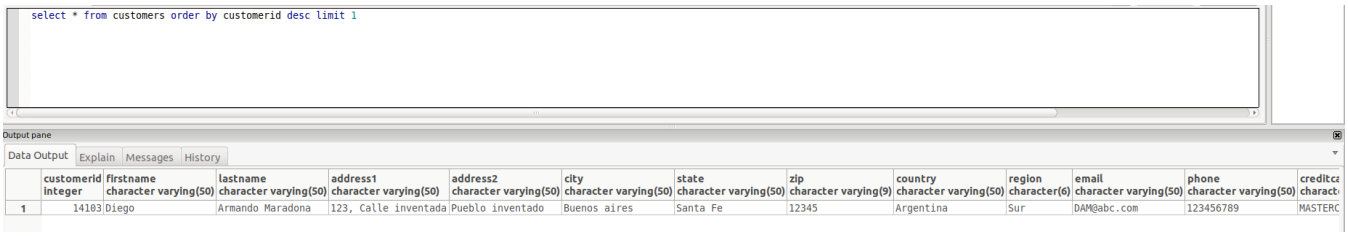
```
        age = 'null'
    if len(age) > 50:
        return render_template('signup.html', title='signup', logged=logged(), error="Los datos no fueron introducidos correctamente")

    gender = request.form['gender']
    if gender == '':
        gender = 'null'
    else:
        gender = "'"+gender[:1]+"'"


    creditcard = creditcard.replace(' ', '')

    if database.registrar(firstname, lastname, address1, address2,
                          city, state, zipcode, country, region, email,
                          phone, creditcardType, creditcard, creditcardexpiration,
                          username, password, age, gender) == False:
        session.permanent = False
        session['usuario'] = username
        return render_template('signup.html', title='signup', logged=logged(), error="Ya existe ese usurname o hubo un error")
    else:
        session.permanent = False
        session['usuario'] = username
        return redirect(url_for('index'))
```

Para comprobar el correcto funcionamiento podemos ejecutar una consulta en la base de datos.

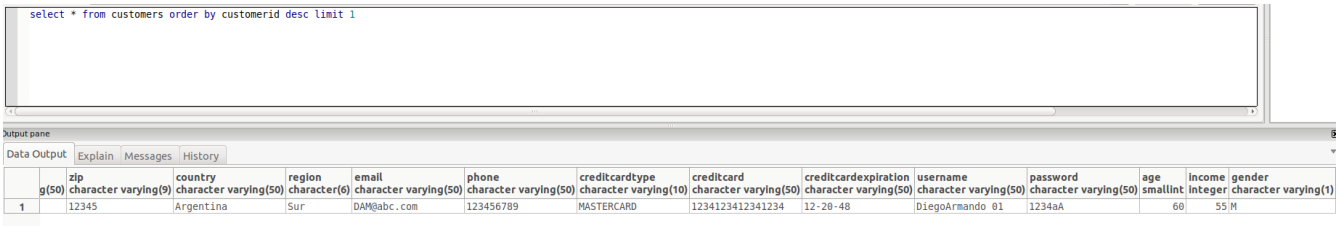


```
select * from customers order by customerid desc limit 1
```

Output pane

Data Output | Explain | Messages | History

| customerid integer | firstname character varying(50) | lastname character varying(50) | address1 character varying(50) | address2 character varying(50) | city character varying(50) | state character varying(50) | zip character varying(9) | country character varying(50) | region character(6) | email character varying(50) | phone character varying(50) | creditca characte |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 14103 | Diego | Armando Maradona | 123, Calle inventada | Pueblo inventado | Buenos aires | Santa Fe | 12345 | Argentina | Sur | DAM@abc.com | 123456789 | MASTERC |



```
select * from customers order by customerid desc limit 1
```

Output pane

Data Output | Explain | Messages | History

| g(50) | zip character varying(9) | country character varying(50) | region character(6) | email character varying(50) | phone character varying(50) | creditcardtype character varying(10) | creditcard character varying(50) | creditcardexpiration character varying(50) | username character varying(50) | password character varying(50) | age smallint | income integer | gender character varying(1) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 12345 | Argentina | Sur | DAM@abc.com | 123456789 | MASTERCARD | 1234123412341234 | 12-20-48 | DiegoArmando 01 | 1234aA | 60 | 55 | M |

Podemos ver en el resultado que los datos son los introducidos y que los tenemos en la base de datos.

Una de las decisiones que tomamos a la hora de crear el customer es la de coger la última customerid, sumarle 1 y asignarle al usuario esta, así evitaríamos la repetición de las ids.

Para hacer login se ejecuta un proceso similar, en este caso la query que ejecutamos es para buscar el username y comprobar si la password es la correspondiente.


Si introducimos los valores y pulsamos el botón login, se ejecutará el código para comprobar dichos datos en la base de datos.

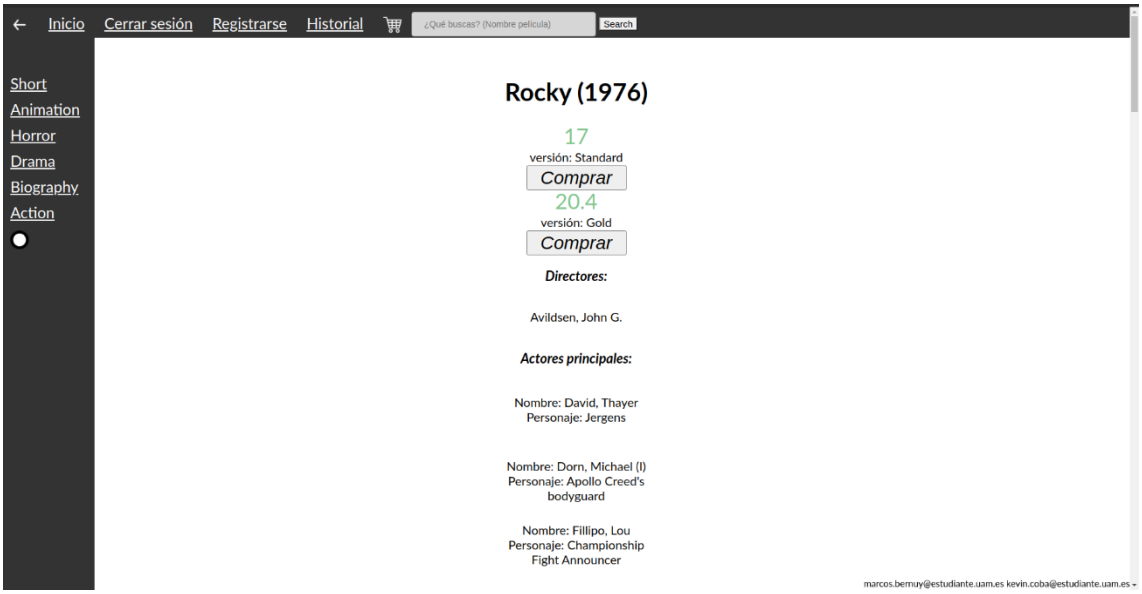
El código con respecto a estas funcionalidades es muy simple, simplemente ejecutamos consultas para ver si los usuarios existen y para insertar en la tabla. Las funciones que se crearon para esto son:

- **validar**. Dados un *username* y una *password*, buscamos ambos elementos en la base de datos y miramos si uno corresponde al otro.
- **registrar**. Dados los campos introducidos en la página de registro creamos un customer con estos si el username no existe. Con respecto a los campos que no son NOT NULL, se pueden dejar vacíos.

c. Implementación del resto de películas en el portal.
Para poder visualizar las películas correctamente creamos las siguientes funciones:

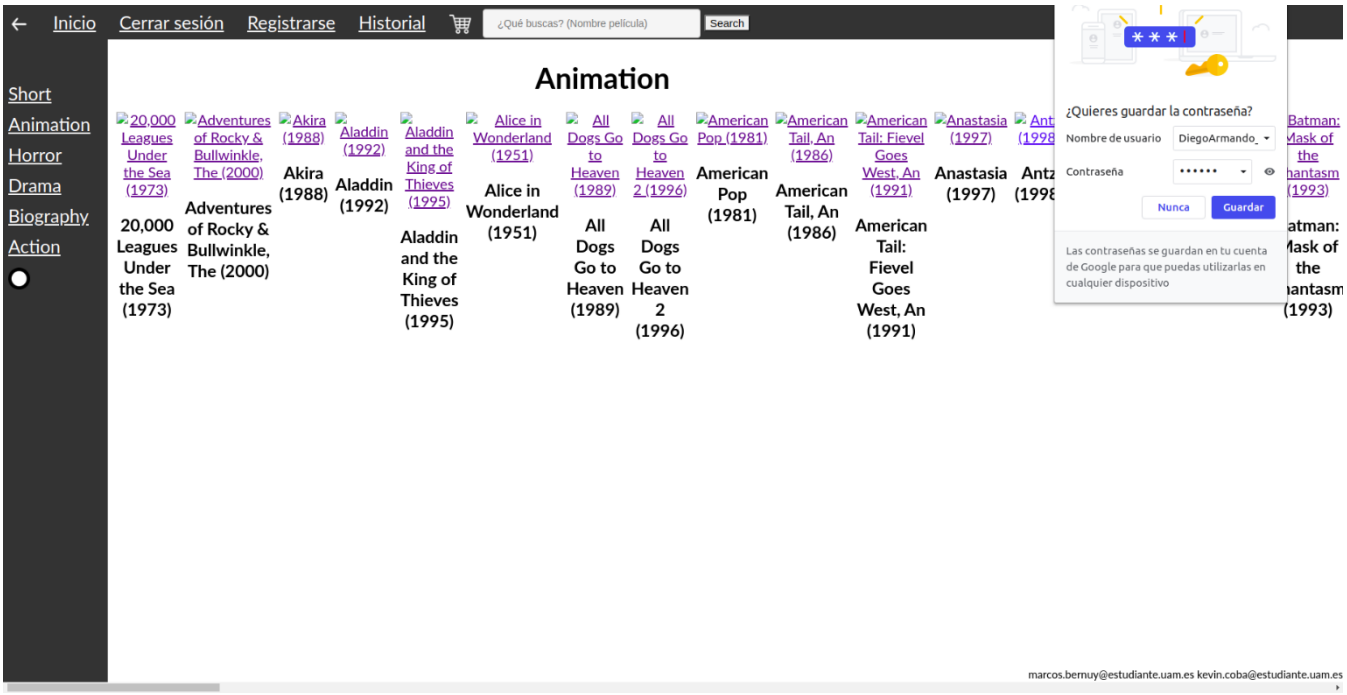
- **getPelicula**. Pasando la id de una película, la función devuelve toda la información de esta de la tabla imdb_movies.
- **getDirectores**. Pasando la id de una película, obtenemos los directores y la información respectiva de estos, que han participado en la película.
- **getActores**. Pasando la id de una película, obtenemos los actores y la información respectiva a estos, que han participado en la película.
- **getPrecio**. Pasando la id una película, obtenemos la información de los productos (versiones) relacionados con la película.

Con estas funciones podemos visualizar la información con respecto a una película.




En cuanto a las páginas de categorías y el buscador, hicimos las siguientes funciones:

- **buscarPeliculas**. Dada una string, buscamos en la base de datos todas aquellas películas que contengan dicha string en el nombre. Hemos limitado el numero de películas que pueden aparecer a 100.
- **categoria**. Dada una categoría, se devuelven todas las películas que pertenecen a dicha categoría.




Se puede apreciar la falta de imágenes en las películas, esto es debido a que son demasiadas.

d. Funcionalidad del carrito.

Por último, tenemos la parte relacionada con el carrito, para esto tenemos varios casos de uso. Imaginémonos que sin hacer login añadimos varias películas a nuestro carrito. Al intentar comprarlas veremos que esto no es posible debido a que no hemos hecho login. Si hacemos login, las películas se añadirán a nuestro carrito en la base datos. Pueden pasar dos cosas:

- La primera es que no haya un carrito. En este caso se crea una nueva order en el customer, esta order tendrá el status a null.
- La segunda es que ya haya un carrito. En este caso se añaden las películas de la session a la order.

Otra cosa es que siempre que se haga login o register se añaden las películas usando la función **addSessionTocarrito**.

Para añadir películas al carrito tenemos dos funciones:

- **anadirFilm (en database)**: Esta función se encuentra en database.py y se usa cuando hemos hecho login, y añade un orderdetail a la order con estado null, pero si ya hay un orderdetail del producto que vamos a añadir, solo aumentamos la cantidad de ese producto.
- **eliminarFilm (en database)**: Esta función se usa cuando estamos logeados, elimina un producto del carrito, básicamente elimina un orderdetail del order si este orderdetail tiene quantity = 1, en caso contrario simplemente decrementamos la variable quantity.
- **eliminar_carrito (en Routes)**: Esta función añade un id de un producto en una session.
- **anhadir_carrito (en Routes)**: Esta función elimina un id de un producto de una session.

Con estas funciones se añaden o eliminan películas en el carrito.

Para comprar las películas siempre habrá que haber hecho login. Se pueden comprar películas por unidades o comprar el carrito entero, tenemos estas funciones:

- **comprarUnidad**. Esta función compra un producto. Para esto creamos una orderid nueva con el orderdetail correspondiente y ponemos el status a “Paid”.
- **comprarTodo**. Esta función compra todos los productos que pertenecen a el carrito. Simplemente ponemos el status a Paid del carrito.

Esto se hace siempre que haya suficiente saldo. Hemos decidido que el saldo de un usuario sea la columna *income*.