|  |  |  |
| --- | --- | --- |
| Bases de datos | | CASO PRACTICO 1 UD 6 |
|  | | |
|  | | |
|  |  | |
| alumno cesur 24/25  Alejandro Muñoz de la Sierra | PROFESOR  Inmaculada Morales Quesada | |

**PROCEDIMIENTOS ALMACENADOS EN ORACLE SQL DEVELOPER**

# 01

introduccion

Este caso práctico se enfoca en crear procedimientos almacenados, funciones y disparadores para gestionar y automatizar tareas en una base de datos. La idea es, básicamente, tomar un diagrama entidad-relación (DER) y, partiendo de él, construir una base que reúna de forma ordenada la información sobre productos, pedidos y detalles – elementos esenciales para el buen funcionamiento de la empresa.

Durante el desarrollo se usará Oracle SQL Developer, una herramienta realmente potente que permite tanto armar la base de datos como escribir y ejecutar esos procedimientos, implementar disparadores y cursores, y además controlar excepciones. En la mayoría de los casos, esto no solo ayuda a manejar los datos de forma más directa, sino que también abre la puerta a automatizar tareas que se repiten y a cuidar la integridad del sistema.

Empezamos con procesos simples, por ejemplo, generar mensajes o validar parámetros, y luego nos lanzamos a construir mecanismos más complejos. Esto implica, por un lado, verificar la existencia de un producto antes de registrarlo, por otro, ajustar el stock a través de disparadores y, de paso, realizar consultas personalizadas con cursores. También se presta especial atención al manejo de errores –algo que buscamos gestionar sin que se interrumpan las operaciones claves.

Este método paulatino nos ayuda a familiarizarnos con la gestión de bases de datos relacionales y nos oferta las herramientas necesarias para desarrollar sistemas de gestión de datos que, en general, sean tanto eficientes como consistentes. Es, en definitiva, un camino que mezcla estructura y flexibilidad para enfrentarse a mayores desafíos en el manejo de la información.

# 02

implementacion de procedimientos almacenados

**1. Procedimiento 1. ‘Hola mundo’:**

CREATE OR REPLACE PROCEDURE mostrar\_hola\_mundo IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hola mundo');

END;

**Explicación detallada del procedimiento:**

CREATE OR REPLACE PROCEDURE mostrar\_hola\_mundo IS

Con esta sentencia se crea (o, más bien, se actualiza automáticamente si ya existía) un procedimiento que llevamos a llamar mostrar\_hola\_mundo. Usar CREATE OR REPLACE nos evita el tener que eliminarlo manualmente, ya que, generalmente, simplemente se sobreescribe; la palabra IS abre la sección donde definimos lo que hará este bloque de instrucciones.

BEGIN

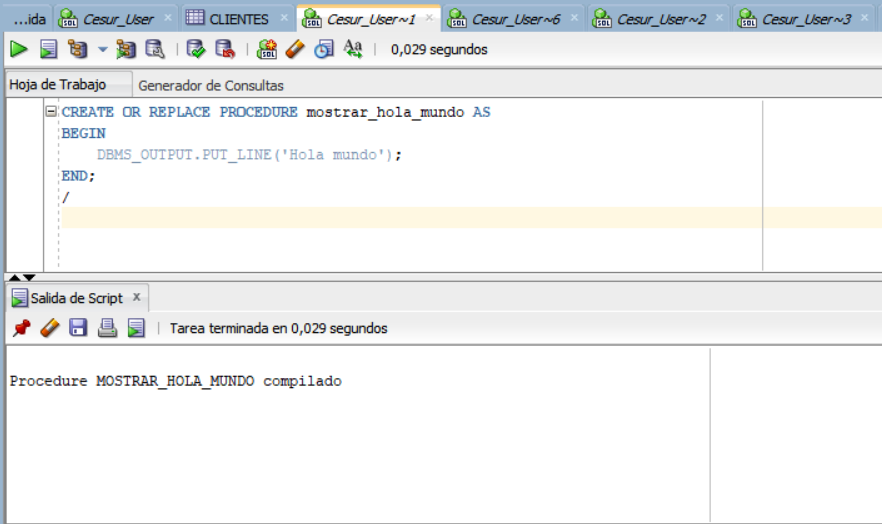
Esta palabra, en la mayoría de los casos, indica el inicio del bloque ejecutable. Es decir, es aquí donde se agrupan las órdenes que se correrán cuando invoquemos el procedimiento.

DBMS\_OUTPUT.PUT\_LINE('Hola mundo');

Aquí se utiliza el paquete interno DBMS\_OUTPUT de Oracle, que nos permite mostrar mensajes en consola. Así, al invocar el método PUT\_LINE, se imprime el texto “Hola mundo”, lo que resulta muy útil para comprobar que el código funcione o para realizar pruebas. Vale la pena notar que este paso repite, de forma simple, la idea de verificar que todo esté en orden.

END;

Con esta breve línea se señala que hemos finalizado la definición del bloque de instrucciones, y en pocas palabras, que ya no se agregan más comandos.



**Ejecución de ejemplo:**

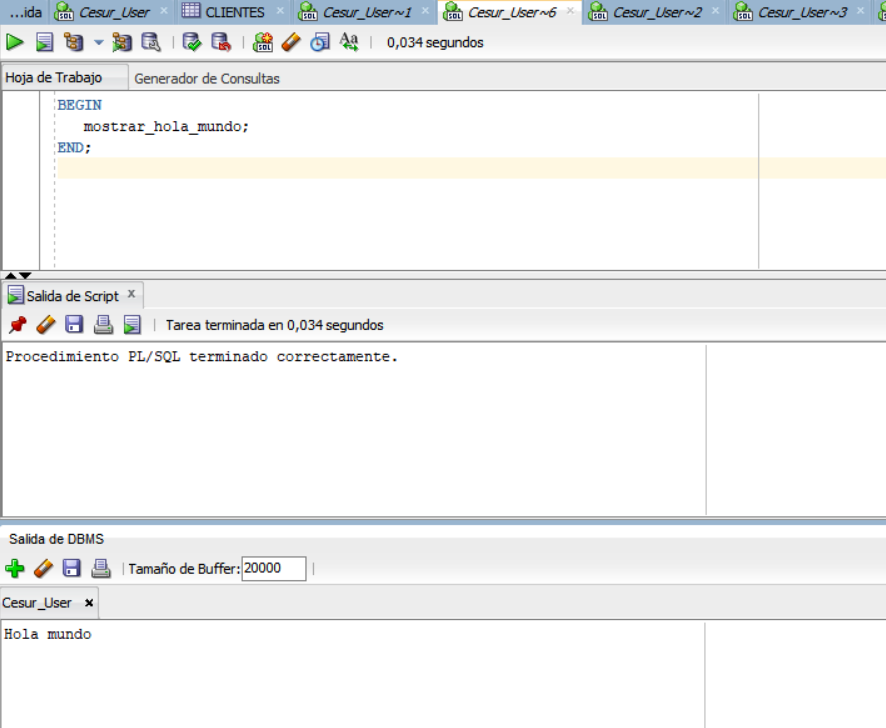
El procedimiento, que no requiere parámetros y se dedica únicamente a mostrar un mensaje, se llama mediante un bloque anónimo. Por ejemplo, lo podemos lanzar de esta manera:

BEGIN

mostrar\_hola\_mundo;

END;

Con esa estructura se activa el procedimiento y, en consecuencia, se verá el mensaje “Hola mundo” en la consola. Esta manera de estructurar el código, a veces de forma un poco desordenada, enfatiza cómo se enlazan los comandos sin necesidad de demasiadas formalidades.



**2. Procedimiento 2 . Indicar si un número es mayor o menor que 10:**

CREATE OR REPLACE PROCEDURE comprobar\_numero(p\_num IN NUMBER) IS

BEGIN

IF p\_num > 10 THEN

DBMS\_OUTPUT.PUT\_LINE('El número ' || p\_num || ' es mayor que 10');

ELSE

DBMS\_OUTPUT.PUT\_LINE('El número ' || p\_num || ' es menor o igual que 10');

END IF;

END;

**Explicación detallada del procedimiento:**

CREATE OR REPLACE PROCEDURE comprobar\_numero(p\_num IN NUMBER) IS

Se inicia con este comando, que crea (o reemplaza, si ya existía) un procedimiento llamado comprobar\_numero.

p\_num es un número, literalmente del tipo NUMBER, usado como parámetro de entrada – o sea, el dato lo suministra quien llama al procedimiento.

El uso de IS abre el bloque donde se van a definir todas las acciones que realizará este procedimiento, lo que viene a ser su “código interno”.

BEGIN

Desde aquí comienza el bloque ejecutable; es el conjunto de instrucciones que efectivamente se ejecutarán al invocar el procedimiento.

IF p\_num > 10 THEN

En esta línea se chequea, de forma muy directa, si el valor de p\_num es mayor a 10 – algo que en la mayoría de los casos decide el camino del flujo del programa.

DBMS\_OUTPUT.PUT\_LINE('El número ' || p\_num || ' es mayor que 10');

Si el número resulta mayor que 10, el sistema usa DBMS\_OUTPUT.PUT\_LINE para mostrar un mensaje en la consola. El mensaje se forma al unir la cadena “El número ”, el valor de p\_num y, finalmente, la frase “ es mayor que 10”.

ELSE

Si no se cumple que p\_num sea mayor que 10, se salta a este otro bloque – la alternativa para los números que no superan ese límite.

DBMS\_OUTPUT.PUT\_LINE('El número ' || p\_num || ' es menor o igual que 10');

En este caso, se imprime un mensaje similar, confirmando que el número es menor o igual a 10.

END IF;

Con esto se cierra la estructura condicional, ya que se han contemplado ambos escenarios posibles.

END;

Y aquí se concluye el procedimiento, terminando el bloque de instrucciones.



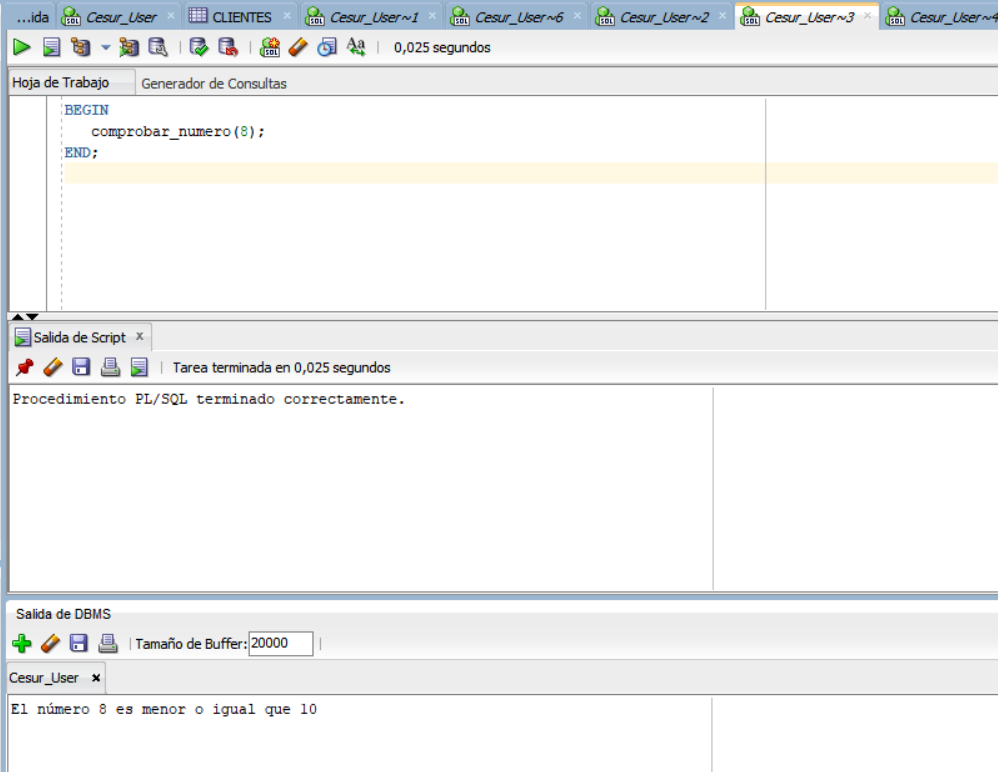
**Ejecución de ejemplo:**

BEGIN

comprobar\_numero(8);

END;

Al ejecutar este bloque se llama al procedimiento con el número 8; de tal modo, al comprobarse que 8 no excede 10, se mostrará en la consola: “El número 8 es menor o igual que 10”.



**3. Procedimiento 3 . Devolver el número multiplicado por 2:**

CREATE OR REPLACE PROCEDURE multiplicar\_por\_dos(p\_num IN NUMBER, p\_resultado OUT NUMBER) IS

BEGIN

p\_resultado := p\_num \* 2;

DBMS\_OUTPUT.PUT\_LINE('El número ' || p\_num || ' multiplicado por 2 es ' || p\_resultado);

END;

**Explicación detallada del procedimiento:**

CREATE OR REPLACE PROCEDURE multiplicar\_por\_dos(p\_num IN NUMBER, p\_resultado OUT NUMBER) IS

Aquí se lanza la idea de un proceso llamado multiplicar\_por\_dos. Se le pasan dos parámetros: el primero, p\_num, es un número que se introduce (es decir, se usa para ingresar datos) – y el segundo, p\_resultado, está pensado para entregar el valor calculado. En la práctica, p\_num se convierte en la entrada y p\_resultado en la salida de la operación.

BEGIN.

Aquí se abre la parte donde todo realmente ocurre; en otras palabras, es la sección que se ejecuta al llamar al procedimiento.

p\_resultado := p\_num \* 2;

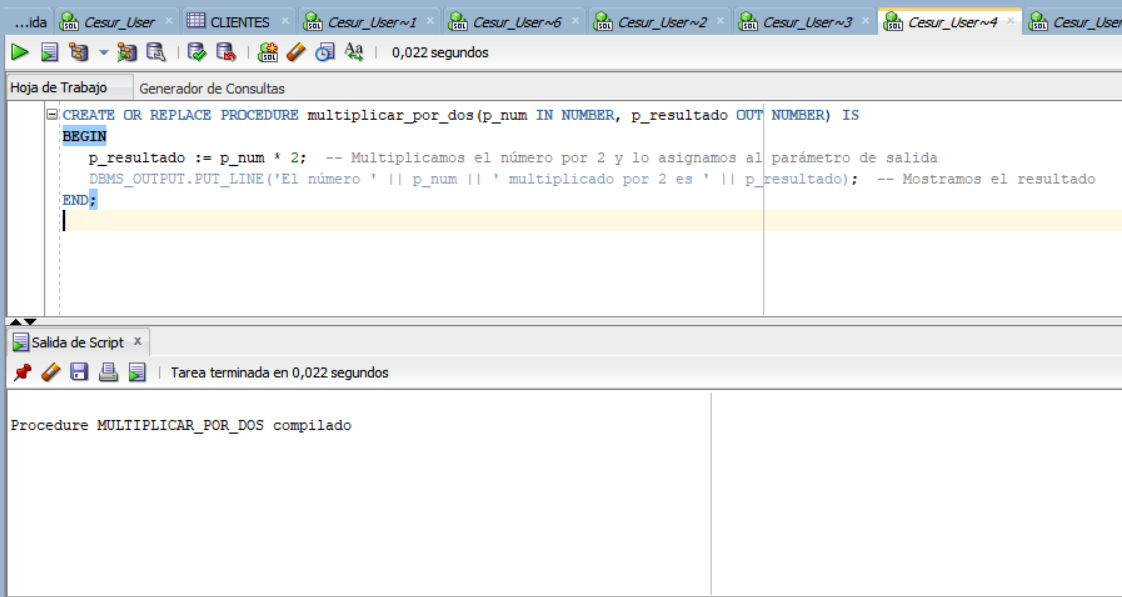
ocurre lo esencial: se toma el valor de p\_num, se multiplica por 2 y se guarda ese resultado en p\_resultado. La operación es directa, aunque puede parecer repetitiva al escucharlo, pero la idea es sencilla.

DBMS\_OUTPUT.PUT\_LINE('El número ' || p\_num || ' multiplicado por 2 es ' || p\_resultado);

se encarga de imprimir un mensaje en la consola. Con esto, se muestra el número original y el resultado producto de la multiplicación, dándote una visión clara de lo que se calculó – algo muy útil para ver de inmediato la respuesta.

END;

se termina el bloque del procedimiento. Es como cerrar un libro después de haber relatado todo el proceso interno.



**Ejecución de ejemplo:**

DECLARE

resultado NUMBER;

BEGIN

multiplicar\_por\_dos(5, resultado);

DBMS\_OUTPUT.PUT\_LINE('Resultado: ' || resultado);

END;

En este bloque se comienza declarando una variable llamada resultado. Luego, se invoca el procedimiento multiplicar\_por\_dos pasando el número 5; al hacerlo, el proceso duplica el 5 (en términos simples, 5 \* 2) y guarda el 10 en la variable resultado. Seguidamente, se envía un mensaje a la consola que dice "Resultado: 10".



**4. Procedimiento 4 . Mostrar los números del 1 al 100 con un WHILE:**

CREATE OR REPLACE PROCEDURE mostrar\_numeros\_1\_a\_100 IS

v\_num NUMBER := 1;

BEGIN

WHILE v\_num <= 100 LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_num);

v\_num := v\_num + 1;

END LOOP;

END;

**Explicación detallada del procedimiento:**

CREATE OR REPLACE PROCEDURE mostrar\_numeros\_1\_a\_100 IS

Empezamos definiendo la rutina mostrar\_numeros\_1\_a\_100. Usar CREATE OR REPLACE significa que, si ya existe una rutina con ese nombre, se sobreescribe sin darnos muchas vueltas. La palabra IS, casi como un aviso, nos dice que ahora se irán declarando variables o se escribirá el cuerpo principal.

v\_num NUMBER := 1;

Aquí se crea la variable v\_num con el valor inicial 1, la cual funcionará como contador para recorrer desde el número 1 hasta el 100. Es un dato bastante simple, pero fundamental para la tarea.

BEGIN

Con BEGIN arranca el bloque donde se encuentran todas las órdenes a ejecutar cuando se llama a la rutina. Es el inicio del “acción” propiamente dicha.

WHILE v\_num <= 100 LOOP

Ahora se introduce un bucle WHILE, que se repetirá mientras v\_num sea menor o igual a 100. En la práctica, esto quiere decir que, en la mayoría de los casos, se ejecutará 100 veces: cada vuelta es una oportunidad de mostrar un número.

DBMS\_OUTPUT.PUT\_LINE(v\_num);

Dentro del bucle, se invoca DBMS\_OUTPUT.PUT\_LINE(v\_num) para imprimir en la consola el valor actual del contador. Esta función de Oracle es muy útil, tanto para depurar como para mostrar datos, y aquí cumple su rol sin complicaciones.

v\_num := v\_num + 1;

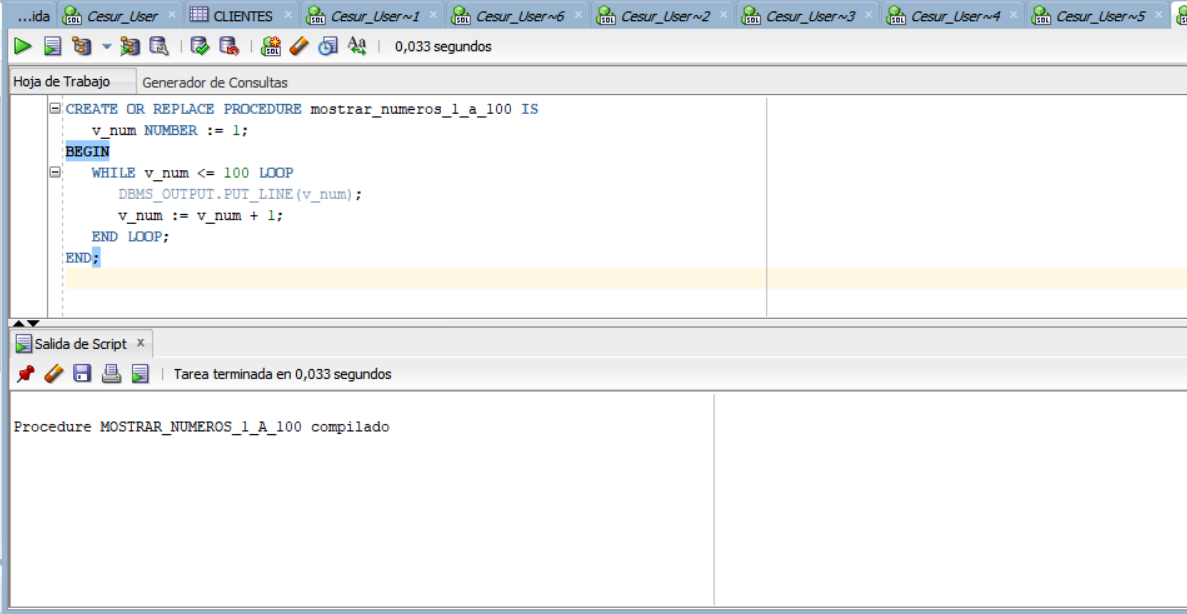
Luego se incrementa v\_num en 1; esto es clave para pasar al siguiente número en la siguiente iteración, permitiendo que el ciclo avance de manera natural (o, para decirlo de otro modo, nunca se quede estancado).

END LOOP;

Con END LOOP se cierra el ciclo. Una vez que v\_num se convierte en 101, la condición ya no se cumple y el bucle se termina sin más, concluyendo su función.

END;

Finalmente, END cierra todo el procedimiento, sellando de forma definitiva la rutina.



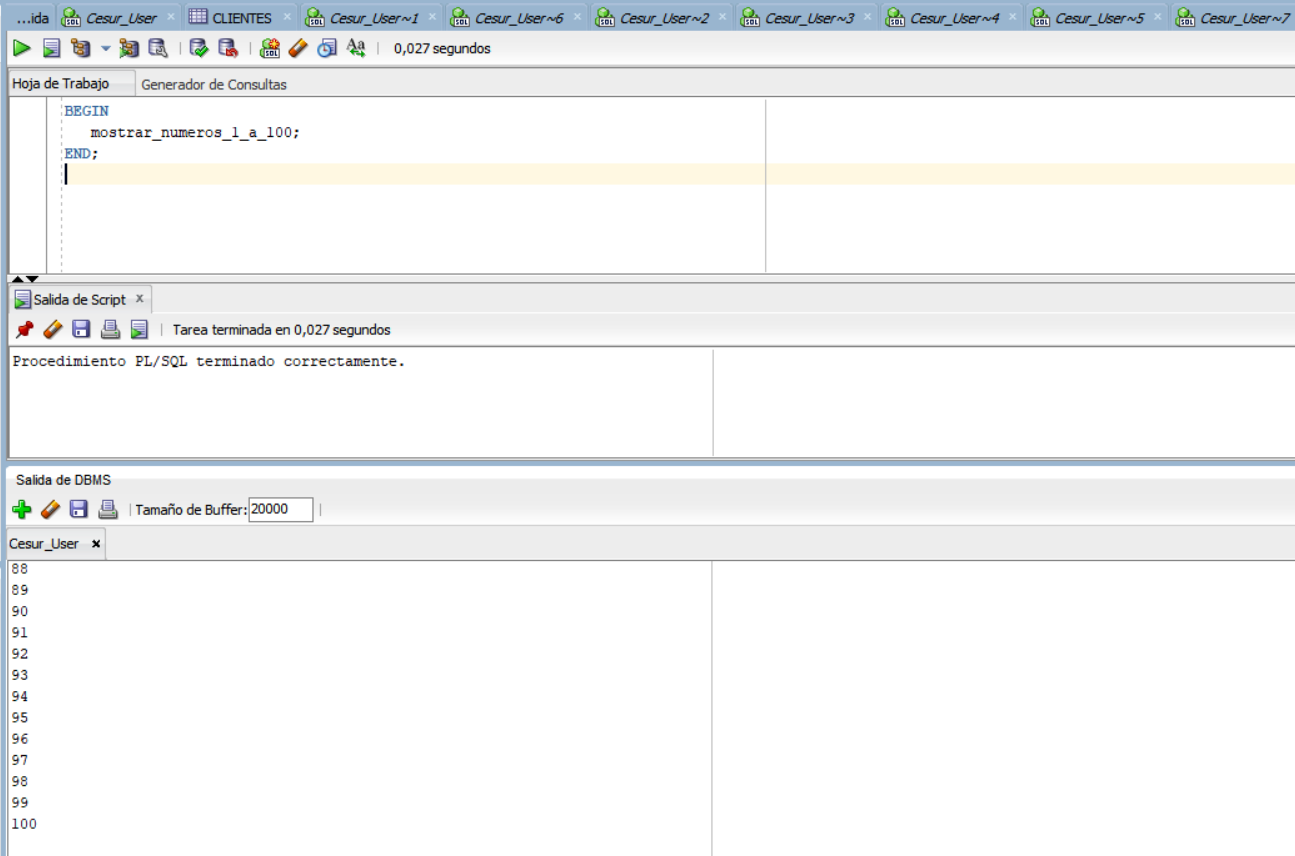
**Ejecución de ejemplo:**

BEGIN

mostrar\_numeros\_1\_a\_100;

END;

Al ejecutar este bloque, el procedimiento se pone en marcha y, en la consola, se irán visualizando uno a uno los números del 1 al 100. Todo fluye de forma continua, con cada número apareciendo en su momento, lo que demuestra la sencillez –y a la vez la efectividad– de esta estructura.



**5.Procedimiento 5 . Mostrar los números pares del 1 al 100 con un FOR:**

CREATE OR REPLACE PROCEDURE mostrar\_pares\_1\_a\_100 IS

BEGIN

FOR i IN 1..100 LOOP

IF MOD(i, 2) = 0 THEN

DBMS\_OUTPUT.PUT\_LINE(i);

END IF;

END LOOP;

END;

**Explicación detallada del procedimiento:**

CREATE OR REPLACE PROCEDURE mostrar\_pares\_1\_a\_100 IS.

Esto significa que, si ya existiera alguna versión anterior, se reemplaza sin mayores complicaciones. La palabra IS se utiliza para marcar que lo que sigue es el bloque en el que se especifica qué va a hacer el procedimiento.

BEGIN

momento en el que se activan todas las órdenes que se ejecutarán al llamar al procedimiento. Es aquí donde comienza la acción.

FOR i IN 1..100 LOOP

que recorre uno a uno los números del 1 al 100; cada vez, i toma un valor distinto al avanzar por la secuencia. En cada iteración se verifica una condición:

IF MOD(i, 2) = 0 THEN.

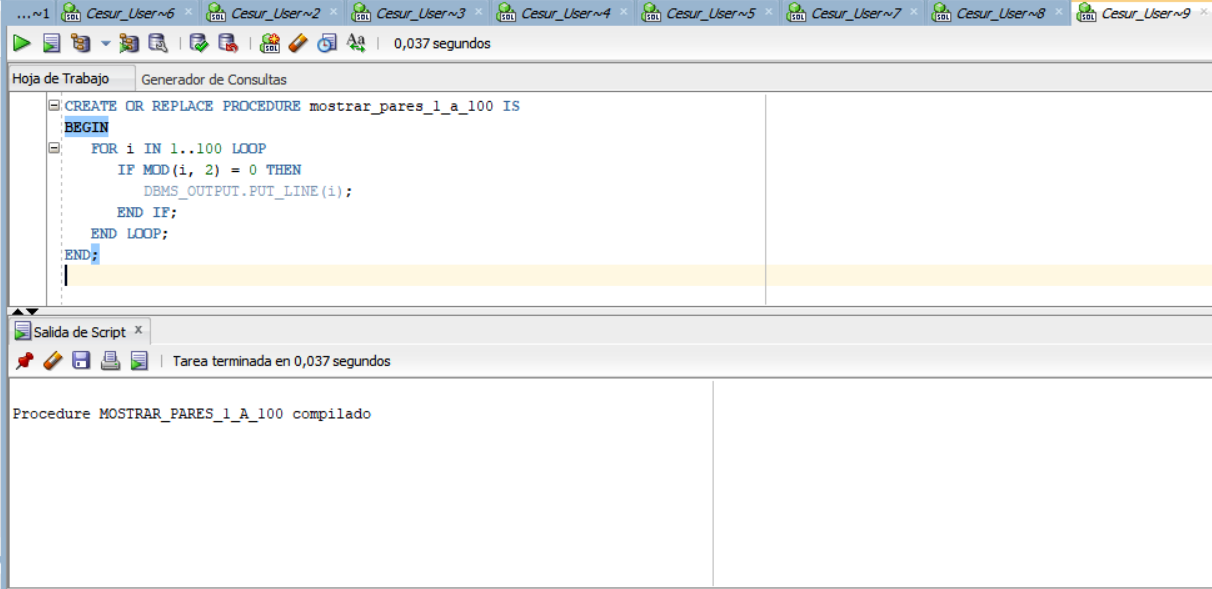
Dicho de otro modo, se comprueba si al dividir el número i entre 2 se obtiene un resto igual a cero, lo que nos indica que estamos frente a un número par.

DBMS\_OUTPUT.PUT\_LINE(i)

para imprimir el valor en pantalla; en caso de que i no sea par, simplemente se omite y el bucle continúa con el siguiente valor. Después de esta comprobación se cierra la condición con END IF; (¡vale decir, se descarta sin acción si no encaja con lo buscado!).

END LOOP;

en este punto ya se han revisado todos los números entre 1 y 100. Finalmente, se concluye el procedimiento con END;, cerrando la definición de toda la lógica programada.



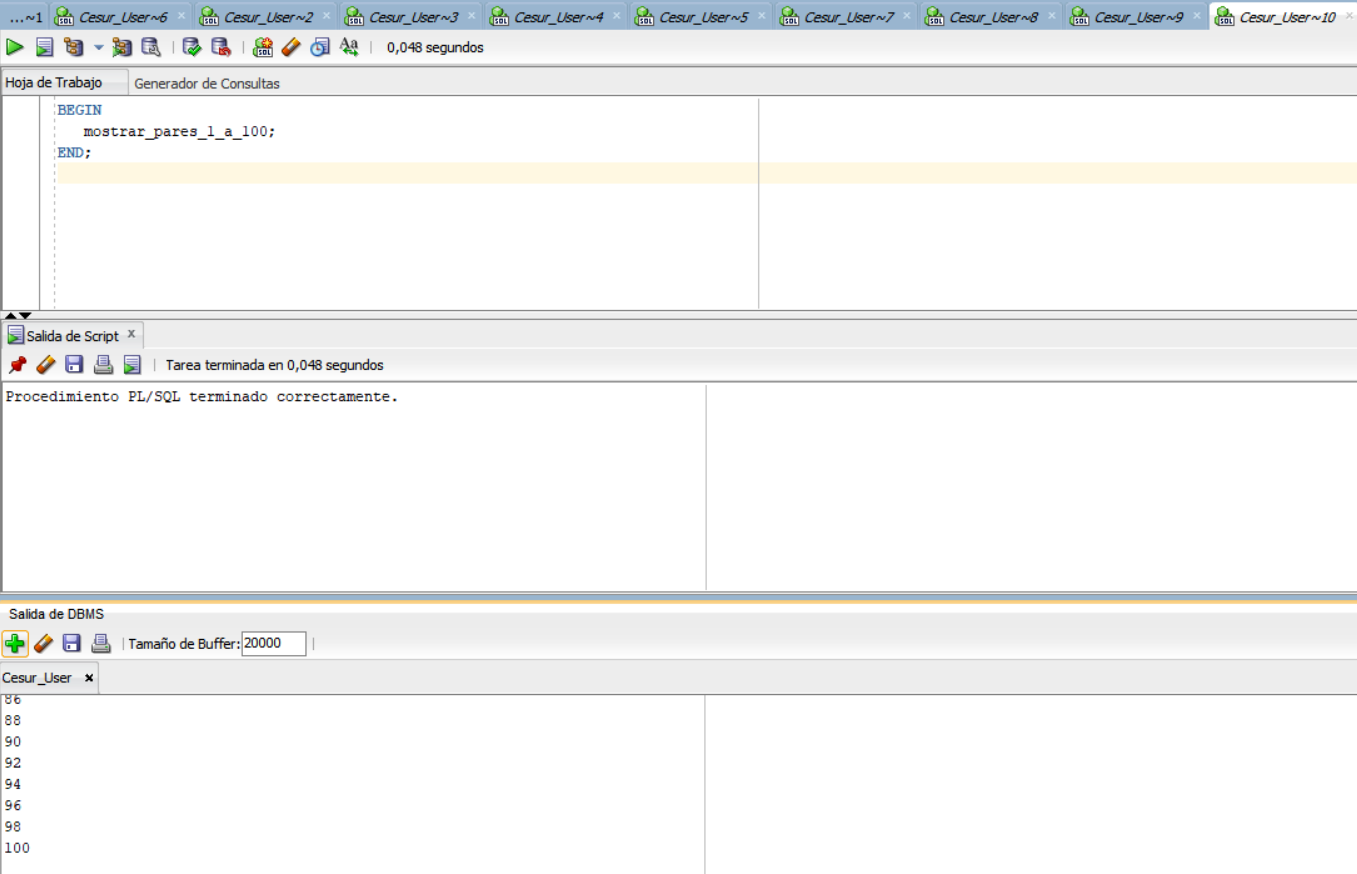
**Ejecución de ejemplo:**

BEGIN

mostrar\_pares\_1\_a\_100;

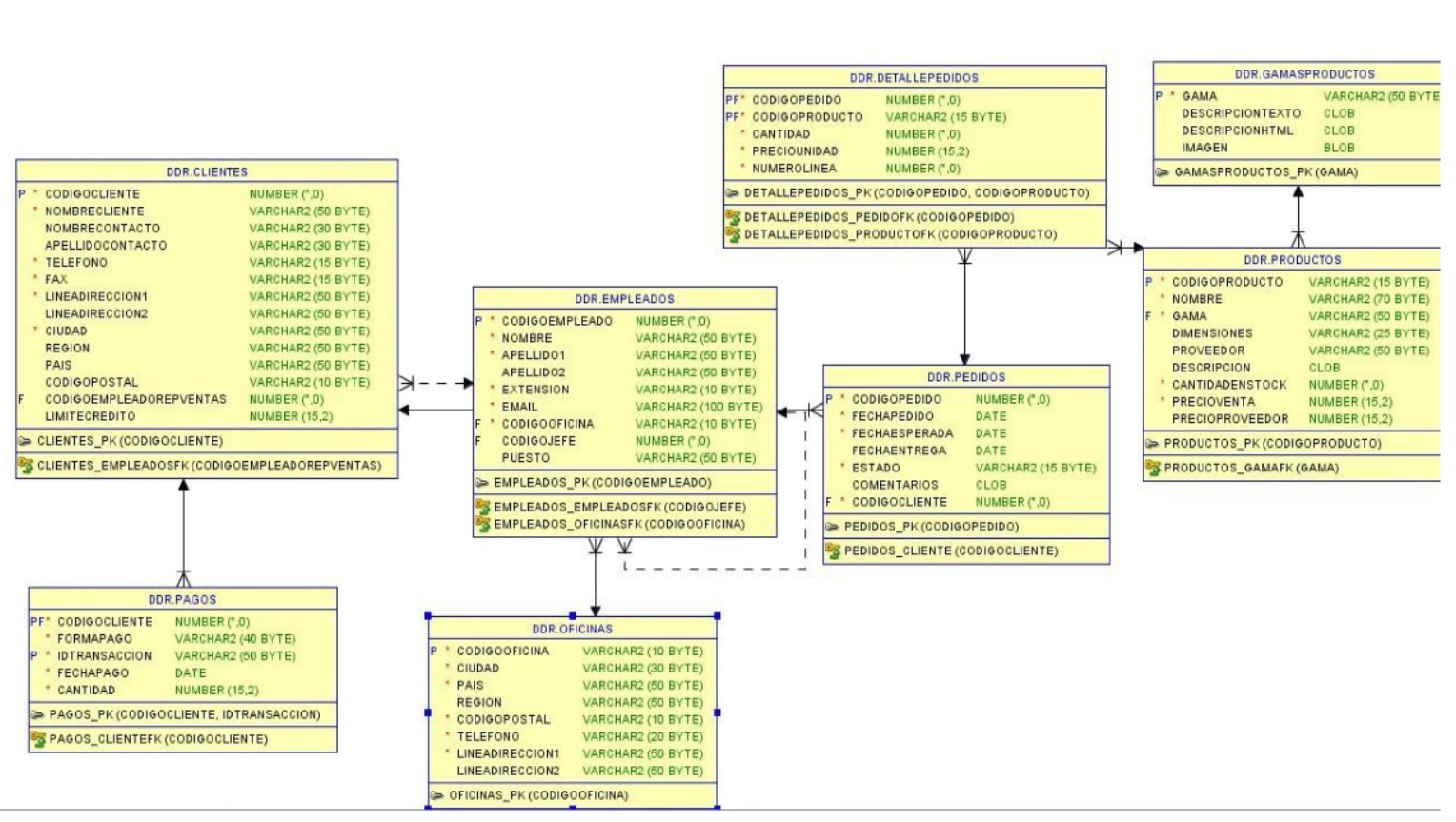
END;

Con esta ejecución, y de forma bastante directa, se mostrará en la consola una secuencia de números que, en definitiva, corresponden a todos los pares existentes entre 1 y 100.



# 02

creacion de la base de datos apartir del diagrama uml



**Interpretación y desarrollo inicial de la base de datos**

**1. Análisis del diagrama UML y la relación de entidades**

Antes de aventurarnos en la construcción de la base, nos sentamos a revisar con calma el diagrama UML que nos entregaron. Ese dibujo, que más o menos era una representación del modelo de datos de una empresa, abarcaba elementos tan variados como oficinas, empleados, clientes, pedidos, pagos, productos y sus gamas.

Al ir repasándolo, nos quedó claro cuáles eran las piezas claves del sistema:

• OFICINAS

• EMPLEADOS

• GAMASPRODUCTOS

• PRODUCTOS

• CLIENTES

• PEDIDOS

• PAGOS

• DETALLEPEDIDOS

Se podía notar, de forma bastante natural, cómo se vinculaban unas cosas con otras. Por ejemplo, cada empleado figura asignado a una oficina; además, en la tabla EMPLEADOS se aprecia una autorelación que permite que un empleado figure como jefe de otro. También se observa que cada cliente tiene asignado, de forma casi directa, un representante de ventas (que, claro, es también un empleado). En el caso de los productos, cada uno se atiene a una gama específica, mientras que los pedidos y pagos se relacionan íntimamente con los clientes. Por último, cada detalle de pedido une un producto a un pedido concreto.

**2. Creación de la base de datos paso a paso**

Con el modelo ya desmenuzado, empezamos a montar la base de datos siguiendo un orden algo flexible. Iniciamos por aquellas tablas que no dependían de otras, como OFICINAS o GAMASPRODUCTOS, y de a poco fuimos incorporando las claves primarias y foráneas para mantener la integridad entre tablas.

En situaciones algo más intrincadas –como la autorrelación en EMPLEADOS– decidimos dejar la clave foránea (CODIGOJEFE) para una segunda fase, una vez que la estructura central estuviera bien asentada. Se optó por usar tipos de datos específicos: VARCHAR2 para textos, NUMBER para números, CLOB para textos extensos y BLOB para imágenes. Al final, hicimos una revisión general para cerciorarnos de que no quedasen errores de integridad y que cada conexión estuviese definida de manera correcta.

-- Tabla OFICINAS

CREATE TABLE OFICINAS (

CODIGOOFICINA VARCHAR2(10 BYTE) NOT NULL,

CIUDAD VARCHAR2(30 BYTE),

PAIS VARCHAR2(50 BYTE),

REGION VARCHAR2(50 BYTE),

CODIGOPOSTAL VARCHAR2(10 BYTE),

TELEFONO VARCHAR2(20 BYTE),

LINEADIRECCION1 VARCHAR2(50 BYTE),

LINEADIRECCION2 VARCHAR2(50 BYTE),

CONSTRAINT OFICINAS\_PK PRIMARY KEY (CODIGOOFICINA)

);

-- Tabla EMPLEADOS

-- Nota: Se crea la tabla primero y luego se añade la FK a sí misma (CODIGOJEFE)

-- para evitar problemas de dependencia circular durante la creación.

CREATE TABLE EMPLEADOS (

CODIGOEMPLEADO NUMBER NOT NULL,

NOMBRE VARCHAR2(50 BYTE),

APELLIDO1 VARCHAR2(50 BYTE),

APELLIDO2 VARCHAR2(50 BYTE),

EXTENSION VARCHAR2(10 BYTE),

EMAIL VARCHAR2(100 BYTE),

CODIGOOFICINA VARCHAR2(10 BYTE),

CODIGOJEFE NUMBER,

PUESTO VARCHAR2(50 BYTE),

CONSTRAINT EMPLEADOS\_PK PRIMARY KEY (CODIGOEMPLEADO),

CONSTRAINT EMPLEADOS\_OFICINASFK FOREIGN KEY (CODIGOOFICINA) REFERENCES OFICINAS(CODIGOOFICINA)

-- La FK para CODIGOJEFE se añade después

);

-- Añadir la FK de auto-referencia para EMPLEADOS.CODIGOJEFE

ALTER TABLE EMPLEADOS ADD CONSTRAINT EMPLEADOS\_EMPLEADOSFK FOREIGN KEY (CODIGOJEFE) REFERENCES EMPLEADOS(CODIGOEMPLEADO);

-- Tabla GAMASPRODUCTOS

CREATE TABLE GAMASPRODUCTOS (

GAMA VARCHAR2(50 BYTE) NOT NULL,

DESCRIPCIONTEXTO CLOB,

DESCRIPCIONHTML CLOB,

IMAGEN BLOB,

CONSTRAINT GAMASPRODUCTOS\_PK PRIMARY KEY (GAMA)

);

-- Tabla PRODUCTOS

CREATE TABLE PRODUCTOS (

CODIGOPRODUCTO VARCHAR2(15 BYTE) NOT NULL,

NOMBRE VARCHAR2(70 BYTE),

GAMA VARCHAR2(50 BYTE),

DIMENSIONES VARCHAR2(25 BYTE), -- Ajustado según diagrama (era 26 BYTE)

PROVEEDOR VARCHAR2(50 BYTE),

DESCRIPCION CLOB,

CANTIDADENSTOCK NUMBER,

PRECIOVENTA NUMBER(15, 2),

PRECIOPROVEEDOR NUMBER(15, 2),

CONSTRAINT PRODUCTOS\_PK PRIMARY KEY (CODIGOPRODUCTO),

CONSTRAINT PRODUCTOS\_GAMAFK FOREIGN KEY (GAMA) REFERENCES GAMASPRODUCTOS(GAMA)

);

-- Tabla CLIENTES

CREATE TABLE CLIENTES (

CODIGOCLIENTE NUMBER NOT NULL,

NOMBRECLIENTE VARCHAR2(50 BYTE),

NOMBRECONTACTO VARCHAR2(30 BYTE),

APELLIDOCONTACTO VARCHAR2(30 BYTE),

TELEFONO VARCHAR2(15 BYTE),

FAX VARCHAR2(15 BYTE),

LINEADIRECCION1 VARCHAR2(50 BYTE),

LINEADIRECCION2 VARCHAR2(50 BYTE),

CIUDAD VARCHAR2(50 BYTE),

REGION VARCHAR2(50 BYTE),

PAIS VARCHAR2(50 BYTE),

CODIGOPOSTAL VARCHAR2(10 BYTE),

CODIGOEMPLEADOREPVENTAS NUMBER,

LIMITECREDITO NUMBER(15, 2),

CONSTRAINT CLIENTES\_PK PRIMARY KEY (CODIGOCLIENTE),

CONSTRAINT CLIENTES\_EMPLEADOSFK FOREIGN KEY (CODIGOEMPLEADOREPVENTAS) REFERENCES EMPLEADOS(CODIGOEMPLEADO)

);

-- Tabla PEDIDOS

CREATE TABLE PEDIDOS (

CODIGOPEDIDO NUMBER NOT NULL,

FECHAPEDIDO DATE,

FECHAESPERADA DATE,

FECHAENTREGA DATE,

ESTADO VARCHAR2(15 BYTE),

COMENTARIOS CLOB,

CODIGOCLIENTE NUMBER,

CONSTRAINT PEDIDOS\_PK PRIMARY KEY (CODIGOPEDIDO),

CONSTRAINT PEDIDOS\_CLIENTE FOREIGN KEY (CODIGOCLIENTE) REFERENCES CLIENTES(CODIGOCLIENTE)

);

-- Tabla PAGOS

CREATE TABLE PAGOS (

CODIGOCLIENTE NUMBER NOT NULL,

FORMAPAGO VARCHAR2(40 BYTE),

IDTRANSACCION VARCHAR2(50 BYTE) NOT NULL,

FECHAPAGO DATE,

CANTIDAD NUMBER(15, 2),

CONSTRAINT PAGOS\_PK PRIMARY KEY (CODIGOCLIENTE, IDTRANSACCION),

CONSTRAINT PAGOS\_CLIENTEFK FOREIGN KEY (CODIGOCLIENTE) REFERENCES CLIENTES(CODIGOCLIENTE)

);

-- Tabla DETALLEPEDIDOS

CREATE TABLE DETALLEPEDIDOS (

CODIGOPEDIDO NUMBER NOT NULL,

CODIGOPRODUCTO VARCHAR2(15 BYTE) NOT NULL,

CANTIDAD NUMBER,

PRECIOUNIDAD NUMBER(15, 2),

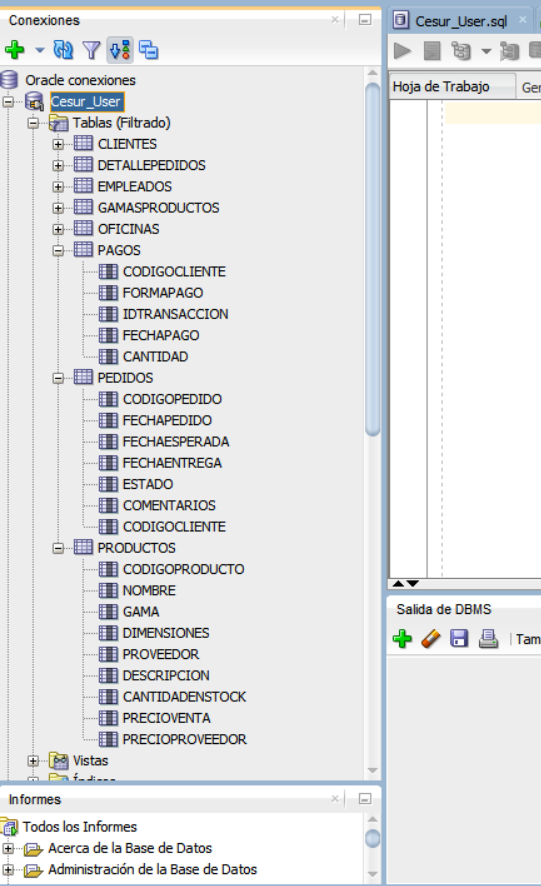
NUMEROLINEA NUMBER,

CONSTRAINT DETALLEPEDIDOS\_PK PRIMARY KEY (CODIGOPEDIDO, CODIGOPRODUCTO),

CONSTRAINT DETALLEPEDIDOS\_PEDIDOFK FOREIGN KEY (CODIGOPEDIDO) REFERENCES PEDIDOS(CODIGOPEDIDO),

CONSTRAINT DETALLEPEDIDOS\_PRODUCTOFK FOREIGN KEY (CODIGOPRODUCTO) REFERENCES PRODUCTOS(CODIGOPRODUCTO)

);



**3. Poblado de la base de datos**

Para probar a fondo los procedimientos que, resulta imprescindible poblar la base con datos de prueba. Esto nos permite acercarnos al comportamiento real del sistema.

Insertamos registros en las tablas de la siguiente forma:

– OFICINAS: para garantizar que existiesen códigos de oficina válidos.

– EMPLEADOS: necesarios, por un lado, para las referencias con clientes y, por otro, para verificar la jerarquía interna.

– CLIENTES: pues de ellos se originan los pedidos y pagos.

– GAMASPRODUCTOS: de modo que cada producto se vincule a una gama definida.

– PRODUCTOS: indispensables para la creación de pedidos.

– PEDIDOS: para posteriormente asociar los detalles correspondientes.

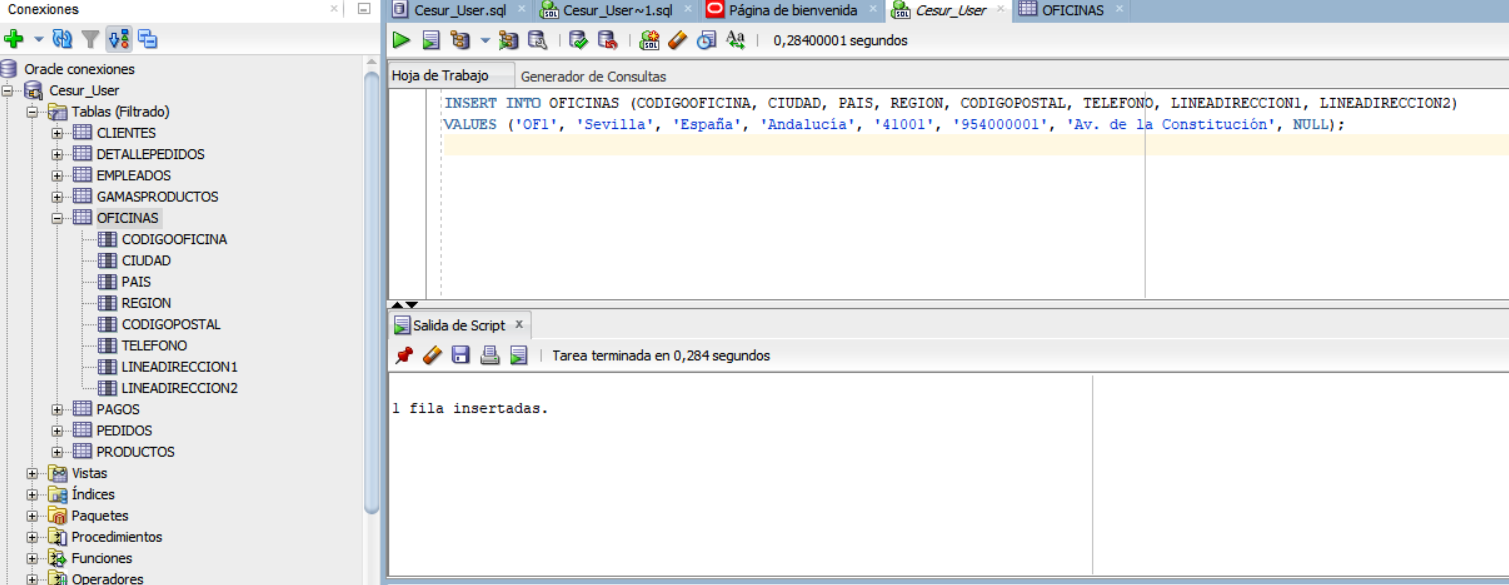
– DETALLEPEDIDOS: cruciales para confirmar el correcto funcionamiento del trigger (Caso 6).

**Bloque 1: Inserción de OFICINAS**

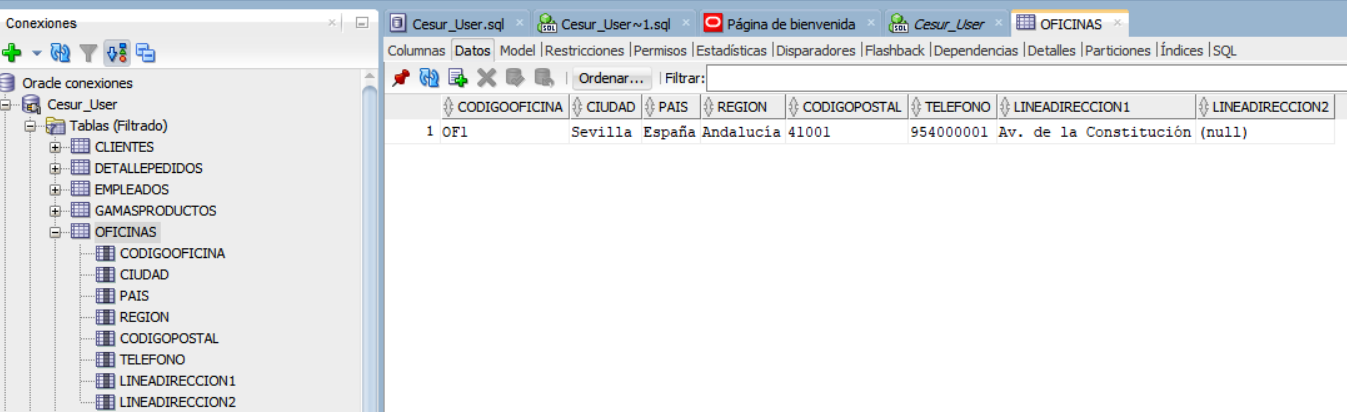
INSERT INTO OFICINAS (CODIGOOFICINA, CIUDAD, PAIS, REGION, CODIGOPOSTAL, TELEFONO, LINEADIRECCION1, LINEADIRECCION2)

VALUES ('OF1', 'Sevilla', 'España', 'Andalucía', '41001', '954000001', 'Av. de la Constitución', NULL);

Ejecutamos el script:



Comprobamos que los datos quedan insertados correctamente:



Procedemos de igual forma para el resto de los datos necesarios de la base

**Bloque 2: Inserción de EMPLEADOS**

-- Insertamos un jefe

INSERT INTO EMPLEADOS (CODIGOEMPLEADO, NOMBRE, APELLIDO1, APELLIDO2, EXTENSION, EMAIL, CODIGOOFICINA, CODIGOJEFE, PUESTO)

VALUES (1, 'Ana', 'García', 'López', 'x101', 'ana.garcia@empresa.com', 'OF1', NULL, 'Director General');

-- Insertamos un empleado que depende del jefe

INSERT INTO EMPLEADOS (CODIGOEMPLEADO, NOMBRE, APELLIDO1, APELLIDO2, EXTENSION, EMAIL, CODIGOOFICINA, CODIGOJEFE, PUESTO)

VALUES (2, 'Luis', 'Pérez', 'Martínez', 'x102', 'luis.perez@empresa.com', 'OF1', 1, 'Representante Ventas');

**Bloque 3: Inserción de GAMASPRODUCTOS**

INSERT INTO GAMASPRODUCTOS (GAMA, DESCRIPCIONTEXTO, DESCRIPCIONHTML, IMAGEN)

VALUES ('Clásicos', 'Colección de productos clásicos', '<p>Colección de productos clásicos</p>', NULL);

**Bloque 4: Inserción de PRODUCTOS**

INSERT INTO PRODUCTOS (CODIGOPRODUCTO, NOMBRE, GAMA, DIMENSIONES, PROVEEDOR, DESCRIPCION, CANTIDADENSTOCK, PRECIOVENTA, PRECIOPROVEEDOR)

VALUES ('P001', 'Coche Antiguo', 'Clásicos', '20x10x8', 'Proveedor1', 'Modelo de coche antiguo', 100, 45.99, 30.00);

INSERT INTO PRODUCTOS (CODIGOPRODUCTO, NOMBRE, GAMA, DIMENSIONES, PROVEEDOR, DESCRIPCION, CANTIDADENSTOCK, PRECIOVENTA, PRECIOPROVEEDOR)

VALUES ('P002', 'Moto Vintage', 'Clásicos', '15x5x8', 'Proveedor2', 'Moto de colección vintage', 80, 55.99, 35.00);

**Bloque 5: Inserción de CLIENTES**

INSERT INTO CLIENTES (CODIGOCLIENTE, NOMBRECLIENTE, NOMBRECONTACTO, APELLIDOCONTACTO, TELEFONO, FAX, LINEADIRECCION1, LINEADIRECCION2, CIUDAD, REGION, PAIS, CODIGOPOSTAL, CODIGOEMPLEADOREPVENTAS, LIMITECREDITO)

VALUES (1001, 'Juguetes Reyes', 'Carlos', 'Domínguez', '911223344', '911223345', 'Calle Real, 1', NULL, 'Madrid', 'Madrid', 'España', '28001', 2, 1000.00);

**Bloque 6: Inserción de PEDIDOS**

INSERT INTO PEDIDOS (CODIGOPEDIDO, FECHAPEDIDO, FECHAESPERADA, FECHAENTREGA, ESTADO, COMENTARIOS, CODIGOCLIENTE)

VALUES (5001, TO\_DATE('2025-04-01','YYYY-MM-DD'), TO\_DATE('2025-04-05','YYYY-MM-DD'), TO\_DATE('2025-04-03','YYYY-MM-DD'), 'Entregado', 'Entrega puntual', 1001);

**Bloque 7: Inserción de DETALLEPEDIDOS**

INSERT INTO DETALLEPEDIDOS (CODIGOPEDIDO, CODIGOPRODUCTO, CANTIDAD, PRECIOUNIDAD, NUMEROLINEA)

VALUES (5001, 'P001', 2, 45.99, 1);

INSERT INTO DETALLEPEDIDOS (CODIGOPEDIDO, CODIGOPRODUCTO, CANTIDAD, PRECIOUNIDAD, NUMEROLINEA)

VALUES (5001, 'P002', 1, 55.99, 2);

**Bloque 8: Inserción de PAGOS**

INSERT INTO PAGOS (CODIGOCLIENTE, FORMAPAGO, IDTRANSACCION, FECHAPAGO, CANTIDAD)

VALUES (1001, 'Transferencia', 'TX001', TO\_DATE('2025-04-02','YYYY-MM-DD'), 147.97);

# 03

procedimiento almacenado trigger (punto6) aplicado a la base creada y datos insertados

Ya con la base de datos montada y habiendo cargado los registros de prueba, nos lanzamos a desarrollar ese trigger que se pedía en el enunciado. La misión es sencilla: cada vez que se añade o se quita un detalle de pedido, queremos que el stock del producto se ajuste automáticamente, sin que nadie tenga que hacerlo a mano.

**Paso 1 – Crear tabla AUDITORIA\_STOCK**

CREATE TABLE AUDITORIA\_STOCK (

ID\_AUDITORIA NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,

CODIGOPRODUCTO VARCHAR2(15),

FECHA\_MODIFICACION DATE DEFAULT SYSDATE,

STOCK\_ANTERIOR NUMBER,

STOCK\_NUEVO NUMBER

);

Empezamos creando una tabla auxiliar, a la que llamamos AUDITORIA\_STOCK, y que se ocupará de registrar cada vez que el stock de un producto cambie. Cada ajuste se anotará en un registro distinto, lo que nos permitirá llevar un historial de cómo se han modificado los niveles de inventario. Entre la información que se guardará destacan:

ID\_AUDITORIA: Es el identificador único de cada entrada; se genera automáticamente con una secuencia implícita (IDENTITY).

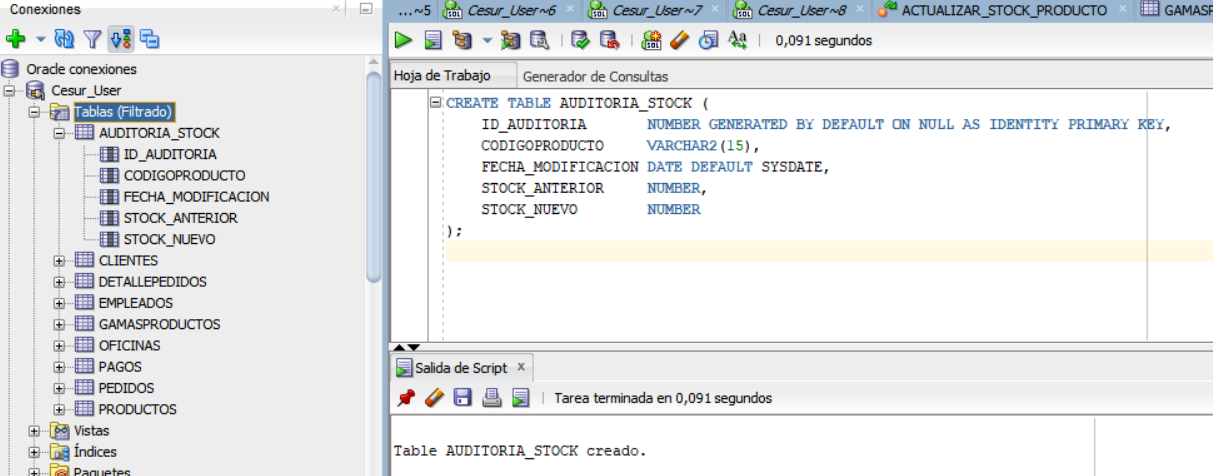
CODIGOPRODUCTO: Aquí se almacena el identificador del producto que ha sido modificado.

FECHA\_MODIFICACION: Se registra, de manera precisa, la fecha y la hora en que ocurrió el cambio.

STOCK\_ANTERIOR: Guarda el valor que tenía el stock antes de la actualización.

STOCK\_NUEVO: Captura el nuevo valor del stock tras el cambio.

Esta tabla resulta fundamental para mantener un control histórico de las variaciones en el inventario y, en la mayoría de los casos, servirá de base para el trigger que implementamos a continuación.



**Paso 2 – Crear trigger AUDITORIA\_STOCK**

CREATE OR REPLACE TRIGGER TR\_AUDITORIA\_STOCK

BEFORE UPDATE OF CANTIDADENSTOCK ON PRODUCTOS

FOR EACH ROW

WHEN (OLD.CANTIDADENSTOCK != NEW.CANTIDADENSTOCK)

BEGIN

INSERT INTO AUDITORIA\_STOCK (

CODIGOPRODUCTO,

STOCK\_ANTERIOR,

STOCK\_NUEVO

) VALUES (

:OLD.CODIGOPRODUCTO,

:OLD.CANTIDADENSTOCK,

:NEW.CANTIDADENSTOCK

);

END;

Este mecanismo guarda cada modificación en el inventario de productos. La idea es sencilla: llevar un recuento de los cambios para tener un control y, en general, una trazabilidad de las actualizaciones. Ahora vamos desmenuzando el código del trigger de forma menos formal:

CREATE OR REPLACE TRIGGER TR\_AUDITORIA\_STOCK

Se lanza con este comando que, básicamente, crea el trigger; si ya estaba creado, se reemplaza sin hacer líos manuales. Es como decir “si ya está, que lo sobreescriba” y así evitamos pasos extra.

BEFORE UPDATE OF CANTIDADENSTOCK ON PRODUCTOS

Aquí se especifica que el activador saltará justo antes de modificar el campo CANTIDADENSTOCK en la tabla PRODUCTOS. O sea, antes de que el stock cambie, nos permite capturar el valor anterior; esto resulta muy útil.

FOR EACH ROW

Con esto indicamos que el trigger se ejecuta para cada fila afectada en la actualización, no solo una vez por la sentencia completa. Esto es clave si queremos saber, sin dejar lugar a dudas, qué producto y cómo ha sido modificado.

WHEN (OLD.CANTIDADENSTOCK != NEW.CANTIDADENSTOCK)

La función del condicional es simple: solo actuamos si el stock realmente varió. Se compara el valor viejo (OLD.CANTIDADENSTOCK) con el nuevo (NEW.CANTIDADENSTOCK) y si no hay diferencia, el trigger se queda callado, evitando registros innecesarios.

BEGIN … END;

Dentro de este bloque se encierra todo el código que se debe ejecutar cuando la condición se cumple. Ahí se definen, en pocas palabras, las acciones a realizar al dispararse el trigger.

INSERT INTO AUDITORIA\_STOCK (...) VALUES (...);

En este paso se inserta un registro en la tabla AUDITORIA\_STOCK. Así se va archivando el historial del inventario, guardando datos del producto, el stock antes de la acción y su nuevo valor, cosa que suele ser esencial para revisiones más tarde.

:OLD.CODIGOPRODUCTO

A pesar de que no se modifica, se accede al valor anterior del campo CODIGOPRODUCTO para identificar de qué producto se trata. Es muy útil tener este dato, aun cuando no se actualice directamente.

:OLD.CANTIDADENSTOCK

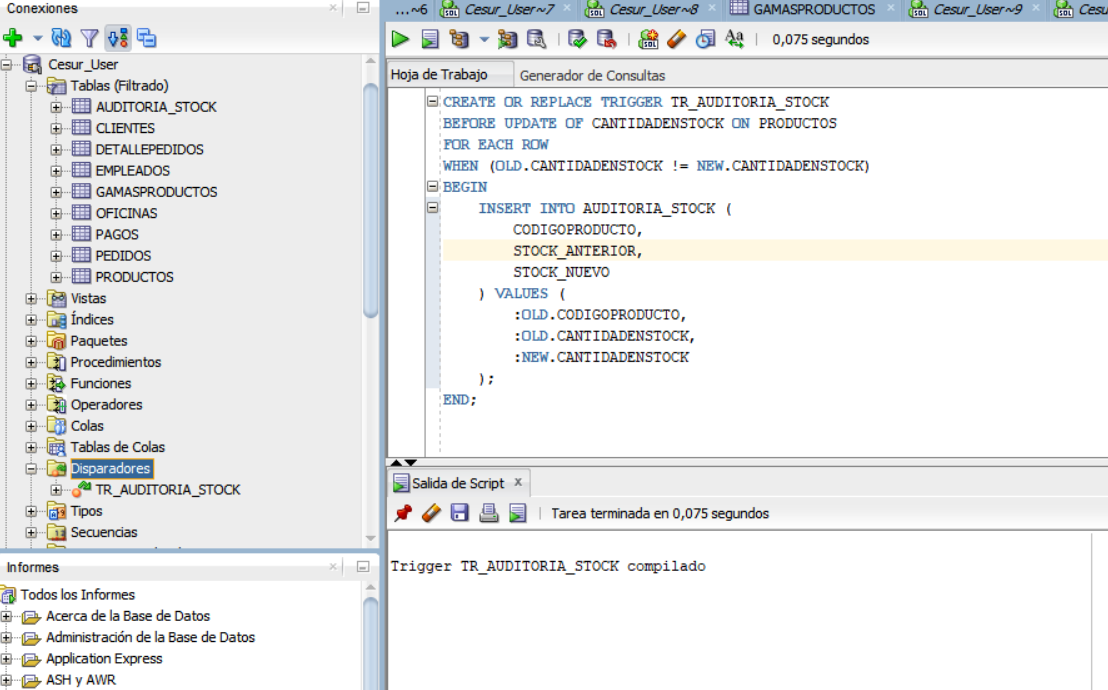
Este marcador nos muestra cuál era el stock previo a la actualización, obtenido mediante la referencia a :OLD; es como echar una mirada atrás justo antes de que cambie todo.

:NEW.CANTIDADENSTOCK

Finalmente, aquí se recoge el valor que se asignará como nuevo stock, usando la referencia :NEW para que quede claro qué se está actualizando.

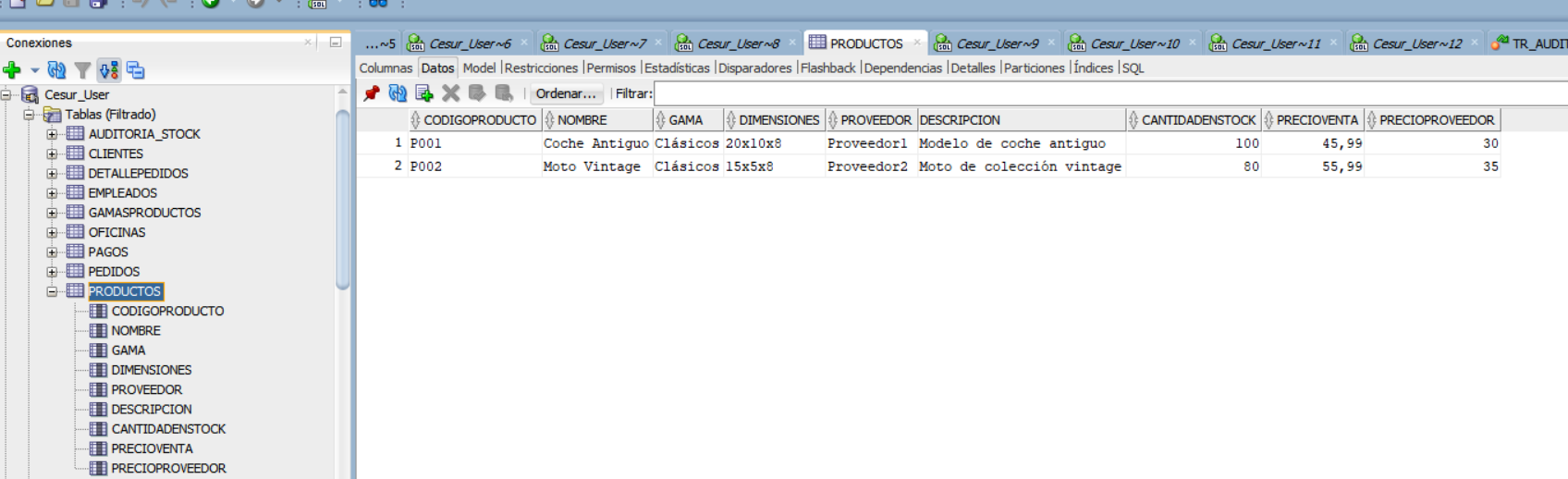
En resumen

El trigger es como un vigilante silencioso: detecta cualquier cambio en el inventario, registrando tanto el código del producto como el valor antiguo y el nuevo del stock en una tabla de auditoría. De esta manera, se facilita un seguimiento detallado del historial de movimientos del inventario, ideal para análisis posteriores o verificaciones puntuales.



**Paso 3 – -modificar PRODUCTOS y comprobar AUDITORIA\_STOCK**

Datos originales en PRODUCTOS:

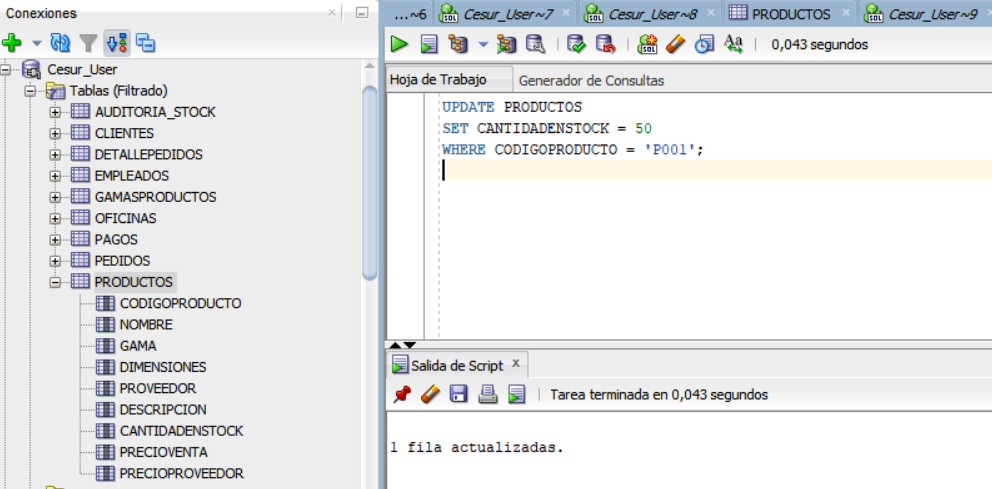


**Ejecutamos UPDATE** para modificar el stock de un producto. En este caso, vamos a usar el código P001.

UPDATE PRODUCTOS

SET CANTIDADENSTOCK = 50

WHERE CODIGOPRODUCTO = 'P001';

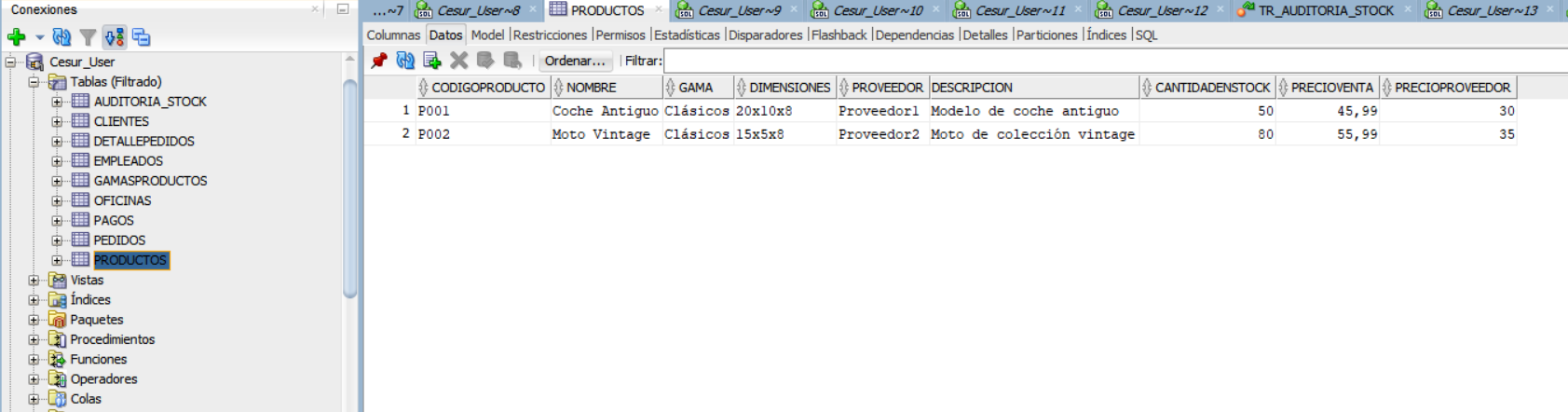


**Verificar la tabla de AUDITORIA\_STOCK y PRODUCTOS otra vez** para asegurarnos de que se ha insertado el registro correctamente.

SELECT \* FROM AUDITORIA\_STOCK

WHERE CODIGOPRODUCTO = 'P001';





# 04

procedimiento almacenado cursor (punto7) aplicado a la base creada y datos insertados

**Procedimiento 7: Presentar los productos y sus cantidades de un pedido**

El objetivo aquí es simple, pero a la vez bastante útil: queremos acceder a los datos de un pedido en concreto recorriendo la tabla DETALLEPEDIDOS. Para ello, se pasa el código del pedido (p\_codigo\_pedido) como entrada y se consigue mostrar tanto el identificador de cada producto como la cantidad asociada. La técnica que vamos a usar se basa en un cursor explícito, que se encarga de unir (sí, literalmente) la información de DETALLEPEDIDOS con la de PRODUCTOS mediante el campo CODIGOPRODUCTO.

Primero, creamos un procedimiento en el que definimos como único parámetro de entrada el código del pedido. Dentro de esta rutina, se declara un cursor que recoge los productos y sus cantidades vinculados al pedido indicado. Seguidamente, se abre el cursor pasando el valor de p\_codigo\_pedido y, a partir de ahí, se recorre la información en un bucle LOOP. En cada vuelta del bucle se recuperan y muestran dichos datos usando, por ejemplo, DBMS\_OUTPUT.PUT\_LINE. Tras procesar todos los registros, se cierra el cursor para liberar los recursos, algo muy importante aunque a veces se pase por alto.

CREATE OR REPLACE PROCEDURE VER\_PRODUCTOS\_PEDIDO (p\_codigo\_pedido IN NUMBER) AS

CURSOR c\_productos IS

SELECT dp.CODIGOPRODUCTO, p.NOMBRE, dp.CANTIDAD

FROM DETALLEPEDIDOS dp

JOIN PRODUCTOS p ON dp.CODIGOPRODUCTO = p.CODIGOPRODUCTO

WHERE dp.CODIGOPEDIDO = p\_codigo\_pedido;

v\_codigo\_producto VARCHAR2(15);

v\_nombre\_producto VARCHAR2(70);

v\_cantidad NUMBER;

BEGIN

OPEN c\_productos;

LOOP

FETCH c\_productos INTO v\_codigo\_producto, v\_nombre\_producto, v\_cantidad;

EXIT WHEN c\_productos%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Producto: ' || v\_nombre\_producto || ' (Código: ' || v\_codigo\_producto || ')');

DBMS\_OUTPUT.PUT\_LINE('Cantidad: ' || v\_cantidad);

END LOOP;

CLOSE c\_productos;

END;

**Detallando el funcionamiento del código:**

Arrancamos con la definición del procedimiento VER\_PRODUCTOS\_PEDIDO: se crea (o se reemplaza, si ya estaba en el sistema) algo que se encarga de mostrar productos de un pedido específico. Básicamente, se usa CREATE OR REPLACE PROCEDURE junto con un parámetro, p\_codigo\_pedido, de tipo NUMBER que nos sirve para identificar el pedido en cuestión. Es decir, si ya existía otra versión con ese mismo nombre, se actualiza de forma inmediata, sin mayores complicaciones.

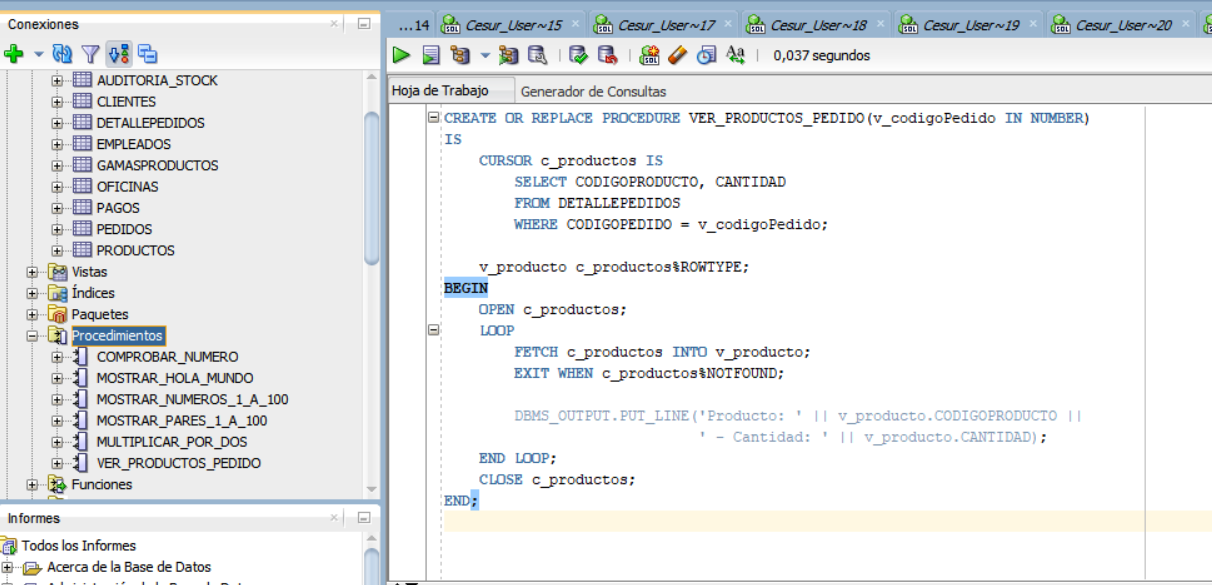
Luego viene el tema del cursor, denominado c\_productos. Este cursor es una herramienta de PL/SQL que, en la mayoría de los casos, sirve para ir recorriendo de a una fila cada vez el resultado de una consulta. La consulta asociada extrae tres campos: dp.CODIGOPRODUCTO (que da el código del producto que figura en el pedido), p.NOMBRE (para mostrar el nombre real del producto, ya que ese dato se saca de la tabla PRODUCTOS) y dp.CANTIDAD (la cantidad pedida). La fuente de datos es la tabla DETALLEPEDIDOS (referida como dp, para hacer la vida más fácil) y se junta con la tabla PRODUCTOS (alias p) mediante la condición de que sus códigos coincidan. Además, se filtra al final con WHERE dp.CODIGOPEDIDO = p\_codigo\_pedido para asegurarse de que solo se traigan los productos del pedido que interesa.

Más adelante, se declaran algunas variables locales: v\_codigo\_producto (un VARCHAR2 que acepta hasta 15 caracteres) guardará el código del producto; v\_nombre\_producto (con espacio para hasta 70 caracteres) almacenará el nombre; y v\_cantidad, una variable numérica, tomará el valor de la cantidad pedida. Estos elementos son esenciales para ir almacenando la información que se extrae.

A partir de aquí , se abre el cursor c\_productos, lo que en esencia prepara el conjunto de resultados, y se entra en un bucle. Dentro del bucle se llama a FETCH para trasladar los valores de la fila actual a nuestras variables; de manera natural, si al intentar FETCH no se encuentran más filas, la condición c\_productos%NOTFOUND hace que se interrumpa el ciclo. En cada vuelta, el procedimiento despliega un par de líneas: la primera combina el nombre del producto y su código (usando DBMS\_OUTPUT.PUT\_LINE para unir 'Producto: ', el nombre, y el código entre paréntesis), y la segunda muestra la cantidad asociada.

No se olvida, por supuesto, que una vez que se han procesado todas las filas, el cursor se cierra para liberar los recursos, culminando así el bloque ejecutable y, por ende, el procedimiento. En resumidas cuentas, este código está diseñado para relacionar, fila a fila, los detalles de un pedido con la información de los productos correspondientes

Comprobamos que el procedimiento compila y se almacena correctamente:



**Comprobación:**

El procedimiento comienza recibiendo un dato clave: el código del pedido (v\_codigoPedido). Con ese dato se lanza un cursor que va pasando, uno a uno, por cada producto relacionado al pedido; es decir, se hace un recorrido por todos ellos, sin saltarse ninguno.

Después, y de forma bastante directa, el sistema imprime en pantalla dos detalles esenciales: primero aparece el nombre del producto y, a continuación, se muestra la cantidad de unidades correspondientes a ese ítem en el pedido.

Primero activamos la salida de datos por consola para poder ver el resultado:

SET SERVEROUTPUT ON;

Y ahora ejecutamos el procedimiento almacenado y comprobamos:

EXEC VER\_PRODUCTOS\_PEDIDO('5001');

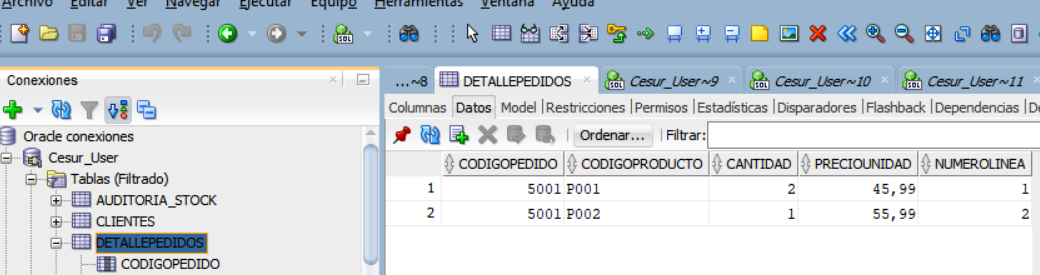
O también:

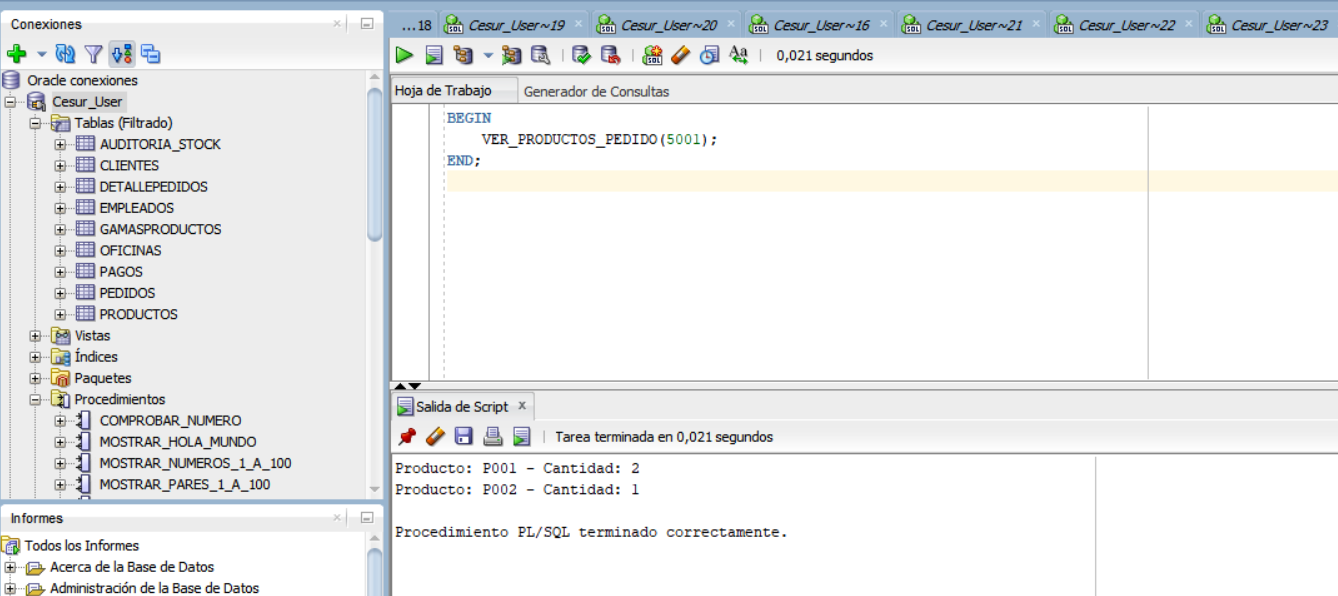
BEGIN

VER\_PRODUCTOS\_PEDIDO(5001);

END;

Podemos utilizar cualquier código de pedido que tengamos en la base de datos. En este caso 5001.





# 05

procedimiento almacenado control de excepciones (punto8) aplicado a la base creada y datos insertados

Debemos controlar si el producto a insertar ya existe , lanzando un mensaje, o no, insertándolo correctamente. El código que necesitamos es el siguiente:

-- Procedimiento almacenado con control de excepciones

CREATE OR REPLACE PROCEDURE ALTA\_PRODUCTO(

p\_codigo\_producto IN VARCHAR2,

p\_nombre IN VARCHAR2,

p\_gama IN VARCHAR2,

p\_cantidad\_stock IN NUMBER,

p\_precio\_venta IN NUMBER

)

IS

BEGIN

-- Intentamos insertar el producto directamente

INSERT INTO PRODUCTOS (

CODIGOPRODUCTO, NOMBRE, GAMA, DIMENSIONES, PROVEEDOR, DESCRIPCION, CANTIDADENSTOCK, PRECIOVENTA, PRECIOPROVEEDOR

)

VALUES (

p\_codigo\_producto, p\_nombre, p\_gama, NULL, NULL, NULL, p\_cantidad\_stock, p\_precio\_venta, NULL

);

-- Si la inserción tiene éxito, mostramos el mensaje

DBMS\_OUTPUT.PUT\_LINE('Producto con código ' || p\_codigo\_producto || ' dado de alta con éxito.');

-- Sección de manejo de excepciones

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Error: el producto con código ' || p\_codigo\_producto || ' ya está dado de alta.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Se ha producido un error inesperado: ' || SQLERRM);

END ALTA\_PRODUCTO;

/

**Detallando el funcionamiento del código:**

CREATE OR REPLACE PROCEDURE ALTA\_PRODUCTO:

Esta línea arranca el proceso para configurar o actualizar un procedimiento almacenado. O sea, si ya había uno con ese nombre en la base de datos, se sobrescribe con el código nuevo, lo que en la mayoría de los casos evita conflictos.

PROCEDURE ALTA\_PRODUCTO:

Este es el nombre que le damos al procedimiento que registra un producto. Básicamente, se usará para dar de alta un nuevo artículo insertando sus detalles en la tabla PRODUCTOS; un método simple para mantener los registros al día.

(p\_codigo\_producto IN VARCHAR2, p\_nombre IN VARCHAR2, p\_gama IN VARCHAR2, p\_cantidad\_stock IN NUMBER, p\_precio\_venta IN NUMBER):

Aquí se enlistan los parámetros de entrada, cada uno con su tipo explícito. Por ejemplo, p\_codigo\_producto es una cadena que identifica el producto; p\_nombre cumple una función similar para el nombre; p\_gama, otra cadena, clasifica o agrupa el producto; mientras que p\_cantidad\_stock, de tipo NUMBER, lleva la cuenta del stock disponible y p\_precio\_venta nos indica el precio asignado. Se soporta algo de repetición para enfatizar que cada parámetro juega un rol esencial en el proceso.

IS:

Esta palabra marca el inicio de la parte lógica. Todo lo que sigue, hasta el bloque BEGIN, se prepara para definir qué se hará realmente cuando se invoque el procedimiento; es como trazar los planos antes de la acción.

BEGIN:

Con BEGIN se abre el bloque de ejecución, y todo lo que esté entre aquí y END se ejecutará al llamar al procedimiento. Es el momento en el que se efectúa la tarea principal: insertar los datos.

INSERT INTO PRODUCTOS (...) VALUES (...):

En este tramo se ordena agregar una fila nueva a la tabla PRODUCTOS. Los campos que aparecen –como CODIGOPRODUCTO, NOMBRE, GAMA, etc.– se llenan con los valores suministrados mediante los parámetros. Para aquellas columnas que no reciben un valor explícito (por ejemplo, DIMENSIONES o PROVEEDOR) se asigna NULL por defecto, asegurando que el registro quede completo a su manera.

DBMS\_OUTPUT.PUT\_LINE('Producto con código ' || p\_codigo\_producto || ' dado de alta con éxito.'):

Aprovechamos esta función para desplegar un mensaje en la consola de Oracle, confirmando que se ha registrado el producto correctamente y mostrando el código correspondiente; es un aviso práctico para el usuario o administrador, aunque a veces se puede ver con leves desviaciones en la puntuación.

EXCEPTION:

Si algo no sale como se esperaba en el bloque de ejecución, entramos en el bloque EXCEPTION. Es la zona de rescate, donde se abordan los errores que puedan aparecer durante la inserción.

WHEN DUP\_VAL\_ON\_INDEX THEN:

Esta parte se activa si se intenta insertar un valor ya existente en una columna con restricción de clave primaria o índice único. En palabras simples, si el código del producto ya está registrado, Oracle lanza automáticamente DUP\_VAL\_ON\_INDEX.

THEN:

Aquí se define qué hacer cuando se detecta ese error en particular. Usamos DBMS\_OUTPUT.PUT\_LINE para notificar que el producto con ese mismo código ya se encuentra dado de alta, evitando insertar duplicados.

WHEN OTHERS THEN:

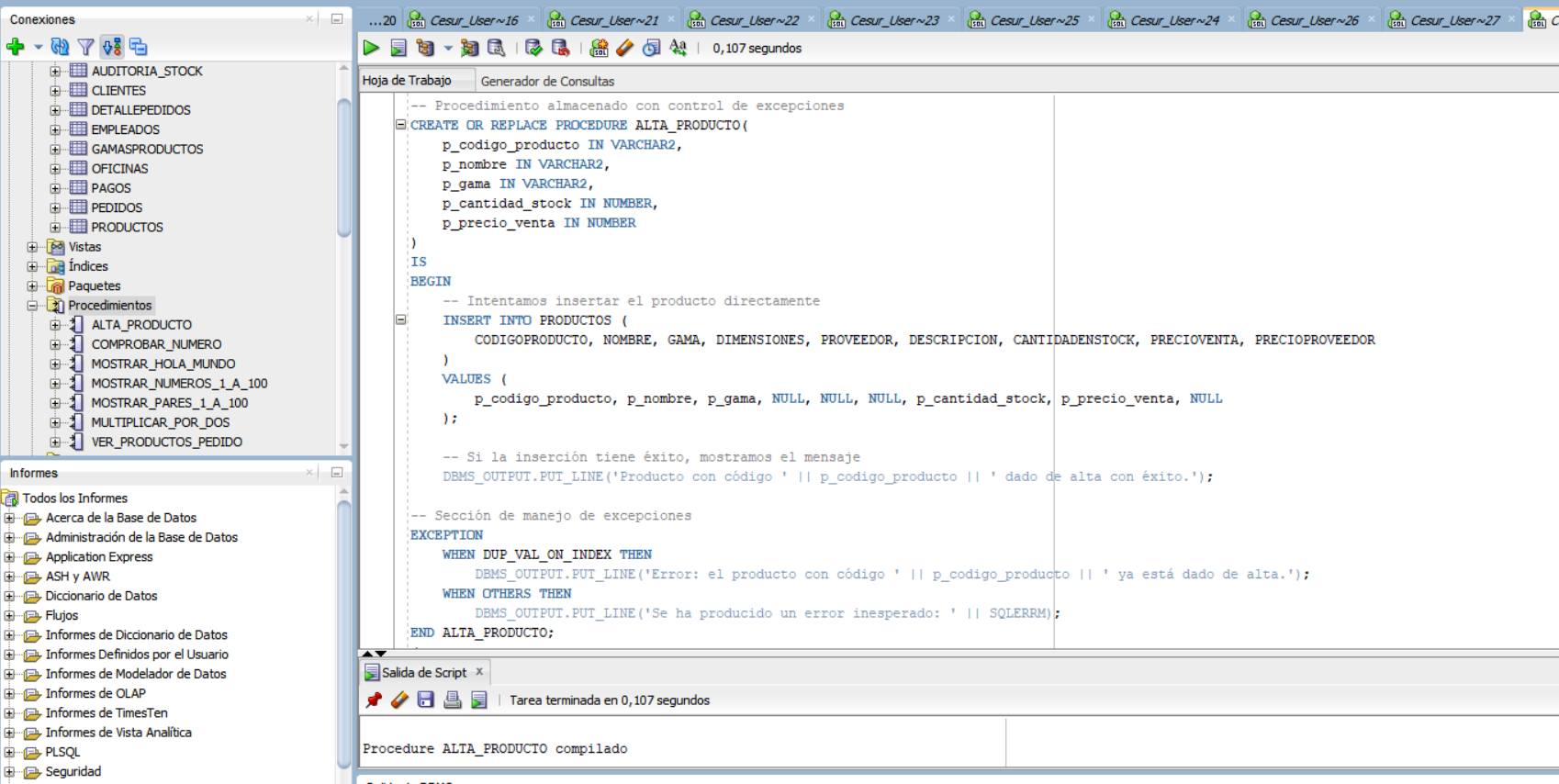
Por último, este bloque es para capturar cualquier otro error imprevisto. Funciona como una red de seguridad ante situaciones inesperadas, sin dejar de responder por completo la ejecución fallida. En las situaciones ya cubiertas por excepciones previas, el código tiene un mecanismo para tratar lo inesperado. Por ejemplo, la instrucción

DBMS\_OUTPUT.PUT\_LINE('Se ha producido un error inesperado: ' || SQLERRM):

sirve para que, si ocurre un error que nadie esperaba, se muestre un aviso. Primero se despliega el mensaje “Se ha producido un error inesperado” y, a continuación, se concatena la información que Oracle ofrece mediante SQLERRM. Esto, en la mayoria de los casos, ayuda a orientar el diagnóstico y la depuración del fallo.

END ALTA\_PRODUCTO;:

La cual marca el fin del procedimiento almacenado ALTA\_PRODUCTO. Con esa instrucción, el proceso se da por terminado de ejecutar.



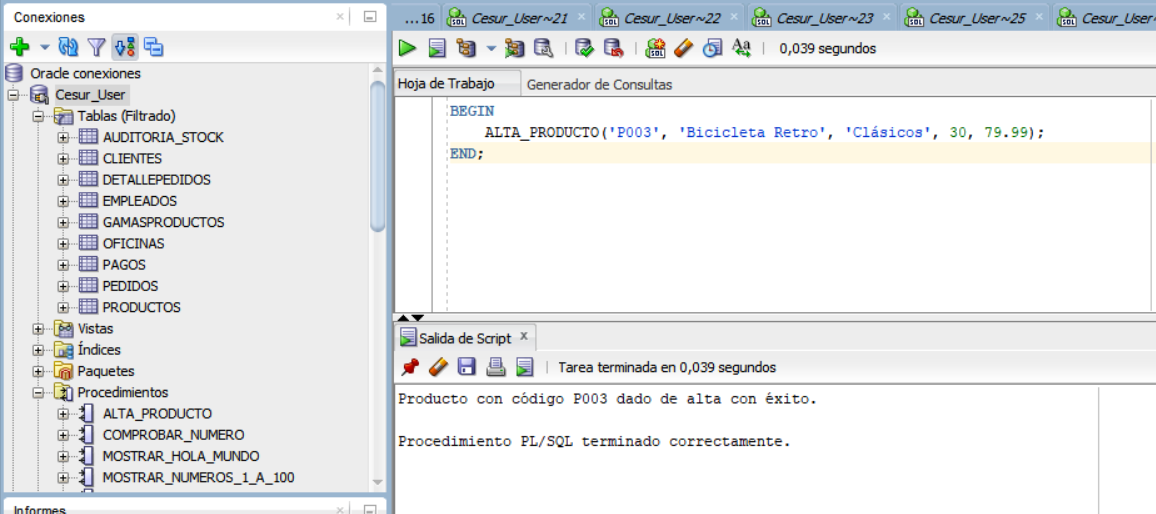
El procedimiento ALTA\_PRODUCTO se encarga de registrar un producto nuevo en la tabla PRODUCTOS. Básicamente, inserta los valores que se pasan como parámetros de entrada. Si se detecta que el código del producto se repite —lo que indica que el producto ya existe— se capta la excepción y se muestra un mensaje al usuario. Aparte, el bloque WHEN OTHERS se ocupa de cualquier otro error no previsto; en otras palabras, se gestiona tanto la duplicidad como cualquier fallo inesperado.

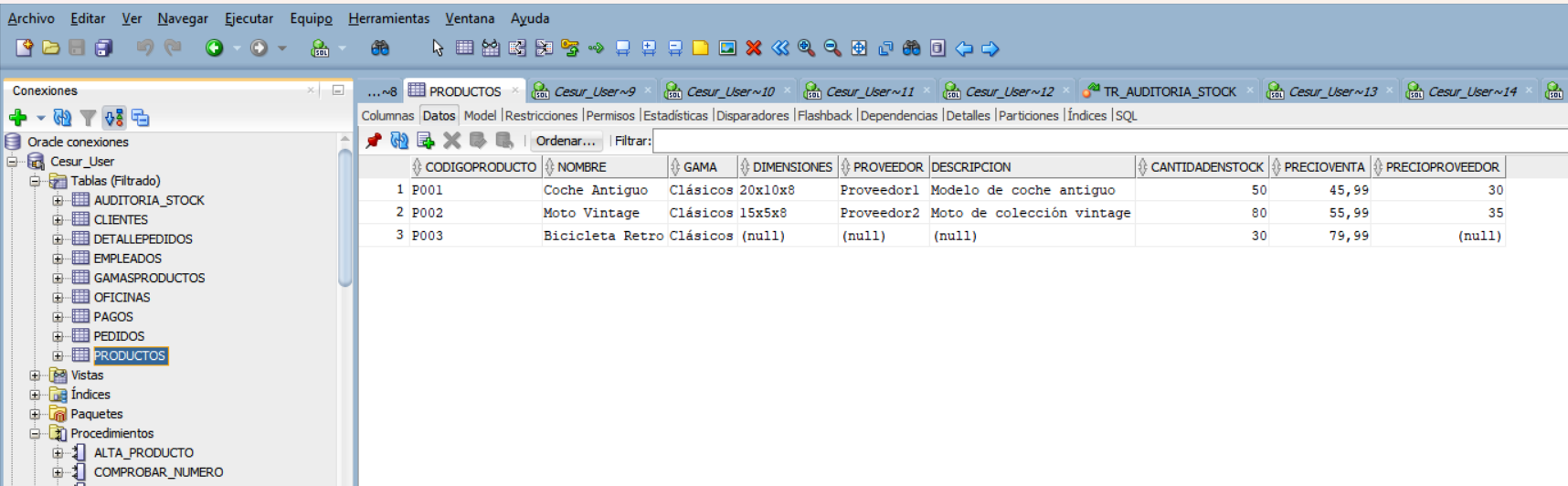
Ahora ejecutamos el procedimiento para comprobar si funciona correctamente:

BEGIN

ALTA\_PRODUCTO('P003', 'Bicicleta Retro', 'Clásicos', 30, 79.99);

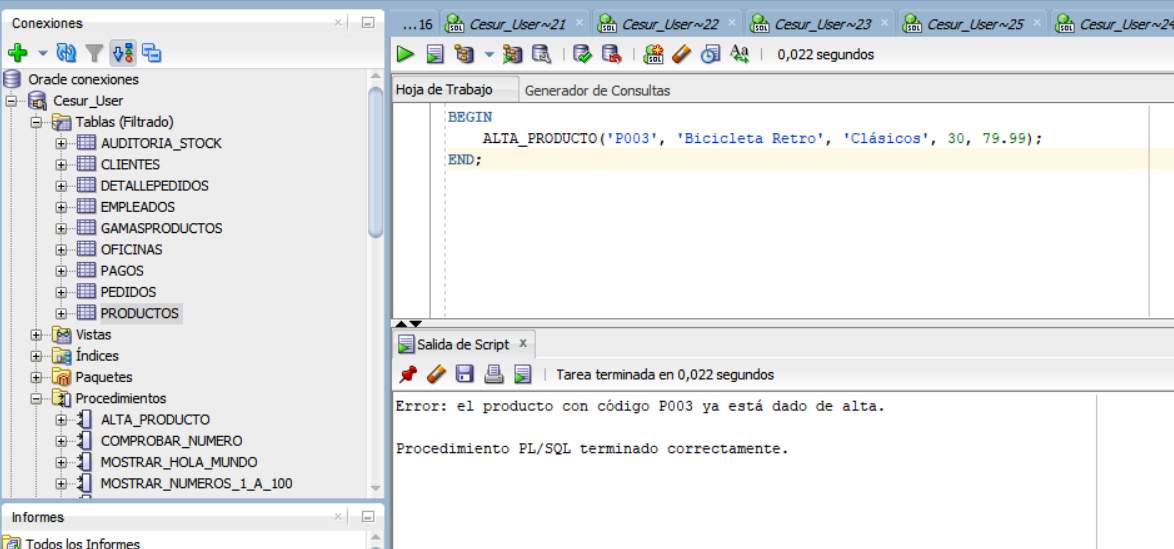
END;





Comprobado que se insertó el producto P003 correctamente.

Ahora para comprobar si el control de excepciones funciona correctamente, volvemos a insertar el mismo producto.



Vemos la salida de la consola que el control de excepciones es correcto y coherente con el código realizado.

# 06

conclusiones

El recorrido práctico nos dejó una base sólida sobre cómo programar en entornos de base de datos, mostrando que los procedimientos guardados, funciones y disparadores son esenciales para automatizar la gestión de datos en una organización. En la mayoría de los casos, optimizamos procesos clave –como el control de stock con triggers–, utilizamos cursores para realizar consultas específicas y manejamos excepciones para proteger la integridad de la información.

Con Oracle SQL Developer logramos crear, modificar y ejecutar las sentencias SQL necesarias para mejorar la administración de productos y pedidos. Nos topamos con desafíos interesantes, como la validación previa de datos antes de integrarlos al sistema, lo que nos empujó a aplicar excepciones para sortear errores y mantener siempre la coherencia, aunque a veces el reto se notara en cada detalle.

En definitiva, la experiencia fue completa, retadora y enriquecedora. No solo afinamos nuestras destrezas en la programación de bases de datos, sino que también aprendimos a combinar distintos componentes y herramientas dentro de un sistema de gestión de información. Los prodecimientos y triggers demostraron ser recursos clave para automatizar tareas repetitivas y agilizar las operaciones diarias –lo que se traduce en un impulso a la productividad en cualquier organización–, ampliando nuestra perspectiva profesional y dejándonos, de manera algo caótica pero muy formativa, mejor preparados para abordar nuevos proyectos.

# 07

referencias

<https://www.oracle.com/es/database/sqldeveloper/>

<https://download.oracle.com/oll/tutorials/SQLDeveloper/index.htm>

[https://www.tutorialspoint.com/plsql/plsql\_procedures.htm](https://www.tutorialspoint.com/plsql/plsql_procedures.htm?utm_source=chatgpt.com)

<https://www.youtube.com/playlist?list=PLiLpmqwkwkCt0QeXD8j7BwIoOaBGBRrZC>

<https://www.youtube.com/watch?v=6FSKfcimXIw>

<https://www.guru99.com/subprograms-procedures-functions-pl-sql.html>

<https://forums.oracle.com/ords/apexds/post/working-with-cursors-in-a-procedure-7188>

<https://www.reddit.com/r/learnprogramming/comments/sd8sec/best_way_to_learn_oracle_sql/>

<https://stackoverflow.com/questions/68845487/simple-stored-procedure-for-learning>

<https://www.youtube.com/watch?v=rnMa6HQ1_AE>