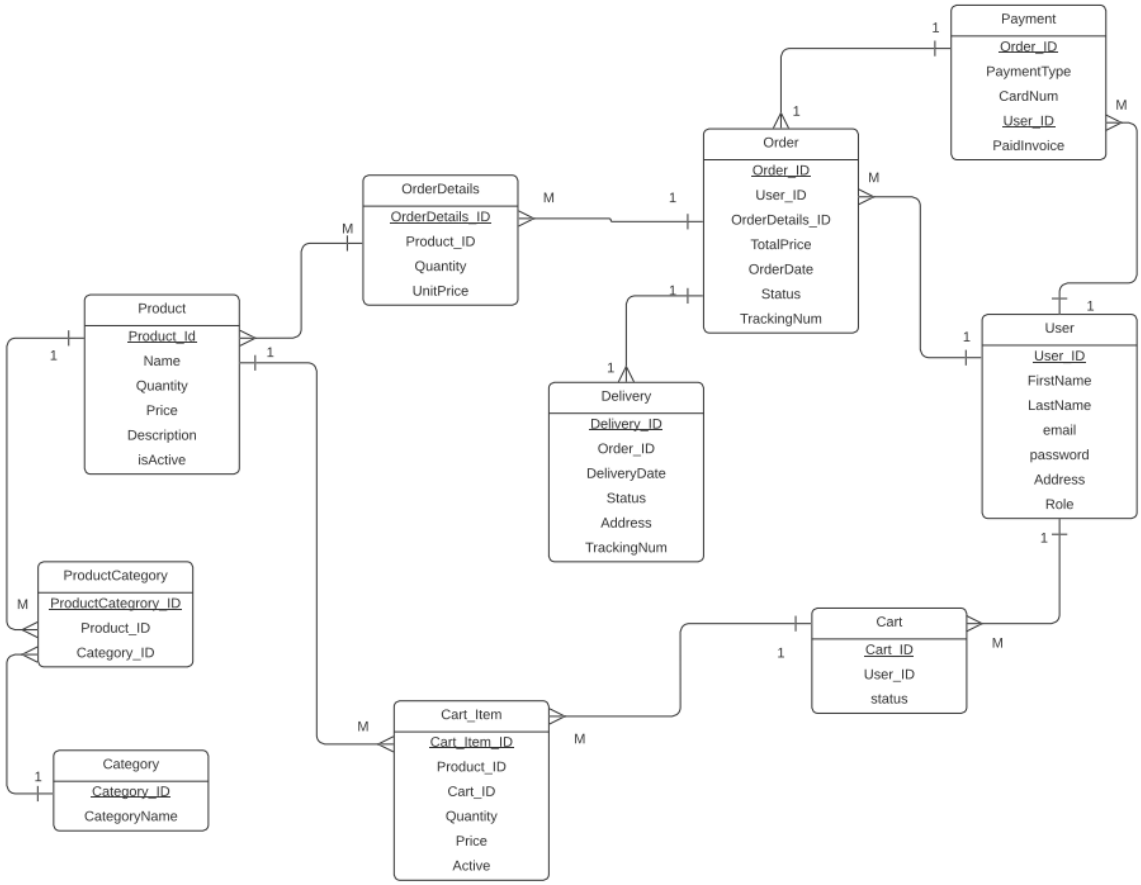
INF 305 | Database Management Systems 2
MIDTERM PROJECT

OUR TEAM :
Ibrakhim Namdar
Berik Yerman
Abzal Slamkozha
Ibrayev Aikyn
Lapidenov Shyngys

ERD:



About this project

Our database consists of such tables as: Product, Category, Product Category, User, Shopping Cart, Product in the cart, Order, Order Item, Delivery, Payment. Each table has its own purpose.

The "Product" table is in 1NF because each cell contains only one value, there are no duplicate rows and each row has a unique identifier (key).
The "Category" table is also in 1NF for the same reasons as the "Product" table.
The "OrderDetails" table is in 1NF because each cell contains only one value and each row has a unique identifier (key). However, there may be duplicate lines here, since each order may contain multiple items.
The "Cart_Item" table is also in 1NF for the same reasons as the "OrderDetails" table.
The "Order" table is in 1NF because each cell contains only one value, and each row has a unique identifier (key).
The "User" table is in 1NF because each cell contains only one value, there are no duplicate rows and each row has a unique identifier (key).
The "Payment" table is also in 1NF, because each cell contains only one value, and each row has a unique identifier (key).
The "Delivery" table is in 1NF because each cell contains only one value, and each row has a unique identifier (key).
The table "ProductCategory" is in the 2nd normal form (2NF), because it satisfies the requirements of 1NF, and each column depends only on the primary key, that is, the table contains two columns: "ProductID" and "CategoryID". Each "ProductID" corresponds to only one "CategoryID", and each "CategoryID" corresponds to only one "ProductID".
The "Cart" table is in 2NF because it meets the requirements of 1NF, and each column depends only on the primary key, that is, the table contains two columns: "CartID" and "UserID". Each "CartID" corresponds to only one "UserID", and each "UserID" can have multiple "cartids".
The "OrderDetails" table is in 2NF because it meets the requirements of 1NF and each column depends only on the primary key. However, the table contains two attributes that can be allocated to separate tables: "OrderID"

and "ProductID". Each "OrderID" can have multiple "ProductIDs", and each "ProductID" can be associated with multiple "OrderIds". Thus, you can create two separate tables: "Order" and "Product", and link them with a "many-to-many" relationship through an intermediate table "OrderDetails".

The "Payment" table is in BCNF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. In this table there is only one non-key attribute "Amount", which depends only on the primary key "OrderID".
The "Delivery" table is also in BCNF, because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. In this table there is only one non-key attribute "DeliveryAddress", which depends only on the primary key "OrderID".
The "Product" table is in 3NF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are two non-key attributes in this table: "ProductName" and "UnitPrice", which depend only on the primary key "ProductID". Also, the table does not contain transitive dependencies, since the "CategoryID" attribute depends only on the primary key "ProductID", and not on other non-key attributes.
The "Category" table is also in 3NF, because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. In this table there is only one non-key attribute "CategoryName", which depends only on the primary key "CategoryID".
The "OrderDetails" table is in 3NF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are three non-key attributes in this table: "OrderID", "ProductID", and "Quantity", which depend only on the primary key "OrderDetailsID".
The Cart_Item table is also in 3NF, because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are three non-key attributes in this table: "CartID", "ProductID", and "Quantity", which depend only on the primary key "CartItemID".
The "Order" table is in BCNF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are three non-key attributes in this table: "OrderDate", "UserID", and "Status", which depend only on the primary key "OrderID".

The "User" table is also in BCNF, because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are four non-key attributes in this table: "Username", "Password", "Email", and "FullName", which depend only on the primary key "UserID".
The "Payment" table is in BCNF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are two non-key attributes in this table: "PaymentMethod" and "Amount", which depend only on the primary key "PaymentID".
The "Delivery" table is in BCNF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are two non-key attributes in this table: "DeliveryAddress" and "DeliveryDate", which depend only on the primary key "OrderID".
The "Cart" table is in BCNF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are two non-key attributes in this table: "CartID" and "UserID", which depend only on the primary key "CartID".
The "ProductCategory" table is in 3NF because it satisfies the requirements of 1NF, 2NF, and each non-key attribute depends only on the primary key. There are two non-key attributes in this table: "ProductID" and "CategoryID", which depend only on the primary key "ProductCategoryID". Also, the table does not contain transitive dependencies, since the attributes "ProductID" and "CategoryID" do not depend on each other, but depend only on the primary key.

Product -> ProductCategory(one-to-many)
Category -> ProductCategory(one-to-many)
Product -> OrderDetails(one-to-many)
Product ->Cart_Item(one-to-many)
Order -> OrderDetails(one-to-many)
User -> Order(one-to-many)
Order -> Payment(one-to-one)
User -> Payment(one-to-one)
Order -> Delivery(one-to-one)

User -> Cart(one-to-many)
Cart -> Cart_Item(one-to-many)

Functions, Procedures and Triggers

Function which counts the number of records

```
CREATE OR REPLACE FUNCTION count_records(nameOfTable IN
VARCHAR2)
 RETURN NUMBER IS
  countOfRecord NUMBER := 0;
BEGIN
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || nameOfTable
INTO countOfRecord;
  RETURN countOfRecord;
END;
```

In this code we create a function that returns the number of rows in the table, first we create the function itself, we specify that we will return the number, and initialize the number itself, and in the main part of the code we write a regular sql query that returns the number of rows in the table, but instead of the name of the table we take the function parameter nameOfTable, and we write the answers in the variable countOfRecords, which we opened in the beginning of the code, and write EXECUTE IMMEDIATE so the query was dynamic. (we borrowed this from the Internet), and at the end we return the answer.

1.This line creates or replaces a function named "count_records" that takes a single input parameter "nameOfTable" of type VARCHAR2.

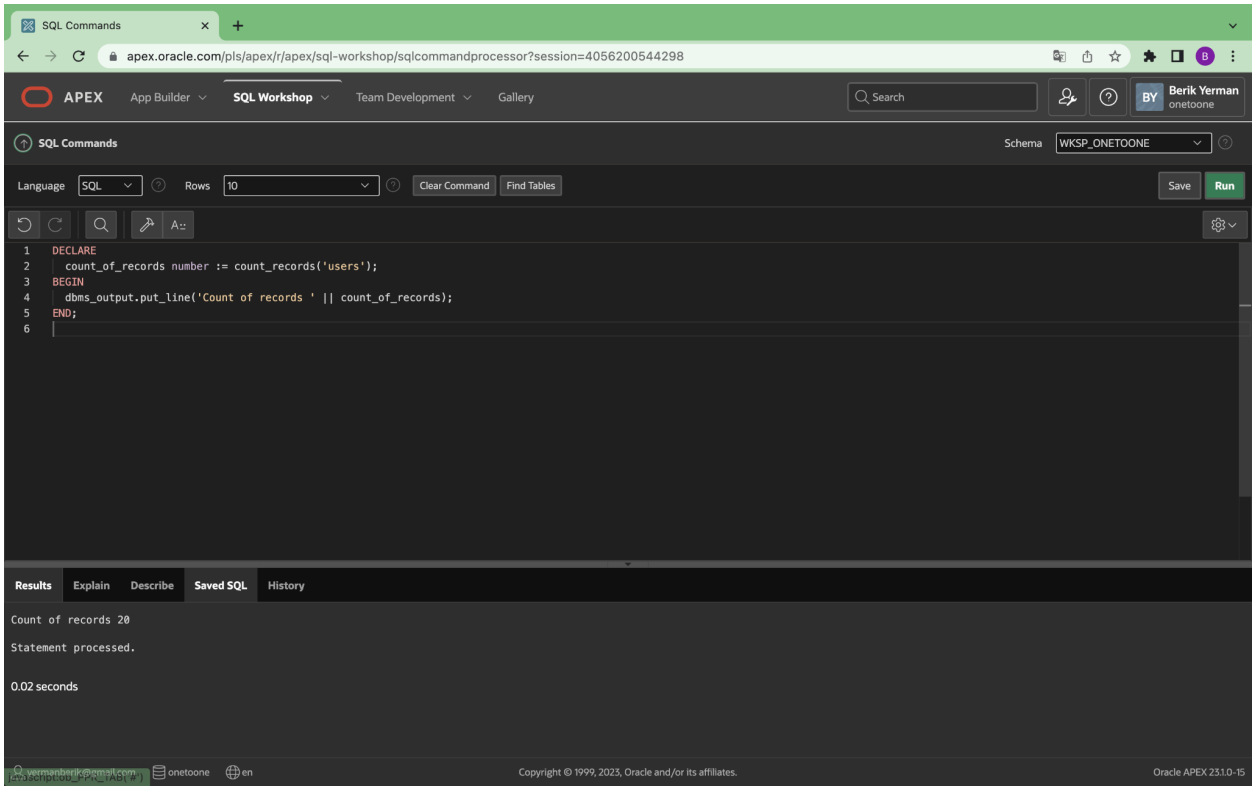
```
CREATE OR REPLACE FUNCTION count_records(nameOfTable IN
VARCHAR2)
```
2.Returns a NUMBER value.
```
RETURN NUMBER IS
```

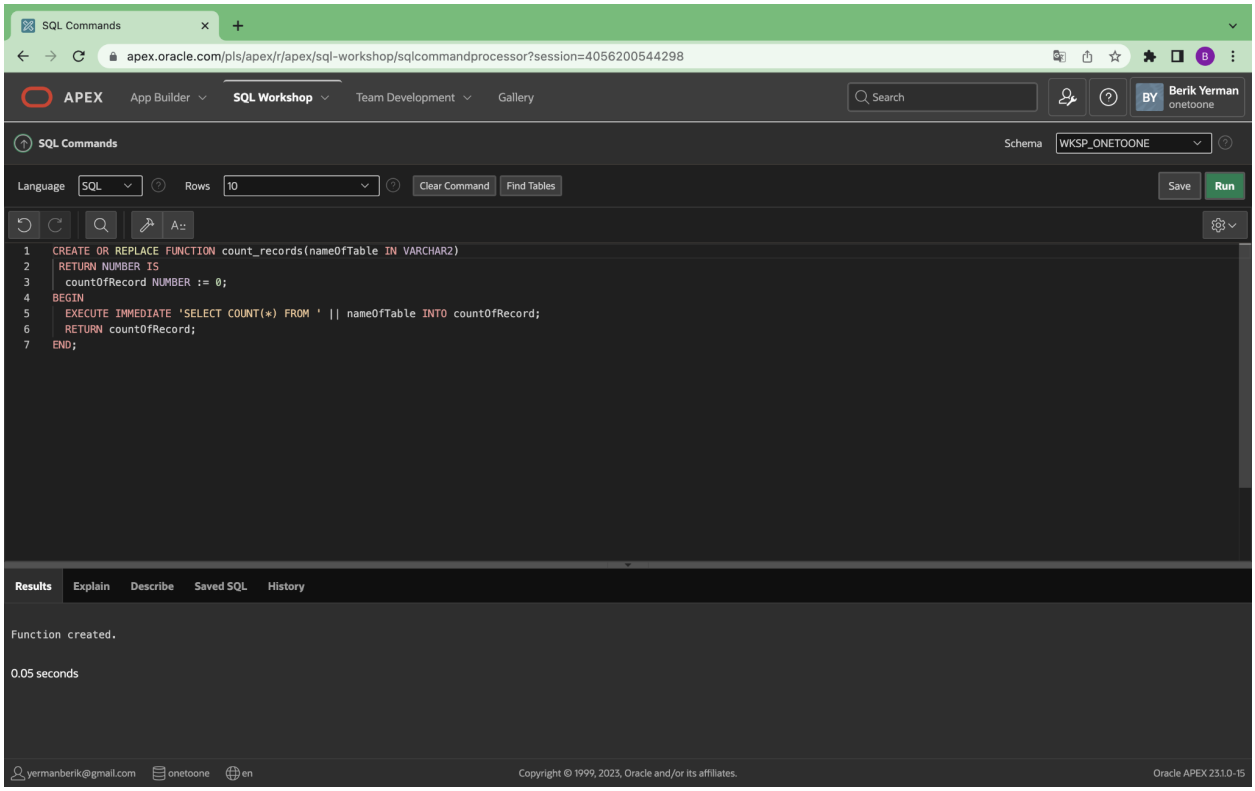
3.This line declares a local variable "count Of Record" of type NUMBER and initialize it with a value of zero.

```
    countOfRecord NUMBER := 0;
```

4.This line indicates the beginning of the executable section of the function.
```
BEGIN
```
5.This block of code uses dynamic SQL to execute a SELECT statement that counts the number of records in the specified table. The table name is passed as a parameter and concatenated with the SELECT statement using the || operator. The result of the SELECT statement is stored in the "countOfRecord" variable using the INTO clause
.
```
 EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || nameOfTable
```
6.This line returns the value of the "countOfRecord" variable as the result of the function.
```
 RETURN countOfRecord;
```

Then we check through this code:


SQL Commands
apex.oracle.com/pls/apex/r/apex/sql-workshop/sqlcommandprocessor?session=4056200544298
APEX
App Builder
SQL Workshop
Team Development
Gallery
Search
Berik Yerman
onetoone
SQL Commands
Schema
WKSP_ONETOONE
Language
SQL
Rows
10
Clear Command
Find Tables
Save
Run
1 DECLARE
2 count_of_records number := count_records('users');
3 BEGIN
4 dbms_output.put_line('Count of records ' || count_of_records);
5 END;
6
Results
Explain
Describe
Saved SQL
History
Count of records 20
Statement processed.
0.02 seconds
onetoone
en
Copyright © 1999, 2023, Oracle and/or its affiliates.
Oracle APEX 23.1.0-15


SQL Commands
apex.oracle.com/pls/apex/r/apex/sql-workshop/sqlcommandprocessor?session=4056200544298
APEX
App Builder
SQL Workshop
Team Development
Gallery
Search
Berik Yerman
onetoone
SQL Commands
Schema
WKSP_ONETOONE
Language
SQL
Rows
10
Clear Command
Find Tables
Save
Run
1 CREATE OR REPLACE FUNCTION count_records(nameOfTable IN VARCHAR2)
2 RETURN NUMBER IS
3 countOfRecord NUMBER := 0;
4 BEGIN
5 EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || nameOfTable INTO countOfRecord;
6 RETURN countOfRecord;
7 END;
Results
Explain
Describe
Saved SQL
History
Function created.
0.05 seconds
yermanberik@gmail.com
onetoone
en
Copyright © 1999, 2023, Oracle and/or its affiliates.
Oracle APEX 23.1.0-15

- **Procedure which uses SQL%ROWCOUNT to determine the number of rows affected**

```
CREATE OR REPLACE PROCEDURE update_iphone_quantity AS
BEGIN
  UPDATE Product
  SET Quantity = Quantity + 30
  WHERE Name LIKE 'Iphone%';
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated
successfully.');
END;
```

1.This line creates or replaces a stored procedure named "update_iphone_quantity".

```
CREATE OR REPLACE PROCEDURE update_iphone_quantity AS
```

2.This block of code updates the quantity of all products in the "Product" table that have a name starting with "Iphone" by adding 30 to the current quantity.

```
BEGIN
  UPDATE Product
  SET Quantity = Quantity + 30
  WHERE Name LIKE 'Iphone%';
```

3.This line outputs the number of rows that were updated by the previous SQL statement using the SQL%ROWCOUNT variable. The DBMS_OUTPUT.PUT_LINE function is used to print the message to the console or output window.

```
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated
successfully.');
```

- Add user-defined exception which disallows to enter title of item (e.g. book) to be less than 5 characters

```
CREATE OR REPLACE TRIGGER check_password_length
BEFORE INSERT ON Users
FOR EACH ROW
DECLARE
  too_short exception;
BEGIN
  IF LENGTH(:new.password) < 5 THEN
    RAISE too_short;
  END IF;
EXCEPTION
  WHEN too_short THEN
    raise_application_error(-20001, 'Password length should
be at least 5 characters');
END;
```

1.This line creates a trigger named "check_password_length" or replaces an existing trigger with the same name.

```
CREATE OR REPLACE TRIGGER check_password_length
```

2.This line specifies that the trigger should fire before an insert operation on the "Users" table.

```
BEFORE INSERT ON Users
```

3.This line declares an exception named "too_short" that will be raised if the password length is less than 5 characters.

```
DECLARE
  too_short exception;
```

4.This block of code checks the length of the "password" column in the new row being inserted. If the length is less than 5 characters, the trigger raises the "too_short" exception.

```
BEGIN
  IF LENGTH(:new.password) < 5 THEN
    RAISE too_short;
```

5.This block of code specifies what should happen if the "too_short" exception is raised. It raises an application error with the message "Password length should be at least 5 characters" and the error code -20001.

```
EXCEPTION
  WHEN too_short THEN
    raise_application_error(-20001, 'Password length should be at least 5 characters');
```

- Create a trigger before insert on any entity which will show the current number of rows in the table

```
1  -- CREATE OR REPLACE TRIGGER row_count_trigger
2  -- BEFORE INSERT ON USERS
3  -- DECLARE
4  --     row_count INT;
5  -- BEGIN
6  --     SELECT COUNT(*) INTO row_count FROM USERS;
7  --     DBMS_OUTPUT.PUT_LINE('Current row count: ' || row_count);
8  -- END;
9
10 INSERT INTO Users VALUES (31, 'Kairat', 'Nurtas', 'Qairosh@gmail.com', 'jurek123', 'Kazakhstan Estrade City 135', 'User');
```

Results | Explain | Describe | Saved SQL | History

```
Current row count: 30

1 row(s) inserted.

0.01 seconds
```

```
CREATE OR REPLACE TRIGGER row_count_trigger
BEFORE INSERT ON USERS
DECLARE
```

```
    row_count INT;
BEGIN
    SELECT COUNT(*) INTO row_count FROM USERS;
    DBMS_OUTPUT.PUT_LINE('Current row count: ' ||
row_count);
END;
```

1.This trigger was created with an aim to show the number of rows that existed before the insertion of any new data into the table

```
CREATE OR REPLACE TRIGGER row_count_trigger
```

2.Initially, it is required to create the trigger itself, named “row_count_trigger”

```
BEFORE INSERT ON USERS
```

3.Now we have to establish a postulate [BEFORE], which activates the trigger upon our intended action, in this case it’s before the action [BEFORE INSERT]

```
DECLARE
    row_count INT;
```

4.We now declare a new variable, for it is mandatory for the output of the results, as we assimilate the new variable with the aggregate function [COUNT(*)]

```
BEGIN
```

5.In this section the course of action which we have constructed is executed[BEGIN… END;].

```
SELECT COUNT(*) INTO row_count FROM USERS;
    DBMS_OUTPUT.PUT_LINE('Current row count: ' ||
row_count);
```

The actions themselves are:

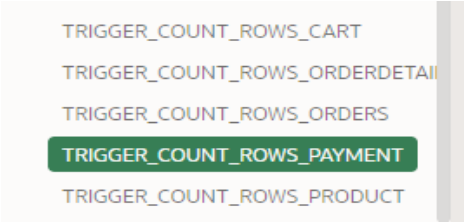
1) Numeration of the rows[COUNT(*)] from the table we had selected[USERS]

2) Incorporation of the new variable with the result of the aggregate function[DECLARE row_count INT]

3) Output of a message together with a variable which incorporated the result value of the aggregate function[DBMS_OUTPUT.PUT_LINE('Current row count: ' || row_count;]

```
END;
```

After the completion of the action sequence will new data be inserted into the table.



In light of the trigger's versatility, utilization of this trigger is not restricted with a single table, it is possible to use it everywhere.

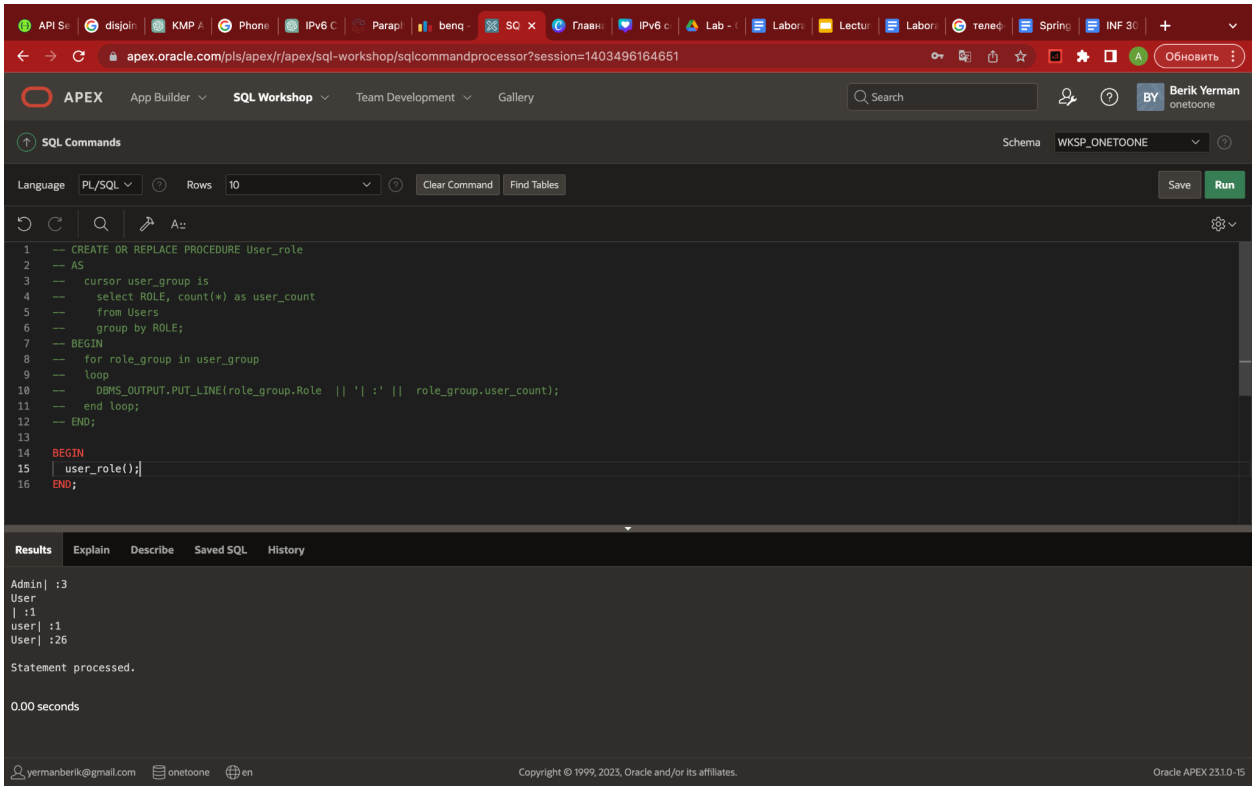
- **Procedure which does group by information**

```
CREATE OR REPLACE PROCEDURE User_role
AS
  cursor user_group is
    select ROLE, count(*) as user_count
    from Users
    group by ROLE;
BEGIN
  for role_group in user_group
```

```
  loop
    DBMS_OUTPUT.PUT_LINE(role_group.Role
|| '| :' ||  role_group.user_count);
  end loop;
END;

BEGIN
  user_role();
END;
```

This code creates or replaces our stored procedure which is called "`group_by_role`", which extracts data from the table through a cursor `c_group`, containing information about the quantity of users with their respective roles from the "`Users`" table.
After, the stored procedure sorts out data through the cursor and outputs the number of records grouping them by their "Role" column using the "`DBMS_OUTPUT.PUT_LINE`" command.



Invoking the procedure successfully displays the records, grouped by roles.

Step-by-Step instruction of the procedure

```
CREATE OR REPLACE PROCEDURE User_role
```

There we create a procedure itself.

```
AS
  cursor user_group is
    select ROLE, count(*) as user_count
    from Users
    group by ROLE;
```

Cursor “c_group” establishes queries to select the «Role» column and the number of records of users from the “Users” table, which are in turn grouped together separately.

```
BEGIN
```

This is the begin section from where the actions are executed

```
for role_group in user_group
  loop
    DBMS_OUTPUT.PUT_LINE(role_group.Role  ||
'| :' ||  role_group.user_count);
  end loop;
```

There we pass through the loop and use the “role_group” variable so we can output the role groups and the number of people with respective roles.

```
BEGIN
  user_role();
END;
```

There we call the procedure and its result output