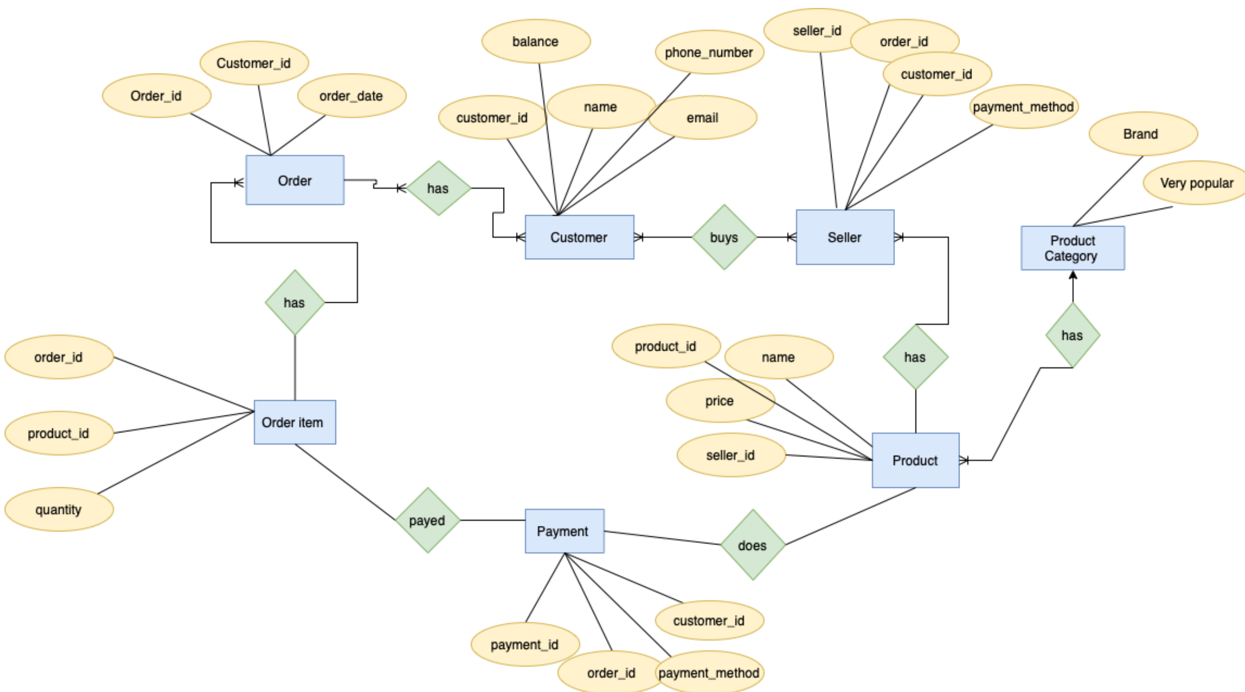
Online Shop

Who is do it
210103096 Nurgali Daulet

In this project first



Sure, I can provide additional information about this ERD for an online shop.

One important concept to understand is the use of primary and foreign keys in the tables. A primary key is a unique identifier for each record in a table. In this ERD, the primary keys for the tables are as follows:

- **Buyer: order_id**
- **Customer: customer_id**
- **Payment: payment_id**
- **Product: product_id**
- **Seller: seller_id**

A foreign key, on the other hand, is a column or set of columns in one table that refers to the primary key of another table. In this ERD, the foreign keys are as follows:

- **Buyer: customer_id (refers to Customer table)**
- **OrderItem: order_id (refers to Buyer table), product_id (refers to Product table)**
- **Payment: order_id (refers to Buyer table), customer_id (refers to Customer table)**
- **Product: seller_id (refers to Seller table)**

By using primary and foreign keys, it is possible to establish relationships between the different tables in the database, making it easier to query and update data.

Another important concept in this ERD is normalization. Normalization is the process of organizing data in a database to minimize redundancy and ensure data consistency. This ERD appears to be normalized up to at least third normal form (3NF), which means that each table is free of repeating groups and dependencies on non-key attributes.

Overall, this ERD provides a solid foundation for an online shop database, allowing for efficient management of customer orders, payments, and product information.

The ERD for the online shop consists of seven tables:

1. **Buyer: This table stores information about the buyer's orders. It has attributes such as order_id (primary key), customer_id (foreign key referencing customer table), and order_date.**
2. **OrderItem: This table stores information about the items that are part of an order. It has attributes such as order_id (foreign key referencing order table), product_id, and quantity.**
3. **Customer: This table stores information about the customers. It has attributes such as customer_id (primary key), balance, name, email, and phone_number.**
4. **Payment: This table stores information about the payments made by customers. It has attributes such as payment_id, order_id (foreign key referencing order table), customer_id (foreign key referencing customer table), and payment_method.**
5. **Product: This table stores information about the products available for sale. It has attributes such as product_id, name, price, and seller_id (foreign key referencing seller table).**

6. **Seller: This table stores information about the sellers. It has attributes such as seller_id (primary key), name, email, and phone_number.**
7. **Product Category**

Functional Dependencies

Functional dependency
Buyer(Order)

Order_id,customer_id → order_date

Order_id,order_date → customer_id

1. **Identify the functional dependencies (FDs) in the given relations.**
2. **Create new relations for each FD with a composite key that includes all the attributes on the right-hand side of the FD.**
3. **Remove the attributes involved in the FDs from the original relations, leaving only the attributes that form the keys for the new relations.**

Given the following relations:

R1(Order_id, customer_id → order_date)

R2(Order_id, order_date → customer_id)

We can identify the following FDs:

FD1: (Order_id, customer_id) → order_date (fromR1)

FD2: (Order_id, order_date) → customer_id (from R2)

To convert these relations to 3NF, we can follow these steps:

Step 1: Create a new relation for FD1 with a composite key of (Order_id, customer_id) and an attribute of order_date.

R3(Order_id, customer_id, order_date)

Step 2: Create a new relation for FD2 with a composite key of (Order_id, order_date) and an attribute of customer_id.

R4(Order_id, order_date, customer_id)

Step 3: Remove the attributes involved in the FDs from the original relations, leaving only the attributes that form the keys for the new relations.

R1(Order_id, customer_id)

R2(Order_id, order_date)

Now, let's check if these relations are in 3NF or not:

1. **All attributes in each relation are atomic - Yes, all attributes are atomic.**
2. **There are no transitive dependencies - Yes, there are no transitive dependencies since we created new relations for each FD with a composite key that includes all the attributes on the right-hand side of the FD.**

Therefore, the resulting relations are in 3NF.

OrderItem

1. **Order_id(FK) → product_id(FK), quantity**
2. **Identify the functional dependencies in the relation.**
3. **Eliminate transitive dependencies.**
4. **Create new relations for the eliminated dependencies.**

Given Relation:

Order_id(FK) → product_id(FK), quantity

Step 1: Identify the functional dependencies in the relation.

The given relation has two functional dependencies:

- **Order_id → product_id**
- **Order_id → quantity**

Step 2: Eliminate transitive dependencies.

There are no transitive dependencies in this relation.

Step 3: Create new relations for the eliminated dependencies.

The given relation is already in 2NF because it has only one composite primary key, Order_id and product_id. To convert it into 3NF, we need to separate the non-key attribute quantity into a separate relation.

New relations:

Relation 1: Order (Order_id(PK))

- **Order_id**

Relation 2: Product (product_id(PK), quantity)

- **product_id**
- **quantity**

Therefore, the given relation is now in 3NF.

Customer

1. **Customer_id(PK) → name, email, phone_number,balance**

1. **Identify the functional dependencies in the relation.**
2. **Eliminate transitive dependencies.**
3. **Create new relations for the eliminated dependencies.**

   Given Relation:

Customer_id(PK) → name, email, phone_number, balance

Step 1: Identify the functional dependencies in the relation.

The given relation has one functional dependency:

- **Customer_id → name, email, phone_number, balance**

Step 2: Eliminate transitive dependencies.

There are no transitive dependencies in this relation.

Step 3: Create new relations for the eliminated dependencies.

Since there are no transitive dependencies in this relation, it is already in 3NF.

Therefore, the given relation is already in 3NF and does not require any further normalization.

Payment

1. **Payment_id → order_id, customer_id, payment_method**
2. **Order_id → customer_id, payment_method**

1. **Identify the functional dependencies in the relation.**
2. **Eliminate transitive dependencies.**
3. **Create new relations for the eliminated dependencies.**

Given Relation:

Payment_id → order_id, customer_id, payment_method

Order_id → customer_id, payment_method

Step 1: Identify the functional dependencies in the relation.

The given relation has two functional dependencies:

- **Payment_id → order_id, customer_id, payment_method**

- **Order_id → customer_id, payment_method**

Step 2: Eliminate transitive dependencies.

There is a transitive dependency between Payment_id and customer_id, and another between Order_id and payment_method.

New Relation 1:

Payment_id → order_id, payment_method

Order_id → customer_id

New Relation 2:

Payment_id → customer_id

New Relation 3:

Customer_id → name, email, phone_number

Step 3: Create new relations for the eliminated dependencies.

- **Relation 1: Payment_id → order_id, payment_method**
- **Relation 2: Order_id → customer_id**
- **Relation 3: Customer_id → name, email, phone_number**

All the new relations are in 3NF.

Therefore, the final 3NF relations are:

- **Payment_id → order_id, payment_method**
- **Order_id → customer_id**
- **Customer_id → name, email, phone_number**

Product

1. **product_id —> name, price, seller_id**

1. **Identify the functional dependencies in the relation.**
2. **Eliminate transitive dependencies.**
3. **Create new relations for the eliminated dependencies.**

Given Relation:

product_id → name, price, seller_id

Step 1: Identify the functional dependencies in the relation.

The given relation has one functional dependency:

- **product_id → name, price, seller_id**

Step 2: Eliminate transitive dependencies.

There are no transitive dependencies in this relation.

Step 3: Create new relations for the eliminated dependencies.

Since there are no transitive dependencies in this relation, it is already in 3NF.

Therefore, the given relation is already in 3NF and does not require any further normalization.

Seller

1. **seller_id(PK)—> name, email, phone_number**

1. **Identify the functional dependencies in the relation.**
2. **Eliminate transitive dependencies.**
3. **Create new relations for the eliminated dependencies.**

Given Relation:

seller_id(PK) → name, email, phone_number

Step 1: Identify the functional dependencies in the relation.

The given relation has one functional dependency:

- **seller_id → name, email, phone_number**

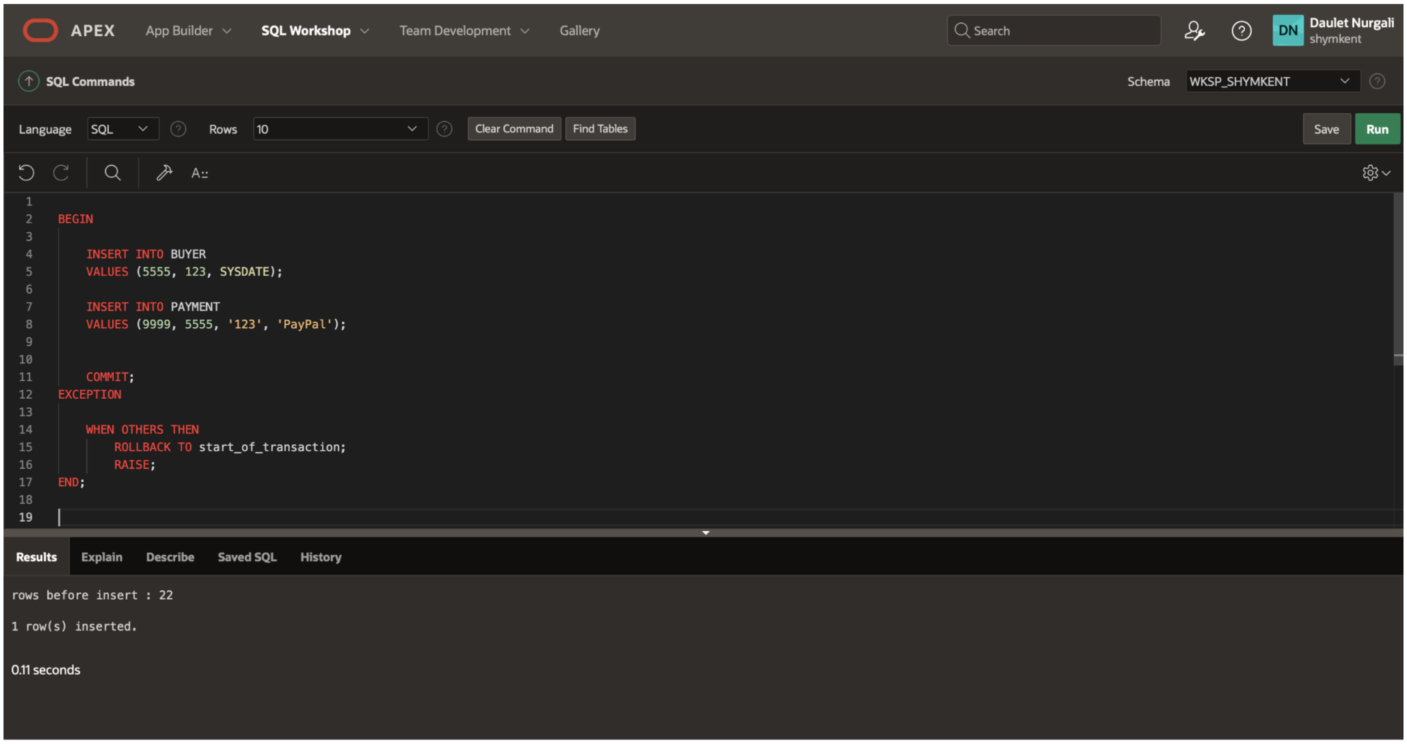
Step 2: Eliminate transitive dependencies.

There are no transitive dependencies in this relation.

Step 3: Create new relations for the eliminated dependencies.

Since there are no transitive dependencies in this relation, it is already in 3NF.

Therefore, the given relation is already in 3NF and does not require any further normalization.



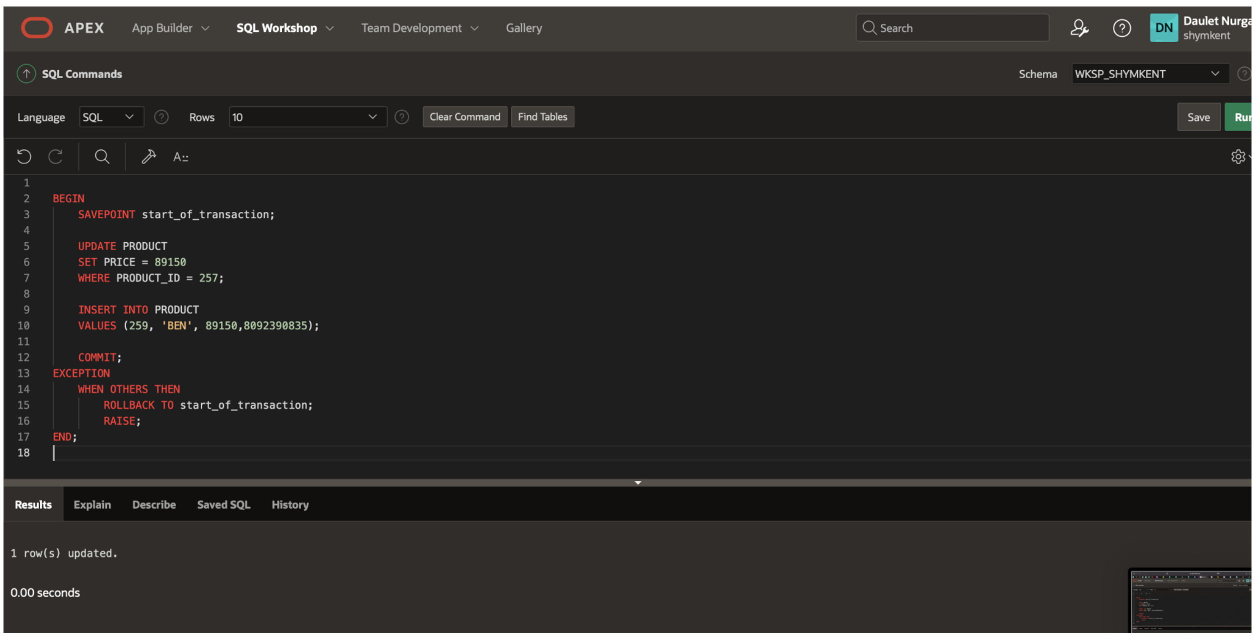
This code is a PL/SQL block that inserts a new record into the "BUYER" table and a corresponding record into the "PAYMENT" table, using the same "ORDER_ID" value for both.

The block begins with a "BEGIN" statement and sets a savepoint called "start_of_transaction". It then inserts a new record into the "BUYER" table with an "ORDER_ID" value of 5555, a "CUSTOMER_ID" value of 123, and the current system date as the "ORDER_DATE".

Next, it inserts a new record into the "PAYMENT" table with a "PAYMENT_ID" value of 9999, an "ORDER_ID" value of 5555 (matching the "ORDER_ID" value of the record just inserted into the "BUYER" table), a "CUSTOMER_ID" value of '123', and a "PAYMENT_METHOD" value of 'PayPal'.

If both of the inserts succeed, the "COMMIT" statement is executed, which makes the changes permanent. However, if an exception is raised during the execution of either the

"INSERT INTO BUYER" or "INSERT INTO PAYMENT" statement, the block catches the exception with a "WHEN OTHERS" clause, rolls back to the "start_of_transaction" savepoint, and re-raises the exception.

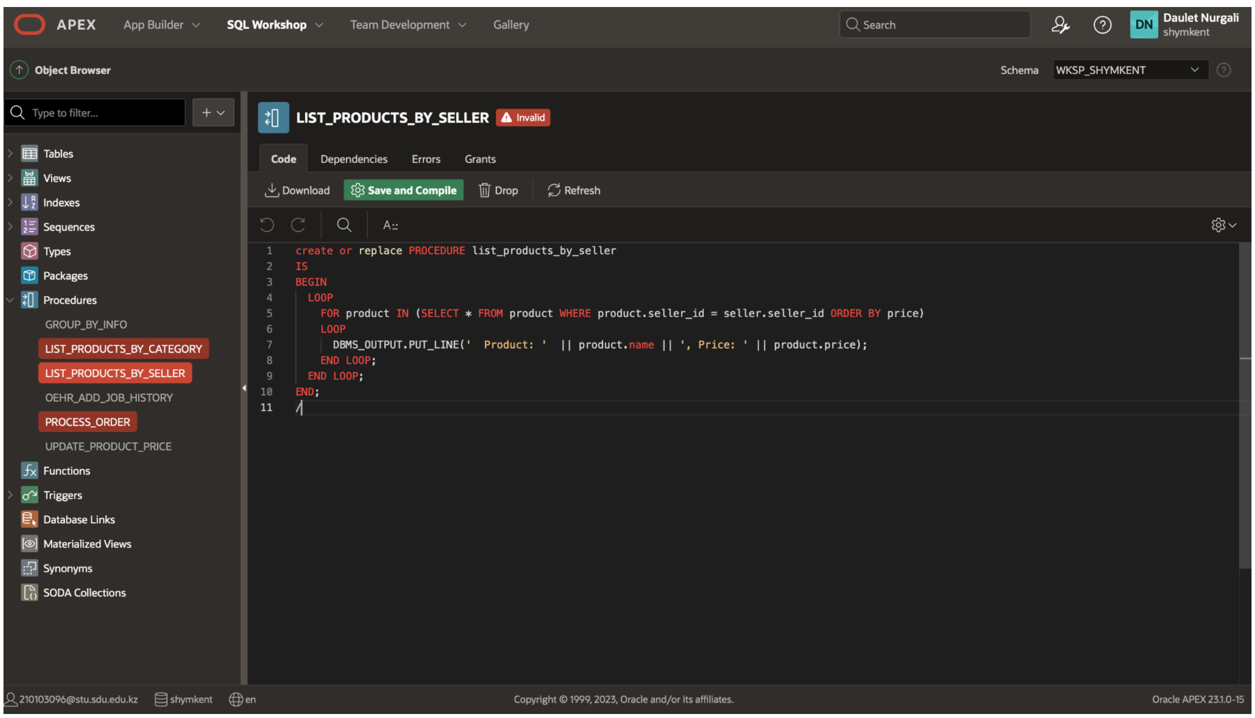


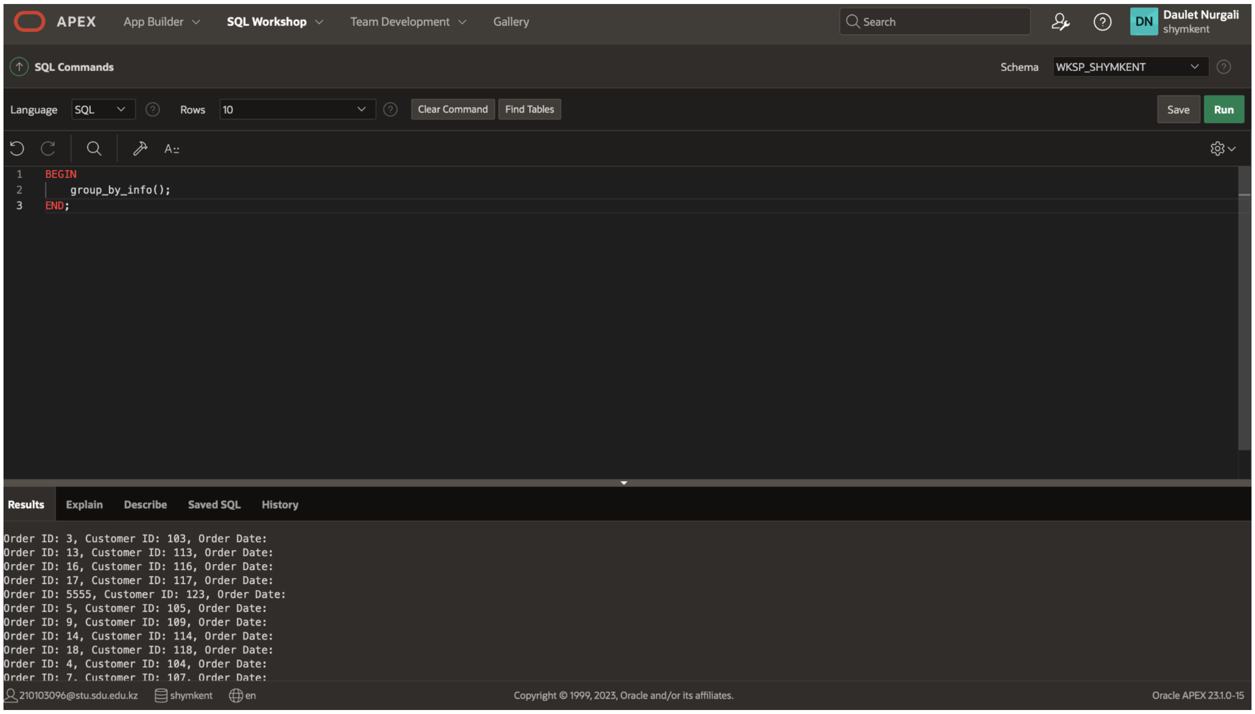
This code is a PL/SQL block that includes SQL statements to update and insert data into a table, with exception handling to handle any errors that might occur. Here's a breakdown of what's happening:

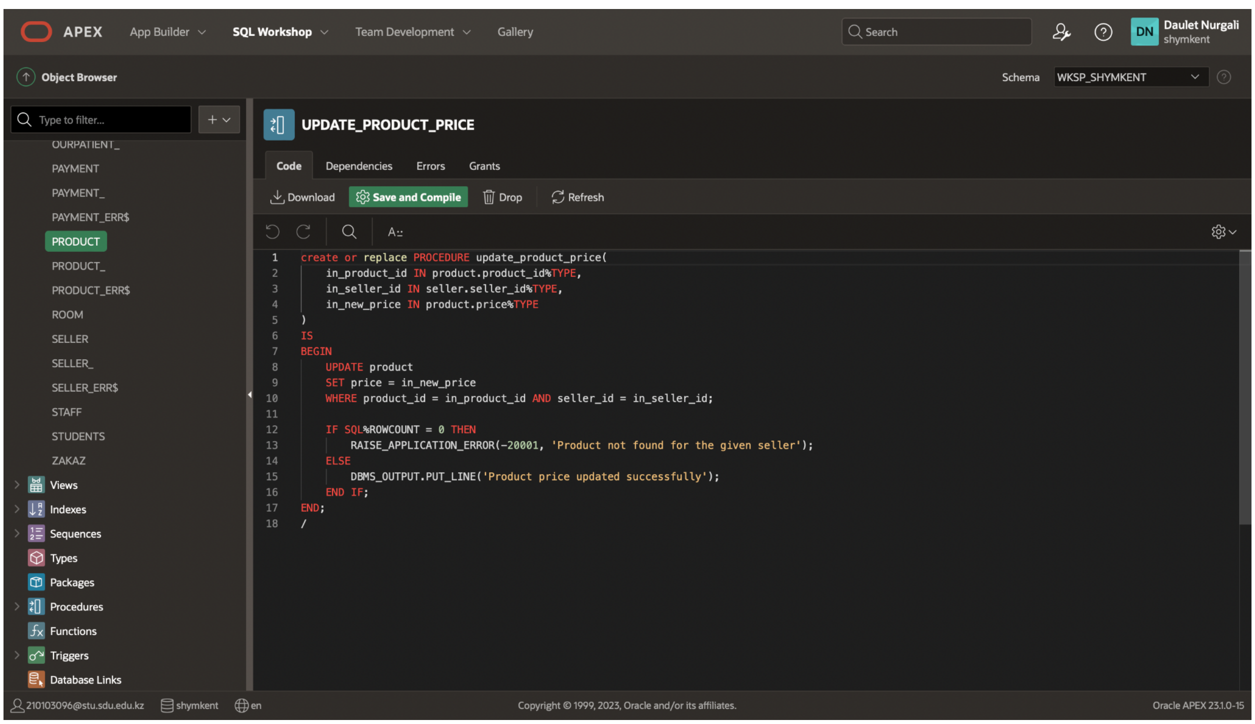
- The block starts with a SAVEPOINT statement to mark the start of the transaction.
- The UPDATE statement modifies the PRICE column of the PRODUCT table for the row with a PRODUCT_ID of 257.
- The INSERT statement adds a new row to the PRODUCT table with a PRODUCT_ID of 259, a NAME of 'BEN', a PRICE of 89150, and a SELLER_ID of 8092390835.

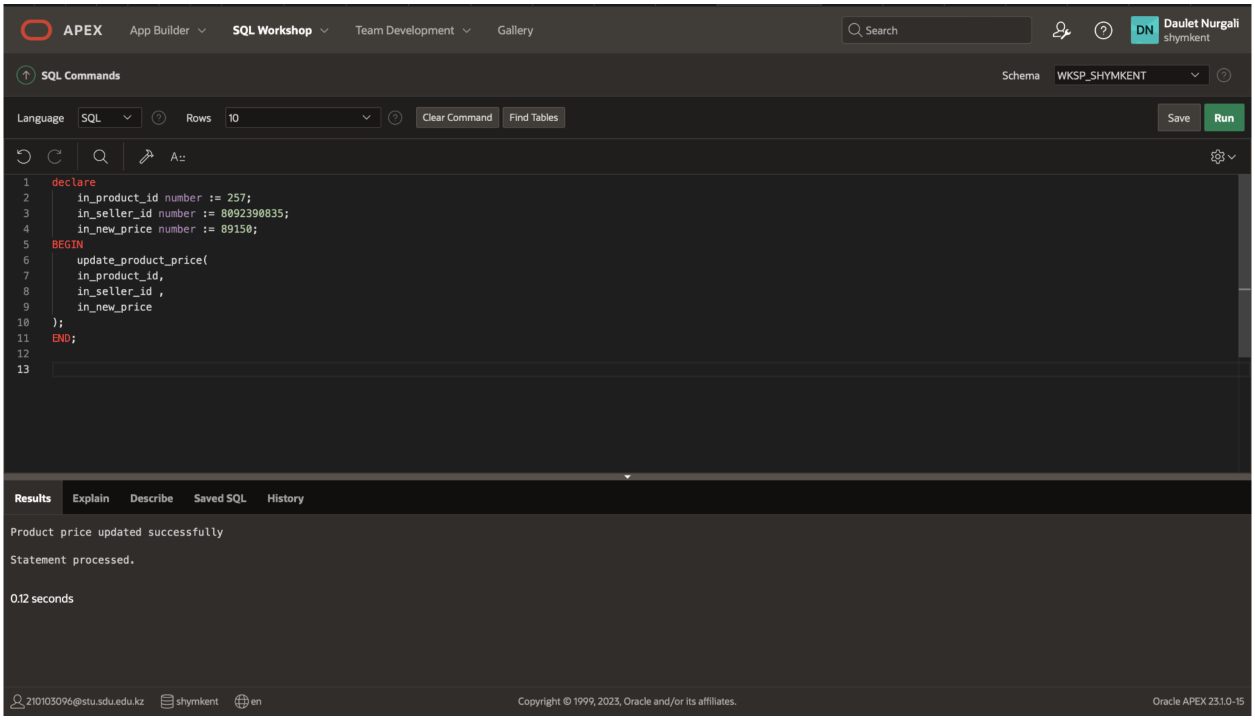
- If both statements are executed successfully, the COMMIT statement is executed to commit the transaction and make the changes permanent in the database.
- If an error occurs during the execution of either statement, the EXCEPTION block is executed.
- The ROLLBACK TO start_of_transaction statement rolls back the transaction to the SAVEPOINT, effectively undoing any changes made by the UPDATE and INSERT statements.

Procedure




APEX
App Builder
SQL Workshop
Team Development
Gallery
Search
Daulet Nurgali
shymkent
SQL Commands
Schema
WKSP_SHYMKENT
Language
SQL
Rows
10
Clear Command
Find Tables
Save
Run
BEGIN
group_by_info();
END;
Results
Explain
Describe
Saved SQL
History
Order ID: 3, Customer ID: 103, Order Date:
Order ID: 13, Customer ID: 113, Order Date:
Order ID: 16, Customer ID: 116, Order Date:
Order ID: 17, Customer ID: 117, Order Date:
Order ID: 5555, Customer ID: 123, Order Date:
Order ID: 5, Customer ID: 105, Order Date:
Order ID: 9, Customer ID: 109, Order Date:
Order ID: 14, Customer ID: 114, Order Date:
Order ID: 18, Customer ID: 118, Order Date:
Order ID: 4, Customer ID: 104, Order Date:
Order ID: 7, Customer ID: 107, Order Date:
210103096@stu.sdu.edu.kz
shymkent
en
Copyright © 1999, 2023, Oracle and/or its affiliates.
Oracle APEX 23.1.0-15


APEX
App Builder
SQL Workshop
Team Development
Gallery
Search
Daulet Nurgali
shymkent
Object Browser
Schema
WKSP_SHYMKENT
Type to filter...
OURPATIENT_
PAYMENT
PAYMENT_
PAYMENT_ERR$
PRODUCT
PRODUCT_
PRODUCT_ERR$
ROOM
SELLER
SELLER_
SELLER_ERR$
STAFF
STUDENTS
ZAKAZ
Views
Indexes
Sequences
Types
Packages
Procedures
Functions
Triggers
Database Links
UPDATE_PRODUCT_PRICE
Code
Dependencies
Errors
Grants
Download
Save and Compile
Drop
Refresh
create or replace PROCEDURE update_product_price(
in_product_id IN product.product_id%TYPE,
in_seller_id IN seller.seller_id%TYPE,
in_new_price IN product.price%TYPE
)
IS
BEGIN
UPDATE product
SET price = in_new_price
WHERE product_id = in_product_id AND seller_id = in_seller_id;
IF SQL%ROWCOUNT = 0 THEN
RAISE_APPLICATION_ERROR(-20001, 'Product not found for the given seller');
ELSE
DBMS_OUTPUT.PUT_LINE('Product price updated successfully');
END IF;
END;
/
210103096@stu.sdu.edu.kz
shymkent
en
Copyright © 1999, 2023, Oracle and/or its affiliates.
Oracle APEX 23.1.0-15



This PL/SQL code defines a procedure called update_product_price which updates the price of a product for a given seller. The procedure takes in three input parameters - in_product_id, in_seller_id, and in_new_price, which are of the same data types as their corresponding columns in the product and seller tables.

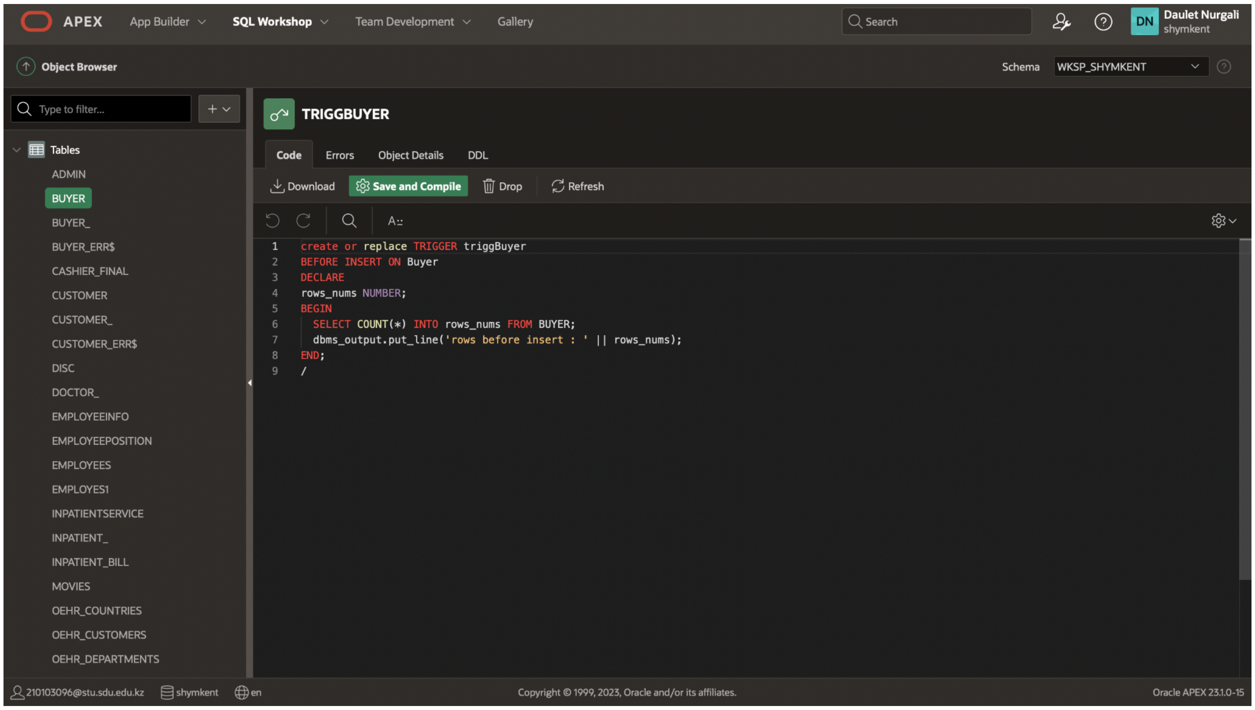
The UPDATE statement in the procedure updates the price column of the product table for the specified product_id and seller_id with the new price provided in the input parameter in_new_price.

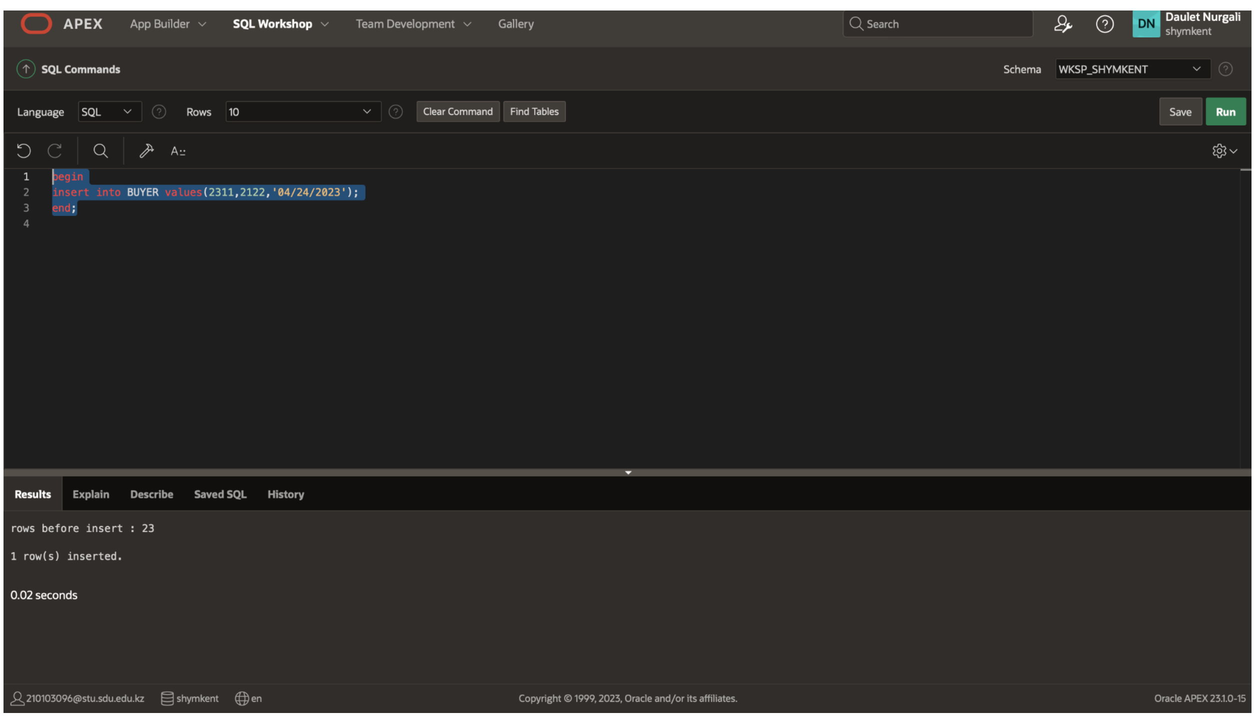
The IF statement that follows checks if the UPDATE statement affected any rows. If no rows were updated, it means that the product with the given product_id and seller_id does not exist, and the procedure raises an error with the RAISE_APPLICATION_ERROR statement.

If the UPDATE statement affected one or more rows, the ELSE block executes, and the procedure prints a message to indicate that the product price has been updated successfully using the DBMS_OUTPUT.PUT_LINE statement.

In summary, this PL/SQL code defines a procedure to update the price of a product for a given seller and ensures that the operation is successful and that the product exists in the database before updating its price.

Trigger

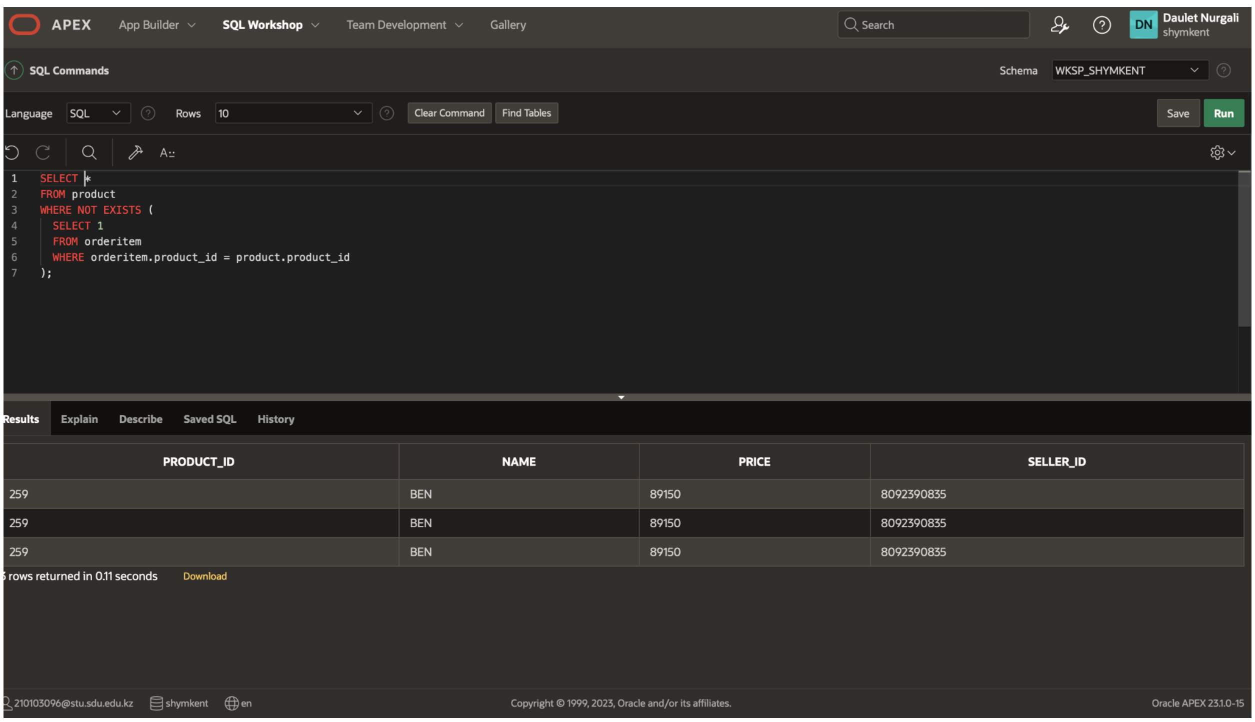

APEX
App Builder
SQL Workshop
Team Development
Gallery
Search
DN
Daulet Nurgali
shymkent
Object Browser
Schema
WKSP_SHYMKENT
Type to filter...
Tables
ADMIN
BUYER
BUYER_
BUYER_ERR$
CASHIER_FINAL
CUSTOMER
CUSTOMER_
CUSTOMER_ERR$
DISC
DOCTOR_
EMPLOYEEINFO
EMPLOYEEPOSITION
EMPLOYEES
EMPLOYES1
INPATIENTSERVICE
INPATIENT_
INPATIENT_BILL
MOVIES
OEHR_COUNTRIES
OEHR_CUSTOMERS
OEHR_DEPARTMENTS
TRIGGBUYER
Code
Errors
Object Details
DDL
Download
Save and Compile
Drop
Refresh
create or replace TRIGGER triggBuyer
BEFORE INSERT ON Buyer
DECLARE
rows_nums NUMBER;
BEGIN
SELECT COUNT(*) INTO rows_nums FROM BUYER;
dbms_output.put_line('rows before insert : ' || rows_nums);
END;
/
210103096@stu.sdu.edu.kz
shymkent
en
Copyright © 1999, 2023, Oracle and/or its affiliates.
Oracle APEX 23.1.0-15


APEX
App Builder
SQL Workshop
Team Development
Gallery
Search
DN
Daulet Nurgali
shymkent
SQL Commands
Schema
WKSP_SHYMKENT
Language
SQL
Rows
10
Clear Command
Find Tables
Save
Run
begin
insert into BUYER values(2311,2122,'04/24/2023');
end;
Results
Explain
Describe
Saved SQL
History
rows before insert : 23
1 row(s) inserted.
0.02 seconds
210103096@stu.sdu.edu.kz
shymkent
en
Copyright © 1999, 2023, Oracle and/or its affiliates.
Oracle APEX 23.1.0-15

The trigger code first declares a variable named rows_nums of type NUMBER. Inside the trigger, a SELECT COUNT(*) statement is executed to count the number of rows in the BUYER table and the result is stored in the rows_nums variable.

Finally, the trigger code uses the dbms_output.put_line function to display a message to the console showing the number of rows in the BUYER table before the insert operation is executed.

Therefore, this trigger will be fired every time a row is inserted into the BUYER table, and it will display the number of rows in the table before the insert operation is performed.

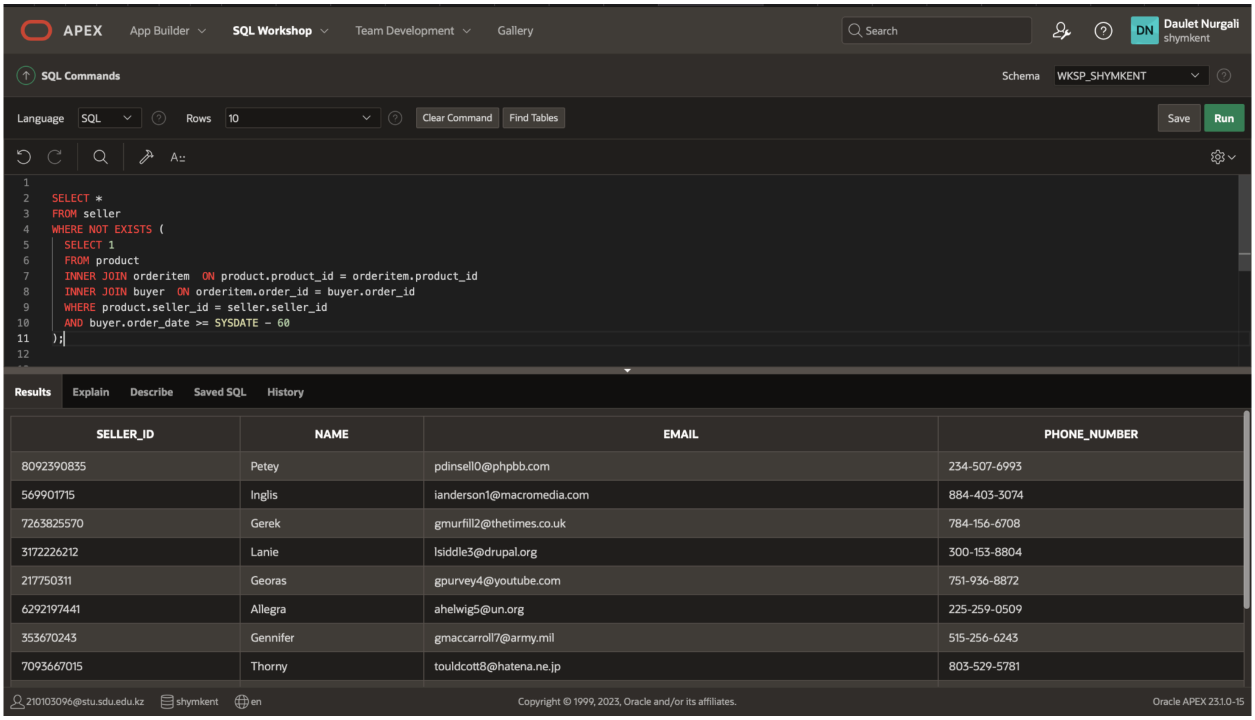


This SQL query is selecting all rows from the seller table where there does not exist any product sold by the seller within the last 60 days.

The query uses a subquery to check if there exists any product sold by the seller within the last 60 days. The subquery joins the product, orderitem, and buyer tables on their foreign key relationships to get the necessary data. It filters the results based on the seller_id of the seller in the main query and the order_date of the buyer table.

The NOT EXISTS operator in the main query checks if the subquery returns any rows. If the subquery returns no rows, it means that there does not exist any product sold by the seller within the last 60 days, and the seller is selected by the main query.

In summary, this query selects all sellers who have not sold any products within the last 60 days.




```sql
SELECT *
FROM seller
WHERE NOT EXISTS (
  SELECT 1
  FROM product
  INNER JOIN orderitem  ON product.product_id = orderitem.product_id
  INNER JOIN buyer  ON orderitem.order_id = buyer.order_id
  WHERE product.seller_id = seller.seller_id
  AND buyer.order_date >= SYSDATE - 60
);
```

SELLER_ID	NAME	EMAIL	PHONE_NUMBER
8092390835	Petey	pdinsell0@phpbb.com	234-507-6993
569901715	Inglis	ianderson1@macromedia.com	884-403-3074
7263825570	Gerek	gmurfill2@thetimes.co.uk	784-156-6708
3172226212	Lanie	lsiddle3@drupal.org	300-153-8804
217750311	Georas	gpurvey4@youtube.com	751-936-8872
6292197441	Allegra	ahelwig5@un.org	225-259-0509
353670243	Gennifer	gmaccarroll7@army.mil	515-256-6243
7093667015	Thorny	touldcott8@hatena.ne.jp	803-529-5781