# GROUP 8 ONLINE BANKING SYSTEM

# USER GUIDE


**Made By:**

- Aryan Khurana
- Bhanu Rakshita Paul
- Shivkumar Virendranath Raval

# APPLICATION INTRODUCTION:

For the DBS311 Final Project, we have made an **'Online Banking System'** Application.
In order to model the database for this application we have used a combination of tables which are related using keys.

# TABLES:

1. **employees table:** This table would store information about the bank's employees, such as their employee ID, name, position, province, and salary. This information would be used to identify and manage employees in the company.

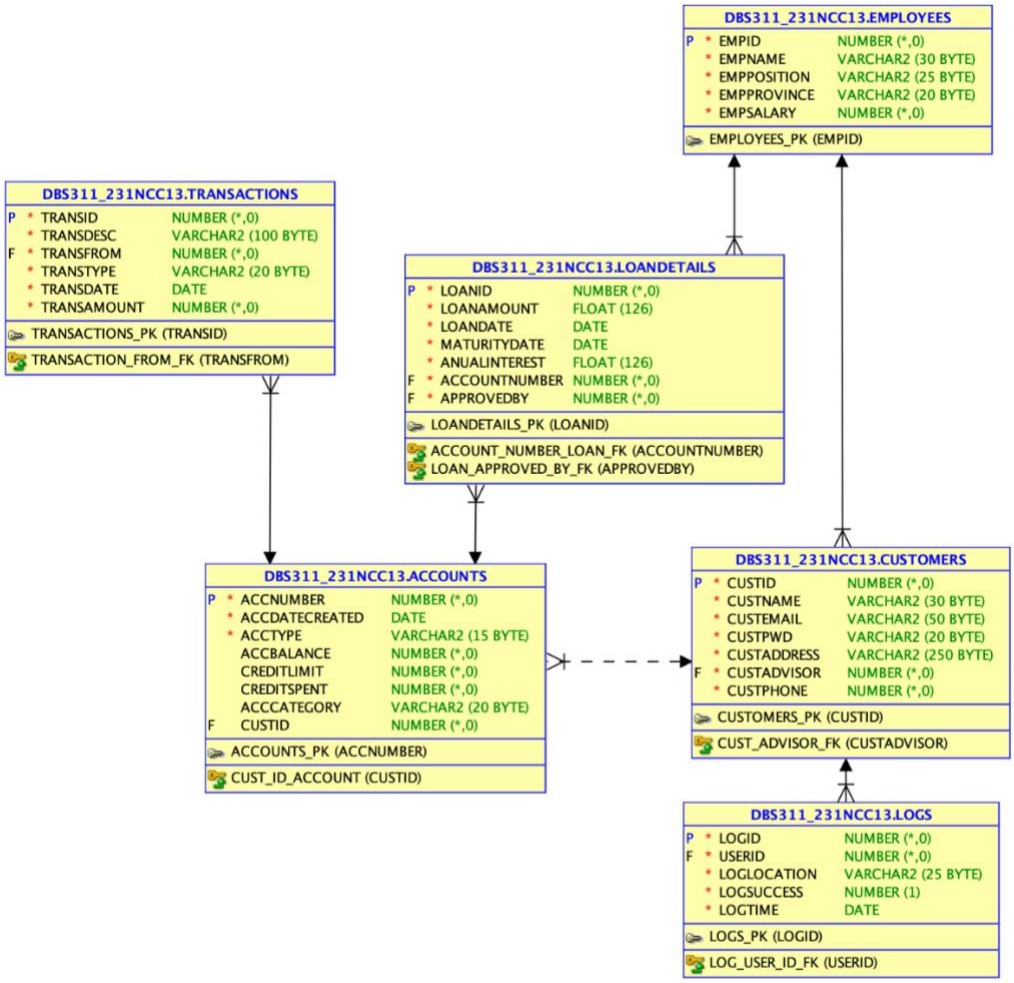
2. **customers table:** This table would store information about the bank's customers, such as their customer ID, name, email, password, address, advisor, and phone number. This information would be used to manage customer accounts, facilitate customer transactions, and provide personalized service to customers based on their needs and preferences.

3. **logs table:** This table would record all online activity carried out by the bank's customers, including log-in attempts and other system interactions. This information would be used for security and audit purposes, to track suspicious activity and potential fraud, and to troubleshoot technical issues that customers may encounter.

4. **accounts table:** This table would store information about the bank's customer accounts, including the account number, date created, type of account, balance, credit limit, credit spent, category, and customer ID. This information would be used to manage customer accounts, track their transactions and balances, and provide relevant information to customers about their account status and history.

5. **loandetails table:** This table would store information about the bank's loans, including the loan ID, loan amount, date, maturity date, annual interest, account number, and approval by employee ID. This information would be used to manage the bank's loan portfolio, track loan payments and maturity, and ensure that loans are approved and managed in accordance with bank policies and regulations.

6. **transactions table:** This table would store information about all bank transactions, including the transaction ID, description, account number, type, date, and amount. This information would be used to track and manage all bank transactions, identify and prevent fraudulent or suspicious activity, and provide customers with accurate and up-to-date information about their transaction history.




---

# RELATIONSHIPS:

- One-to-Many Relationship between Employees and Customers
  The Employees table has a one-to-many relationship with the Customers table. This means that one employee can have many customers, but each customer can only have one assigned advisor. The custadvisor column in the Customers table is a foreign key referencing the empid column in the Employees table.

- One-to-Many Relationship between Customers and Logs
  The Customers table has a one-to-many relationship with the Logs table. This means that one customer can have many logs, but each log can only belong to one customer. The userid column in the Logs table is a foreign key referencing the custid column in the Customers table.

- One-to-Many Relationship between Customers and Accounts

The Customers table has a one-to-many relationship with the Accounts table. This means that one customer can have many accounts, but each account can only belong to one customer. The custid column in the Accounts table is a foreign key referencing the custid column in the Customers table.

- One-to-Many Relationship between Accounts and LoanDetails
  The Accounts table has a one-to-many relationship with the LoanDetails table. This means that one account can have many loans, but each loan can only belong to one account. The accountnumber column in the LoanDetails table is a foreign key referencing the accnumber column in the Accounts table.

- One-to-Many Relationship between Employees and LoanDetails
  The Employees table has a one-to-many relationship with the LoanDetails table. This means that one employee can approve many loans, but each loan can only be approved by one employee. The approvedby column in the LoanDetails table is a foreign key referencing the empid column in the Employees table.

- One-to-Many Relationship between Accounts and Transactions
  The Accounts table has a one-to-many relationship with the Transactions table. This means that one account can have many transactions, but each transaction can only belong to one account. The transfrom column in the Transactions table is a foreign key referencing the accnumber column in the Accounts table.

# CONTRIBUTION OF TEAM MEMBERS:

- ARYAN KHURANA
  - Created the **loandetails** and **transactions** table.
  - Wrote the insertion script for the tables mentioned.
  - Designed procedures to perform CRUD operations on the **loandetails** table.
  - Wrote C++ code to connect to our database and perform operations on it.
- BHANU RAKSHITA PAUL
  - Created the **customers** and **employees** table.
  - Wrote the insertion script for the tables mentioned.
  - Designed procedures to perform CRUD operations on the **customers** table.
  - Wrote C++ code to connect to our database and perform operations on it.
- SHIVKUMAR VIRENDRANATH RAVAL
  - Created the **logs** and **accounts** table.
  - Wrote the insertion script for the tables mentioned.
  - Designed procedures to perform CRUD operations on the **logs** table.
  - Wrote C++ code to connect to our database and perform operations on it.

# PL / SQL PROCEDURES:

- ## spLoanDetailsInsert

```
create or replace PROCEDURE spLoandetailsInsert (
    err_count OUT INTEGER,
    loan_id IN loandetails.loanid%type,
    loan_amount IN loandetails.loanamount%type,
    loan_date IN loandetails.loandate%type,
    maturity_date IN loandetails.maturitydate%type,
    annual_interest IN loandetails.anualinterest%type,
    account_number IN loandetails.accountnumber%type,
    approved_by IN loandetails.approvedby%type
) AS

BEGIN
INSERT INTO loandetails
VALUES (loan_id, loan_amount, loan_date, maturity_date, annual_interest, account_number, approved_by);

err_count := sql%rowcount;
COMMIT;

EXCEPTION
    WHEN NO_DATA_FOUND THEN err_count := -1;
    WHEN TOO_MANY_ROWS THEN err_count := -2;
    WHEN OTHERS THEN err_count := -3;

END spLoandetailsInsert;
```

  - This procedure takes a loan id (NUMBER(38,0)), loan amount (FLOAT), loan date (DATE), maturity date (DATE), annual interest (FLOAT), account number (NUMBER(38,0)) and approved by employee number (NUMBER(38,0)) as input. All of these are entered by the user in our program.
  - The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution.
  - potential error codes:
    - -1: When there is no data found by the procedure.
    - -2: When there are too many rows and there is a problem in grouping.
    - -3: Any other exception that might be thrown in the procedure.
  - Description: This procedure inserts a record into the loandetails table based on whatever inputs the user provides. It doesn't insert the data if the entered data doesn't match the specifications provided by the loandetails table.
  - Example of execution:

```
-- RUN INSERT PROCEDURE
DECLARE
    er INT;
BEGIN
    spLoanDetailsInsert(er, 21, 2000, TO_DATE('2022-01-01', 'yyyy-mm-dd'), TO_DATE('2023-01-01', 'yyyy-mm-dd'), 12,
    10283, 1826);
END;
```

  - An example output from C++ call to this procedure:

```
---------- Insert ----------
Loan ID: 22
Loan Amount: 2000
Loan Date (yyyy-mm-dd) : 2022-01-01
Maturity Date (yyyy-mm-dd) : 2023-01-01
Annual Interest: 12
Account Number: 10283
Employee number of the employee that approved the loan: 1826

SUCCESS: 1 row(s) inserted.
```

- spLoanDetailsUpdate

```
create or replace PROCEDURE spLoandetailsUpdate (
    err_count OUT INTEGER,
    loan_id IN loandetails.loanid%type,
    loan_amount IN loandetails.loanamount%type
) AS

BEGIN
UPDATE loandetails SET  loanamount = loan_amount
WHERE loanid = loan_id;

err_count := sql%rowcount;
COMMIT;

EXCEPTION
    WHEN NO_DATA_FOUND THEN err_count := -1;
    WHEN TOO_MANY_ROWS THEN err_count := -2;
    WHEN OTHERS THEN err_count := -3;

END spLoandetailsUpdate;
```

  - This procedure takes a loan id (NUMBER(38,0)) and loan amount (FLOAT) as input. Both of these values are inputted by the user.
  - The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution.
  - potential error codes:
    - -1: When there is no data found by the procedure.
    - -2: When there are too many rows and there is a problem in grouping.
    - -3: Any other exception that might be thrown in the procedure.
  - Description: This procedure updates the loan amount in a loandetails record that is chosen based on the id that the user inputs. The user also inputs the desired new loan amount.
  - Example of execution:

```
-- RUN UPDATE PROCEDURE
DECLARE
    er INT;
BEGIN
    spLoanDetailsUpdate(er, 21, 5000);
END;
```

  - An example output from C++ call to this procedure:

```
------------ Update -----------
Loan ID: 22
Loan Amount: 30000

SUCCESS: 1 row(s) updated.
```

- spLoanDetailsDelete

```
create or replace PROCEDURE spLoandetailsDelete (
    err_count OUT INTEGER,
    loan_id IN loandetails.loanid%type
) AS

BEGIN
DELETE FROM loandetails
WHERE loanid = loan_id;

err_count := sql%rowcount;
COMMIT;

EXCEPTION
    WHEN NO_DATA_FOUND THEN err_count := -1;
    WHEN TOO_MANY_ROWS THEN err_count := -2;
    WHEN OTHERS THEN err_count := -3;

END spLoandetailsDelete;
```

  - This procedure takes a loan id (NUMBER(38,0)) as an input which is inputted by the user.
  - The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution.
  - potential error codes:
    - -1: When there is no data found by the procedure.
    - -2: When there are too many rows and there is a problem in grouping.
    - -3: Any other exception that might be thrown in the procedure.
  - Description: This procedure deletes a record from the loandetails table based upon the id that is inputted by the user.
  - Example of execution:

```
-- RUN DELETE PROCEDURE
DECLARE
    er INT;
BEGIN
    spLoanDetailsDelete(er, 21);
END;
```

  - An example output from C++ call to this procedure:

```
---------- Delete ----------
Loan ID: 22

SUCCESS: 1 row(s) deleted.
```

- spLoanDetailsSelect

```
create or replace PROCEDURE spLoandetailsSelect (
    err_count OUT INTEGER,
    loan_id IN loandetails.loanid%type,
    loan_amount OUT loandetails.loanamount%type,
    loan_date OUT loandetails.loandate%type,
    maturity_date OUT loandetails.maturitydate%type,
    annual_interest OUT loandetails.anualinterest%type,
    account_number OUT loandetails.accountnumber%type,
    approved_by OUT loandetails.approvedby%type
) AS

BEGIN
SELECT loanamount, loandate, maturitydate, anualinterest, accountnumber, approvedby
INTO loan_amount, loan_date, maturity_date, annual_interest, account_number, approved_by
FROM loandetails
WHERE loanid = loan_id;

err_count := sql%rowcount;

COMMIT;

EXCEPTION
    WHEN NO_DATA_FOUND THEN err_count := -1;
    WHEN TOO_MANY_ROWS THEN err_count := -2;
    WHEN OTHERS THEN err_count := -3;

END spLoandetailsSelect;
```

  - This procedure takes a loan id (NUMBER(38,0)) as an input which is inputted by the user in our program.
  - The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution. The other outputs that we get are from the record that the procedure has fetched. Other outputs include loan amount, loan date, maturity date, annual interest, account number and the employee number of the employee that approved the loan.
  - potential error codes:
    - -1: When there is no data found by the procedure.
    - -2: When there are too many rows and there is a problem in grouping.
    - -3: Any other exception that might be thrown in the procedure.
  - Description: This procedure reads a record from the loandetails table based on the id that is inputted by the user.
  - Example of execution:

```
-- RUN SELECT PROCEDURE
DECLARE
    er NUMBER;
    loan_amount loandetails.loanamount%type;
    loan_date loandetails.loandate%type;
    maturity_date loandetails.maturitydate%type;
    annual_interest loandetails.anualinterest%type;
    account_number loandetails.accountnumber%type;
    approved_by loandetails.approvedby%type;
BEGIN
    spLoanDetailsSelect(er, 20, loan_amount, loan_date, maturity_date, annual_interest, account_number, approved_by);
    DBMS_OUTPUT.PUT_LINE('Loan Amount: ' || loan_amount);
    DBMS_OUTPUT.PUT_LINE('Loan Date: ' || loan_date);
    DBMS_OUTPUT.PUT_LINE('Maturity Date: ' || maturity_date);
    DBMS_OUTPUT.PUT_LINE('Annual Interest: ' || annual_interest);
    DBMS_OUTPUT.PUT_LINE('Account Number: ' || account_number);
    DBMS_OUTPUT.PUT_LINE('Approved By: ' || approved_by);
END;
```

  - An example output from C++ call to this procedure:

```
---------- SELECT -----------
Loan ID: 22
Loan Amount: 30000
Loan Date: 01-JAN-22
Maturity Date: 01-JAN-23
Annual Interest: 12
Account Number: 10283
Approved By: 1826

SUCCESS: 1 row(s) selected.
```

- spCustomersInsert

```
create or replace PROCEDURE SPcustomersINSERT(
err_code OUT INTEGER,
customer_id IN customers.custid%type,
customer_name IN customers.custname%type,
customer_email IN customers.custemail%type,
customer_pwd IN customers.custpwd%type,
customer_address IN customers.custaddress%type,
customer_advisor IN customers.custadvisor%type,
customer_phone IN customers.custphone%type) AS

BEGIN
    INSERT INTO customers
    VALUES (customer_id, customer_name, customer_email, customer_pwd,
    customer_address, customer_advisor, customer_phone);

    err_code := sql%rowcount;
    COMMIT;

    EXCEPTION
        WHEN OTHERS
            THEN err_code := -1;
END;
```

  - **REQUIRED INPUT PARAMETERS**: This procedure takes in customer_id (NUMBER), customer_name(VARCHAR(30)), customer_email (VARCHAR(50)), customer_pwd (VARCHAR(20)), customer_address (VARCHAR(250)), customer_advisor(NUMBER), customer_phone (NUMBER) as input.
  - **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution.
  - **potential error codes:**
    - -1: When the data couldn't be inserted. Occurred mostly when a constraint is violated
    - 1: When data is successfully updated
  - **Description:** This procedure inserts a record into the **'customers'** table based on whatever inputs the user provides. It doesn't insert the data if the entered data doesn't match the specifications provided by the customers table.
  - **Example of execution:**

```
-------------------
--1) INSERT
-------------------
SET SERVEROUTPUT ON;
DECLARE
err_code INT;
BEGIN
    spcustomersinsert(err_code, 1234, 'Bhanu', 'brpaul@myseneca.ca',
    'bhanu12345', 'Seneca College', 3726, 567187328);
    DBMS_OUTPUT.PUT_LINE(err_code);
END;
```

- spCustomersUpdate

```
create or replace PROCEDURE SPcustomersUPDATE(
err_code OUT INTEGER,
customer_id IN customers.custid%type,
address IN customers.custaddress%type) AS
BEGIN
    UPDATE CUSTOMERS SET custaddress=address
    WHERE custid=customer_id;

    err_code := SQL%ROWCOUNT;
    COMMIT;

    EXCEPTION
        WHEN NO_DATA_FOUND
            THEN err_code:=-2;
        WHEN OTHERS
            THEN err_code:=-1;
END;
```

- **REQUIRED INPUT PARAMETERS**: This procedure takes in customer_id (NUMBER), address (VARCHAR(250)) as inputs. The address is the new address of customer with the given id.
- **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution.
- **potential error codes:**
  - -1: When the data couldn't be updated. Occurred mostly when a constraint is violated
  - -2: When address could not be updated because the given record DNE.
  - 1: When data is successfully updated
- **Description:** This procedure updates the address of the given customer in the **'customers'** table based on whatever inputs the user provides. It doesn't update the data if the entered data doesn't match the specifications provided by the customers table.
- **Example of execution:**

```
--------------------
--2) UPDATE
--------------------
SET SERVEROUTPUT ON;
DECLARE
err_code INT;
BEGIN
    spcustomersupdate(err_code, 1234, 'New address');
    DBMS_OUTPUT.PUT_LINE(err_code);
END;
```

- spCustomersDelete

```
create or replace PROCEDURE SPcustomersDELETE(
err_code OUT INTEGER,
customer_id IN NUMBER) AS

BEGIN
    DELETE FROM customers
    WHERE custid=customer_id;

    err_code := SQL%ROWCOUNT;
    COMMIT;

    EXCEPTION
        WHEN OTHERS
            THEN err_code := -1;
END;
```

- **REQUIRED INPUT PARAMETERS**: This procedure takes in customer_id (NUMBER). This should an existing ID of customer in 'customers' table that needs to be deleted
- **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution.
- **potential error codes:**
  - -1: When the data couldn't be deleted. Occurred mostly when id could not be found in 'customers' table.
  - 1: When record is successfully deleted.
- **Description:** This procedure deletes the given customer in the **'customers'** table based on whatever inputs the user provides. It doesn't delete the data if the entered customer_id  does not exist in the customers table.
- **Example of execution:**

```
--------------------
--3) DELETE
--------------------
SET SERVEROUTPUT ON;
DECLARE
err_code INT;
BEGIN
    spcustomersDelete(err_code, 1234);
    DBMS_OUTPUT.PUT_LINE(err_code);
END;
```

- spCustomersSelect

```
create or replace PROCEDURE SPcustomersSELECT(
err_code OUT INTEGER,
cid IN customers.custid%type,
customer_id OUT customers.custid%type,
customer_name OUT customers.custname%type,
customer_phone OUT customers.custphone%type,
customer_email OUT customers.custemail%type) AS

BEGIN
    SELECT custid, custname, custphone, custemail
    INTO customer_id, customer_name, customer_phone, customer_email
    FROM CUSTOMERS
    WHERE custid=cid;

    err_code := SQL%ROWCOUNT;

    EXCEPTION
         WHEN TOO_MANY_ROWS
            THEN err_code := -2;
            --THEN dbms_output.put_line('Multiple customers returned');
        WHEN NO_DATA_FOUND
            THEN err_code := -3;
            --THEN dbms_output.put_line('No customer has the given advisor');
        WHEN OTHERS
            THEN err_code := -1;
            --THEN dbms_output.put_line('Error in fetching rows');
END;
```

- **REQUIRED INPUT PARAMETERS**: This procedure takes in c_id (NUMBER). This should an existing ID of customer in 'customers' table that needs to be fetched.
- **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is positive if there are no errors and is negative if there are errors in the procedure execution. It also gives out customer_id (NUMBER), customer_name(VARCHAR(30)), customer_email (VARCHAR(50)), customer_phone (NUMBER).

- **potential error codes:**
  - -1: When the data couldn't be deleted. Occurred mostly when id could not be found in 'customers' table.
  - 1: When record is successfully deleted.

- **Description:** This procedure outputs the contact details of a customer in the **'customers'** table based on whatever inputs the user provides. It doesn't give any data if the entered customer_id  does not exist in the customers table.

- **Example of execution:**

```
--------------------
--4) SELECT
--------------------
SET SERVEROUTPUT ON;
DECLARE
err_code INT;
customer_id NUMBER;
customer_name VARCHAR(30);
customer_phone NUMBER;
customer_email VARCHAR(50);

BEGIN
    spcustomersSelect(err_code, 1002, customer_id, customer_name, customer_phone, customer_email);
    DBMS_OUTPUT.PUT_LINE('Error code: '|| err_code);
    DBMS_OUTPUT.PUT_LINE('Customer number: '|| customer_id);
    DBMS_OUTPUT.PUT_LINE('Customer name: '|| customer_name);
    DBMS_OUTPUT.PUT_LINE('Customer phone: '|| customer_phone);
    DBMS_OUTPUT.PUT_LINE('Customer email: '|| customer_email);
END;
```

- spLogSelect
    - **REQUIRED INPUT PARAMETERS**: This procedure takes in log_id (INT) as an input, and gives us login information of customers.
    - **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is negative if there some error occur and positive if none. It, shows us the output with userid (int) , logLocation (String), logSuccess(int), logTime(string).

    - **potential error codes:**
        - -1: When data could not be entered which matches the existing ID's.
        - 1: When record is successfully Read.

    - **Description:** This procedure outputs the login data of the customer along with its Location, success, date. This happens only when the Login_id entered by the User is correct and matches the database or else this procedure will give error.
    - **Example of execution:**

```
create or replace PROCEDURE spLogsSelect(err_code out integer,
                                         log_id IN LOGS.LOGID%TYPE,
                                         log_userid out LOGS.USERID%TYPE,
                                         log_location out LOGS.LOGLOCATION%TYPE,
                                         log_success out LOGS.LOGSUCCESS%TYPE,
                                         log_time out LOGS.LOGTIME%TYPE)
AS
BEGIN
    SELECT USERID,LOGLOCATION,LOGSUCCESS,LOGTIME
    INTO log_userid,log_location,log_success,log_time
    from LOGS
    where LOGID = log_id;

    err_code := sql%rowcount;

    EXCEPTION
        WHEN NO_DATA_FOUND
            THEN err_code := -2; dbms_output.put_line('Data not entered');
        WHEN OTHERS THEN
        err_code:=-1; DBMS_OUTPUT.PUT_LINE('INVALID LOG ID');
END spLogsSelect;
```

- spLogDelete
    - **REQUIRED INPUT PARAMETERS**: : This procedure takes in log_id (INT) as an input, and deletes the data regarding it.
    - **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is negative if there some error occur and positive if none. It confirm with meaningful message showing that Delete was Successful.
    - **potential error codes:**
        - -1: When data could not be entered which matches the existing ID's.
        - 1: When record is successfully deleted from database.
    - **Description:** This procedure delete's the login information of the customers. To do it the user need to enter their valid Login ID which has been already entered into the database.
    - **Example of execution:**

```
create or replace PROCEDURE spLogsDELETE(err_code out integer,
                                         log_id IN LOGS.LOGID%TYPE)
AS
BEGIN
    DELETE FROM LOGS
    WHERE LOGID = log_id;
    err_code := sql%rowcount;
    COMMIT;

    EXCEPTION
        WHEN NO_DATA_FOUND
            THEN err_code := -2; dbms_output.put_line('Log ID does not exist');
        WHEN OTHERS THEN
        err_code:=-1; DBMS_OUTPUT.PUT_LINE('LOG ID ALREADY EXIST/CANNOT INSERT');
END spLogsDELETE;
```

- spLogInsert
    - **REQUIRED INPUT PARAMETERS**: This procedure takes in log_id (INT), userId(INT), logLocation(String), logSuccess(int), LogTime(date) as an input,
    - **EXPECTED OUTPUT:** The output we get from the procedure is the err_count which is negative if there some error occur and positive if none. It, shows us the output with meaningful message displaying Inserting Successful.

    - **potential error codes:**
        - -1: When data could not be entered which matches the existing ID's.
        - 1: When record is successfully Insert.

    - **Description:** This procedure make a new row as it takes user input as LoginID,userID,location,success,logTime.
    - **Example of execution:**

```
create or replace PROCEDURE spLogsInsert(err_code out integer,
                                         log_id IN LOGS.LOGID%TYPE,
                                         log_userid IN LOGS.USERID%TYPE,
                                         log_location IN LOGS.LOGLOCATION%TYPE,
                                         log_success IN LOGS.LOGSUCCESS%TYPE,
                                         log_time IN LOGS.LOGTIME%TYPE)
AS
BEGIN
    INSERT INTO LOGS (LOGID,USERID,LOGLOCATION,LOGSUCCESS,LOGTIME)
    VALUES(log_id,log_userid,log_location,log_success,log_time);

    err_code := sql%rowcount;

    EXCEPTION
        WHEN NO_DATA_FOUND
            THEN err_code := -2; dbms_output.put_line('Log ID does not exist');
        WHEN OTHERS THEN
        err_code:=-1; DBMS_OUTPUT.PUT_LINE('LOG ID ALREADY EXIST/CANNOT INSERT');
END spLogsInsert;
```

- spLogUpdate
    - **REQUIRED INPUT PARAMETERS**: This procedure takes LogId(int), LogLocation(String),LogSuccess(int)
    - **EXPECTED OUTPUT:** The output we get from the procedure is the err_count The output we get from the procedure is the err_count which is negative if there some error occur and positive if none. It, shows us the output with meaningful message displaying Delete Successful.
    - 
    - **potential error codes:**
    - -1: When data could not be entered which matches the existing ID's.
    - 1: When record is successfully Delete.
    - **Description:** This procedure Updates the login information of the customers. To do it the user need to enter their valid Login ID which has been already entered into the database.
    - **Example of execution:**

```
create or replace PROCEDURE spLogsUPDATE(err_code out integer,
                                         log_id IN LOGS.LOGID%TYPE,
                                         log_location IN LOGS.LOGLOCATION%TYPE,
                                         log_success IN LOGS.LOGSUCCESS%TYPE)
AS
BEGIN
    UPDATE LOGS SET LOGLOCATION=log_location,LOGSUCCESS=log_success
    WHERE LOGID = log_id;
    err_code := sql%rowcount;
    COMMIT;

    EXCEPTION
        WHEN NO_DATA_FOUND
            THEN err_code := -2; dbms_output.put_line('Data not entered');
        WHEN OTHERS THEN
        err_code:=-1; DBMS_OUTPUT.PUT_LINE('LOG ID ALREADY EXIST/CANNOT INSERT');
END spLogsUPDATE;
```