UNIVERSIDAD POLITÉCNICA DE VICTORIA

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

Administración de Base de Datos

Proyecto Final

Integrantes:
Joshua Nathaniel Arrazola Elizondo
José Guillermo Ibarra Monrreal

M.I. Sheyla Maleny Silva González

21 de Agosto de 2024 Ciudad Victoria, Tamaulipas.

Documentación: Sistema gestor de viajeros

El primer paso para empezar el proyecto, como es lógico, fue leer las instrucciones, donde nos dimos cuenta que durante el proyecto debiamos trabajar en dos paquetes, `traveler_assistance_package` y `traveler_admin_package`.

Paquete **TRAVELER ASSISTANCE**.

Para empezar, creamos el *package header* del paquete **traveler_assistance_package**:

```
-- PACKAGE HEADER
CREATE OR REPLACE PACKAGE traveler_assistance_package
```

Así como también el *package body*:

```
-- PACKAGE BODY
CREATE OR REPLACE PACKAGE BODY traveler_assistance_package
```

El paquete `traveler_assistance_package` debe contener 6 procedimientos:

1. country_demographics
2. find_region_and_currency
3. countries_in_same_region
4. print_region_array
5. country_languages
6. print_language_array

Donde a continuación se documenta cada uno:

1. **country_demographics**

Se nos pide programar un *procedimiento* llamado **country_demographics**, que nos permita mostrar información específica acerca de un país. Además, en la siguiente instrucción, se nos pide pasar `COUNTRY_NAME` como parámetro de entrada.

El primer paso para empezar a programar cada procedimiento dentro de un paquete es colocarlo en el *package header*, por lo cual colocamos el nombre del procedimiento junto con el parámetro que recibe:

```
PROCEDURE country_demographics(v_country_name VARCHAR2); -- Procedimiento 1
// Esto va en el PACKAGE HEADER
```

Posterior a ello, empezamos a escribir el cuerpo del procedimiento dentro del *package body*:

```
PROCEDURE country_demographics(v_country_name VARCHAR2) IS
```

> Donde atendemos la instrucción que nos pide que enviemos como parámetro el `country_name`.

Después, la instrucción nos pide que mostremos las columnas `COUNTRY_NAME`, `LOCATION`, `CAPITOL`, `POPULATION`, `AIRPORTS` Y `CLIMATE`. Sin embargo, la instrucción **no** especifica directamente de cuál tabla es de la que tenemos que obtener todos estos datos, por lo cual el uso de la sentencia **DESCRIBE** nos puede ser útil para estos casos:

```
SQL>   DESCRIBE WF_countries;
Name    Null?   Type
COUNTRY_ID  NOT NULL    NUMBER(4)
REGION_ID   NOT NULL    NUMBER(3)
COUNTRY_NAME    NOT NULL VARCHAR2(70) -- COUNTRY_NAME
COUNTRY_TRANSLATED_NAME     VARCHAR2(40)
LOCATION        VARCHAR2(90) -- LOCATION
CAPITOL     VARCHAR2(50) -- CAPITOL
AREA        NUMBER(15)
COASTLINE       NUMBER(8)
LOWEST_ELEVATION        NUMBER(6)
LOWEST_ELEV_NAME        VARCHAR2(70)
HIGHEST_ELEVATION       NUMBER(6)
HIGHEST_ELEV_NAME       VARCHAR2(50)
DATE_OF_INDEPENDENCE        VARCHAR2(30)
NATIONAL_HOLIDAY_NAME       VARCHAR2(200)
NATIONAL_HOLIDAY_DATE       VARCHAR2(30)
POPULATION      NUMBER(12) -- POPULATION
POPULATION_GROWTH_RATE      VARCHAR2(10)
LIFE_EXPECT_AT_BIRTH        NUMBER(6,2)
MEDIAN_AGE      NUMBER(6,2)
AIRPORTS        NUMBER(6) -- AIRPORTS
CLIMATE     VARCHAR2(1000) -- CLIMATE
FIPS_ID     CHAR(2)
INTERNET_EXTENSION      VARCHAR2(3)
FLAG        BLOB
CURRENCY_CODE   NOT NULL    VARCHAR2(7)
```

> Donde podemos observar que dichas columnas se encuentran en la tabla `WF_COUNTRIES`.

Posteriormente, la instrucción nos pide algo bien específico: "Usa una estructura de registro definido por el Usuario para la cláusula INTO de tu sentencia Select", donde la palabra clave es "*estructura de registro*", que nos insinúa que debemos utilizar alguna estructura de datos capaz de alojar toda la información obtenida. Dada la situación, el uso de un `RECORD` es de gran utilidad.

Teniendo en cuenta los datos que debemos obtener, creamor un `RECORD` que sea capaz de almacenar todo ello, esto primeramente debe estar definido en el *package header*:

```
```

```
TYPE country_record IS RECORD (
    country_name WF_COUNTRIES.COUNTRY_NAME%TYPE,
    location WF_COUNTRIES.LOCATION%TYPE,
    capitol WF_COUNTRIES.CAPITOL%TYPE,
    population WF_COUNTRIES.POPULATION%TYPE,
    airports WF_COUNTRIES.AIRPORTS%TYPE,
    climate WF_COUNTRIES.CLIMATE%TYPE
);
```

Posteriormente, declaramos una variable de dicho tipo en el *package body*, más específicamente en la función que estamos programando:

```
v_country_information country_record;
```

Una vez con todas las variables definidas, solamente queda ejecutar la sentencia **SELECT** para recuperar los datos:

```
SELECT COUNTRY_NAME, LOCATION, CAPITOL, POPULATION, AIRPORTS, CLIMATE
INTO v_country_information
FROM WF_COUNTRIES
WHERE UPPER(COUNTRY_NAME) = UPPER(v_country_name); -- Uso del parámetro
v_country_name
```

> La palabra clave `INTO` guarda los datos obtenidos dentro de la variable `v_country_information`.

Una vez con los datos almacenados, los imprimimos por consola con la función **PUT_LINE** del paquete **DBMS_OUTPUT**:

```
DBMS_OUTPUT.PUT_LINE('Country Name: ' ||
v_country_information.country_name);
DBMS_OUTPUT.PUT_LINE('Location: ' || v_country_information.location);
DBMS_OUTPUT.PUT_LINE('Capitol: ' || v_country_information.capitol);
DBMS_OUTPUT.PUT_LINE('Population: ' ||
TO_CHAR(v_country_information.population));
DBMS_OUTPUT.PUT_LINE('Airports: ' ||
TO_CHAR(v_country_information.airports));
DBMS_OUTPUT.PUT_LINE('Climate: ' || v_country_information.climate);
```

Finalmente, la instrucción nos pide una última cosa, que es lanzar una excepción si el país indicado **NO** existe. Para lograr este objetivo, recurrimos a la *excepción implícita* `NO_DATA_FOUND` para ello, y utilizamos la función `RAISE_APPLICATION_ERROR` para lanzar el error, donde estamos obviando que esto debe ser programado dentro del bloque **EXCEPTION**.

```

```

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'No se encontró información para el
país: ' || v_country_name);
```

2. **find_region_and_currency**

Está instrucción nos pide encontrar un procedimiento llamado `find_region_and_currency`, que lea y regrese la **moneda** y **región** en la cual un país está localizado. Además, también nos pide enviar `COUNTRY_NAME`, o el nombre del país.

Nuevamente, como primer paso, definimos la función en el *package header*:

```
PROCEDURE find_region_and_currency(COUNTRY_NAME IN VARCHAR2, country OUT
country_type); -- Procedimiento 2
```

Así como también en el *package body*:

```
 PROCEDURE find_region_and_currency(COUNTRY_NAME IN VARCHAR2)
```

También nos pide utilizar un registro **definido por el usuario** como parámetro de salida, que devuelva la información del **país**, **región** y **moneda**. Está última instrucción debemos tomarla por partes.

Primero, al igual que en el procedimiento anterior, se entiende que por **registro definido por el usuario**, debemos definir un *RECORD* que sea capaz de almacenar los datos de interés, para ello, definimos dicha estructura en el *package header*:

```
TYPE country_info IS RECORD(
    country_name -- ? Aún no conocemos el tipo, ni la ubicación de estas
columnas
    region -- ? Aún no conocemos el tipo, ni la ubicación de estas columnas
    currency -- ? Aún no conocemos el tipo, ni la ubicación de estas
columnas
);
```

Y modificamos el procedimiento para que regrese una variable de dicho tipo. Sin embargo, es conveniente recordar que un *procedimiento* **no** puede devolver valores, sin embargo, si podemos enviarlo como **OUT**, con el fin de que el valor pueda ser modificado dentro procedimiento.

```
PROCEDURE find_region_and_currency(COUNTRY_NAME IN VARCHAR2, country OUT
country_info)
```

Ahora viene lo complicado, ejecutar la sentencia **SELECT** completa. Nuevamente no nos dicen que tabla/tablas utilizar directamente, sin embargo, podemos hacer `DESCRIBE`:

```
SQL>   DESCRIBE WF_COUNTRIES;
Name    Null?   Type
COUNTRY_ID  NOT NULL    NUMBER(4)
REGION_ID   NOT NULL    NUMBER(3) -- IDENTIFICADOR DE CADA REGIÓN
COUNTRY_NAME    NOT NULL    VARCHAR2(70)
COUNTRY_TRANSLATED_NAME     VARCHAR2(40)
LOCATION        VARCHAR2(90)
CAPITOL     VARCHAR2(50)
AREA        NUMBER(15)
COASTLINE       NUMBER(8)
LOWEST_ELEVATION        NUMBER(6)
LOWEST_ELEV_NAME        VARCHAR2(70)
HIGHEST_ELEVATION       NUMBER(6)
HIGHEST_ELEV_NAME       VARCHAR2(50)
DATE_OF_INDEPENDENCE        VARCHAR2(30)
NATIONAL_HOLIDAY_NAME       VARCHAR2(200)
NATIONAL_HOLIDAY_DATE       VARCHAR2(30)
POPULATION      NUMBER(12)
POPULATION_GROWTH_RATE      VARCHAR2(10)
LIFE_EXPECT_AT_BIRTH        NUMBER(6,2)
MEDIAN_AGE      NUMBER(6,2)
AIRPORTS        NUMBER(6)
CLIMATE     VARCHAR2(1000)
FIPS_ID     CHAR(2)
INTERNET_EXTENSION      VARCHAR2(3)
FLAG        BLOB
CURRENCY_CODE   NOT NULL    VARCHAR2(7) -- IDENTIFICADOR DE MONEDA

SQL>   DESCRIBE WF_WORLD_REGIONS;
Name    Null?   Type
REGION_ID   NOT NULL    NUMBER(3) -- IDENTIFICADOR DE REGIÓN
REGION_NAME NOT NULL    VARCHAR2(35)

SQL>   DESCRIBE WF_CURRENCIES;
Name    Null?   Type
CURRENCY_CODE   NOT NULL    VARCHAR2(7) -- IDENTIFICADOR DE MONEDA
CURRENCY_NAME   NOT NULL    VARCHAR2(40)
COMMENTS        VARCHAR2(150)
```

Donde, primero, para obtener el `CURRENCY_NAME`, notamos que tenemos que realizar el **JOIN** de la columna `WF_COUNTRIES.CURRENCY_CODE` y `WF_CURRENCIES.CURRENCY_CODE`. Por otro lado, para obtener el nombre de la región, podemos notar que podemos obtener el `REGION_NAME` del **JOIN** de la columna `WF_COUNTRIES.REGION_ID` y `WF_WORLD_REGIONS.REGION_ID`.

Una vez identificadas las columnas, podemos también completar la definición del **RECORD**:

```
TYPE country_info IS RECORD(
    country_name WF_COUNTRIES.country_name%TYPE,
    region WF_WORLD_REGIONS.REGION_NAME%TYPE,
    currency WF_CURRENCIES.currency_name%TYPE
);
```

Después, una vez con las columnas ubicadas, procedimos a realizar la sentencia `SELECT` y a guardar el resultado en el *RECORD*:

```
SELECT ctr.country_name, wr.region_name, cur.currency_name INTO country
FROM WF_COUNTRIES ctr, WF_WORLD_REGIONS wr, WF_CURRENCIES cur
WHERE LOWER(ctr.COUNTRY_NAME) = LOWER(COUNTRY_NAME) AND
ctr.REGION_ID = wr.REGION_ID AND
ctr.CURRENCY_CODE = cur.CURRENCY_CODE;
```

Finalmente, mostramos una excepción si no se encontraron resultados:

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'No se encontraron ciudades para '
|| COUNTRY_NAME);
```

3. **countries_in_same_region**

El procedimiento `countries_in_same_region` se encarga de **leer** y **devolver** todos los países que pertenecen a una misma región específica. Este procedimiento recibe `REGION_NAME` como un parámetro de entrada y utiliza un arreglo asociativo de registros (**INDEX BY**) como parámetro de salida, el cual devolverá los campos `REGION_NAME`, `COUNTRY_NAME`, y `CURRENCY_NAME` para todos los países que coincidan con la región solicitada.

Para comenzar, *definimos* el procedimiento en el *package header* con la siguiente declaración:

```
PROCEDURE countries_in_same_region(v_region_name IN VARCHAR2, countries OUT
countries_type);
```

Dentro del cuerpo del procedimiento, se define una variable `v_country` de tipo `country_type`, que almacenará temporalmente los datos de cada país mientras se procesan los resultados de la consulta. También se inicializa una variable `i` de tipo **PLS_INTEGER** con el valor **1**, que funcionará como índice para el arreglo asociativo `countries`:

```
v_country country_type;
i PLS_INTEGER := 1;
```

El procedimiento utiliza un **FOR LOOP** para iterar sobre los resultados de una consulta SQL que selecciona `country_name`, `region_name`, y `currency_name` de las tablas `WF_COUNTRIES`, `WF_WORLD_REGIONS`, y `WF_CURRENCIES`. Los datos se obtienen mediante un **JOIN** sobre las columnas `region_id` y `currency_code`, asegurando que se seleccionen *únicamente* los registros que coincidan con el nombre de la región proporcionado en el **argumento**. La comparación se realiza de manera ***insensible*** a mayúsculas o minúsculas utilizando la función `LOWER`:

```
FOR r IN (SELECT c.country_name, r.region_name, cu.currency_name
          FROM WF_COUNTRIES c
          JOIN WF_WORLD_REGIONS r ON c.region_id = r.region_id
          JOIN WF_CURRENCIES cu ON c.currency_code = cu.currency_code
          WHERE LOWER(r.region_name) = LOWER(v_region_name)) LOOP
```

Dentro del bucle, se asignan los valores recuperados a la variable `v_country`:

```
v_country.country_name := r.country_name;
v_country.region := r.region_name;
v_country.currency := r.currency_name;
```

Estos valores se almacenan luego en el arreglo countries utilizando el índice i, que se incrementa con cada iteración para asegurar que cada registro se almacene en la posición correcta dentro del arreglo:

```
countries(i) := v_country;
i := i + 1;
```

Una vez que el bucle ha procesado todos los registros, se verifica si `i` sigue siendo igual a **1**. Si este es el caso, significa que **no se encontraron registros** que coincidan con el nombre de la región, por lo que se lanza la excepción `NO_DATA_FOUND` para indicar que *no hay resultados* disponibles para la región solicitada:

```
IF i = 1 THEN
    RAISE NO_DATA_FOUND;
END IF;
```

Finalmente, se maneja la excepción `NO_DATA_FOUND` utilizando `RAISE_APPLICATION_ERROR` para generar un mensaje de error que notifique al usuario que **no** se encontraron ciudades para la región especificada:

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'No se encontraron ciudades para '
|| v_region_name);
```

4. **print_region_array**

El procedimiento `print_region_array` está diseñado para **mostrar el contenido de un arreglo de registros** que se pasa como *parámetro de entrada*. Este procedimiento es especialmente útil para *visualizar* la información previamente almacenada en un arreglo asociativo de registros, como el que se genera en el procedimiento `countries_in_same_region`.

Para comenzar, definimos el procedimiento en el *package header* del paquete `traveler_assistance_package` con la siguiente declaración:

```
PROCEDURE print_region_array(countries countries_type); -- Procedimiento 4
```

El propósito de este procedimiento es iterar a través del arreglo asociativo countries, el cual contiene registros que incluyen los campos `country_name`, `region`, y `currency`. El procedimiento *recorrerá* cada uno de estos registros y **mostrará** sus contenidos utilizando la función `DBMS_OUTPUT.PUT_LINE`, que permite imprimir el mensaje por consola.

Dentro del cuerpo del procedimiento, se utiliza un **FOR LOOP** para recorrer el arreglo countries. Este bucle se extiende desde el *primer índice* (`countries.FIRST`) hasta el *último* (`countries.LAST`), asegurando que todos los registros almacenados en el arreglo se procesen:

```
FOR i IN countries.FIRST .. countries.LAST LOOP
```

Dentro del bucle, para cada índice `i`, se imprimen los valores de `country_name`, `region`, y `currency`, correspondientes al registro almacenado en esa posición del arreglo. La impresión se realiza en un formato **claro** y **legible** para el usuario:

```
DBMS_OUTPUT.PUT_LINE('Country Name : ' || countries(i).country_name);
DBMS_OUTPUT.PUT_LINE('Region       : ' || countries(i).region);
DBMS_OUTPUT.PUT_LINE('Currency     : ' || countries(i).currency);
```

Además, para **mejorar la legibilidad** de la salida, se incluye una línea de separación entre los registros utilizando una serie de guiones:

```
DBMS_OUTPUT.PUT_LINE('-----------------------------------');
```

5. **country_languages**

El procedimiento `country_languages` tiene como objetivo **leer** y **devolver** todos los idiomas hablados en un *país determinado*, así como identificar **cuál** de esos idiomas es el idioma oficial. Este *procedimiento* recibe `COUNTRY_NAME` como un *parámetro de entrada* y utiliza un arreglo asociativo de registros como

parámetro de salida. El arreglo devuelto contiene los campos `COUNTRY_NAME`, `LANGUAGE_NAME`, y `OFFICIAL` para cada idioma asociado con el país especificado.

El procedimiento se define en el `package header` del paquete `traveler_assistance_package` con la siguiente declaración:

```
PROCEDURE country_languages(v_country_name IN VARCHAR2, country_lang OUT
country_languages_type); -- Procedimiento 5
```

Dentro del cuerpo del procedimiento, se inicia declarando una variable `v_language` de tipo `country_language_type`, que se utilizará para **almacenar temporalmente** los datos de cada idioma recuperado. También se inicializa una variable `i` de tipo **PLS_INTEGER** con el valor 1, que funcionará como índice para el arreglo asociativo `country_lang`.

El procedimiento utiliza un FOR LOOP que itera sobre los resultados de una consulta SQL. Esta consulta selecciona los campos `country_name`, `language_name`, y `official` de las tablas `WF_COUNTRIES`, `WF_LANGUAGES`, y `WF_SPOKEN_LANGUAGES`. Para obtener estos datos, se realiza un **JOIN** entre las tablas utilizando las columnas `country_id` y `language_id`, lo que asegura que se recuperen *únicamente* los registros que coincidan con el nombre del país proporcionado. La comparación del nombre del país se realiza de manera **insensible** a mayúsculas o minúsculas utilizando la función **UPPER**:

```
FOR r IN (SELECT c.country_name, l.language_name, sl.official
          FROM WF_COUNTRIES c
          JOIN WF_SPOKEN_LANGUAGES sl ON c.country_id = sl.country_id
          JOIN WF_LANGUAGES l ON sl.language_id = l.language_id
          WHERE UPPER(c.country_name) = UPPER(v_country_name)) LOOP
```

Dentro del bucle, los valores recuperados se asignan a la variable `v_language`, que almacena *temporalmente* el nombre del país, el nombre del idioma y si es oficial o no. Estos valores se asignan de la siguiente manera:

```
v_language.country_name := r.country_name;
v_language.language_name := r.language_name;
v_language.official_language := r.official;
```

Una vez que los valores se han asignado a `v_language`, se almacenan en el arreglo `country_lang` utilizando el índice `i`, que se incrementa en cada iteración para asegurar que cada registro se almacene en la **posición correcta** dentro del arreglo:

```
country_lang(i) := v_language;
i := i + 1;
```

Después de que el bucle ha procesado **todos** los registros, se verifica si `i` sigue siendo igual a **1**. Si este es el caso, significa que no se encontraron registros que coincidan con el nombre del país, por lo que se *lanza* la excepción `NO_DATA_FOUND` para indicar que no hay resultados disponibles para el país solicitado:

```
IF i = 1 THEN
    RAISE NO_DATA_FOUND;
END IF;
```

Finalmente, se maneja la excepción `NO_DATA_FOUND` utilizando `RAISE_APPLICATION_ERROR` para generar un mensaje de error que notifique al usuario que no se encontraron idiomas para el país especificado:

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'No se encontraron idiomas para el
país: ' || v_country_name);
```

6. **print_language_array**

El procedimiento `print_language_array` está diseñado para **mostrar** el contenido de un *arreglo asociativo* de registros que se pasa como *parámetro de entrada*. Este procedimiento es especialmente **útil** para **visualizar** la información previamente almacenada en el arreglo generado por el procedimiento `country_languages`.

Para comenzar, se define el procedimiento en el *package header* del paquete `traveler_assistance_package` con la siguiente declaración:

```
PROCEDURE print_language_array(country_language country_languages_type); --
Procedimiento 6
```

El propósito de este procedimiento es iterar a través del arreglo asociativo `country_language`, el cual contiene registros que incluyen los campos `country_name`, `language_name`, y `official_language`. El procedimiento recorre cada uno de estos **registros** y **muestra** sus contenidos utilizando la función `DBMS_OUTPUT.PUT_LINE`, que permite imprimir mensajes en consola.

Dentro del cuerpo del procedimiento, se utiliza un **FOR LOOP** para recorrer el arreglo `country_language`. Este bucle se extiende desde el **primer índice** (`country_language.FIRST`) hasta el **último** (`country_language.LAST`), asegurando que todos los registros almacenados en el arreglo se procesen:

```
FOR i IN country_language.FIRST .. country_language.LAST LOOP
```

Durante cada iteración del bucle, se imprimen los valores de `country_name`, `language_name`, y `official_language` correspondientes al registro almacenado en esa posición del arreglo. La impresión se

realiza en un formato **claro** y **legible** para el usuario:

```
DBMS_OUTPUT.PUT_LINE('Country Name      : ' ||
country_language(i).country_name);
DBMS_OUTPUT.PUT_LINE('Language Name     : ' ||
country_language(i).language_name);
DBMS_OUTPUT.PUT_LINE('Official Language : ' ||
country_language(i).official_language);
```

Para mejorar la legibilidad de la salida, se incluye una **línea de separación** entre los registros utilizando una serie de guiones:

```
DBMS_OUTPUT.PUT_LINE('-------------------------');
```

Pruebas de funcionamiento

Prueba 1:

```
SELECT COUNTRY_NAME, LOCATION, CAPITOL, POPULATION, AIRPORTS, CLIMATE FROM
WF_COUNTRIES WHERE COUNTRY_NAME = 'Mongolia';

BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.COUNTRY_DEMOGRAPHICS('Mongolia');
END;
```

Para probar el procedimiento `country_demographics` del paquete `traveler_assistance_package`, comenzamos realizando una **consulta directa** a la tabla `WF_COUNTRIES` para obtener los valores de `COUNTRY_NAME`, `LOCATION`, `CAPITOL`, `POPULATION`, `AIRPORTS`, y `CLIMATE` para el país **"Mongolia"**. Esta consulta nos sirve como referencia para validar la salida generada por el procedimiento. La consulta que utilizamos es la siguiente:

```
SELECT COUNTRY_NAME, LOCATION, CAPITOL, POPULATION, AIRPORTS, CLIMATE
FROM WF_COUNTRIES
WHERE COUNTRY_NAME = 'Mongolia';
```

Después de obtener estos resultados, ejecutamos el procedimiento `country_demographics` pasando **"Mongolia"** como parámetro. La ejecución del procedimiento se realiza con el siguiente bloque PL/SQL:

```
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.COUNTRY_DEMOGRAPHICS('Mongolia');
END;
```

Al ejecutar este bloque, el procedimiento debería mostrar en la consola los valores correspondientes a `COUNTRY_NAME`, `LOCATION`, `CAPITOL`, `POPULATION`, `AIRPORTS`, y `CLIMATE` para **"Mongolia"**. Los resultados que el procedimiento imprime **deben coincidir** con los obtenidos en la consulta directa que realizamos *previamente*. Si la información mostrada por el procedimiento es la misma que la obtenida a través de la consulta SQL, podemos **concluir** que el procedimiento `country_demographics` funciona **correctamente**.

Prueba 2:

```
SELECT c.country_name, wr.region_name, cu.currency_name
        FROM WF_COUNTRIES c, WF_WORLD_REGIONS wr, WF_CURRENCIES cu
        WHERE LOWER(c.COUNTRY_NAME) = LOWER('Mongolia') AND
                c.REGION_ID = wr.REGION_ID AND
                c.CURRENCY_CODE = cu.CURRENCY_CODE;

DECLARE
    country_name VARCHAR2(50) := 'Mongolia';
    country TRAVELER_ASSISTANCE_PACKAGE.country_type;
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.FIND_REGION_AND_CURRENCY(country_name,
country);
    DBMS_OUTPUT.PUT_LINE(country.country_name || ', ' || country.region ||
', ' || country.currency);
END;
```

Para probar el procedimiento `find_region_and_currency` del paquete `traveler_assistance_package`, primero realizamos una consulta directa a las tablas `WF_COUNTRIES`, `WF_WORLD_REGIONS`, y `WF_CURRENCIES` para obtener los valores de `COUNTRY_NAME`, `REGION_NAME`, y `CURRENCY_NAME` correspondientes al país **"Mongolia"**. Esta consulta nos permite verificar que los datos obtenidos por el procedimiento son correctos. La consulta utilizada es la siguiente:

```
SELECT c.country_name, wr.region_name, cu.currency_name
FROM WF_COUNTRIES c, WF_WORLD_REGIONS wr, WF_CURRENCIES cu
WHERE LOWER(c.COUNTRY_NAME) = LOWER('Mongolia')
AND c.REGION_ID = wr.REGION_ID
AND c.CURRENCY_CODE = cu.CURRENCY_CODE;
```

Tras obtener los resultados de esta consulta, procedemos a **ejecutar** el procedimiento `find_region_and_currency`, pasando **"Mongolia"** como *parámetro de entrada*. El procedimiento recupera los datos correspondientes al **nombre** del país, su **región** y su **moneda**, los cuales se almacenan en una variable de tipo `country_type`. Finalmente, estos datos son *impresos* en la consola utilizando `DBMS_OUTPUT.PUT_LINE`. El código para ejecutar esta prueba es el siguiente:

```
DECLARE
    country_name VARCHAR2(50) := 'Mongolia';
    country TRAVELER_ASSISTANCE_PACKAGE.country_type;
```

```
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.FIND_REGION_AND_CURRENCY(country_name,
country);
    DBMS_OUTPUT.PUT_LINE(country.country_name || ', ' || country.region ||
', ' || country.currency);
END;
```

El procedimiento **debería** imprimir en la consola los valores de `COUNTRY_NAME`, `REGION_NAME`, y `CURRENCY_NAME` para **"Mongolia"**, que deben coincidir con los resultados obtenidos de la consulta directa previa. Si los datos mostrados por el procedimiento son **consistentes** con los obtenidos en la consulta, podemos concluir que el procedimiento `find_region_and_currency` está funcionando **correctamente**.

Prueba 3:

```
SELECT c.country_name, r.region_name, cu.currency_name
FROM WF_COUNTRIES c
JOIN WF_WORLD_REGIONS r ON c.region_id = r.region_id
JOIN WF_CURRENCIES cu ON c.currency_code = cu.currency_code
WHERE LOWER(r.region_name) = LOWER('Central America');

DECLARE
    region_name VARCHAR2(50) := 'Central America';
    countries TRAVELER_ASSISTANCE_PACKAGE.countries_type;
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.COUNTRIES_IN_SAME_REGION(region_name,
countries);
    TRAVELER_ASSISTANCE_PACKAGE.PRINT_REGION_ARRAY(countries);
END;
```

Para probar el procedimiento `countries_in_same_region` del paquete `traveler_assistance_package`, comenzamos realizando una consulta directa a las tablas `WF_COUNTRIES`, `WF_WORLD_REGIONS`, y `WF_CURRENCIES` para obtener los valores de `COUNTRY_NAME`, `REGION_NAME`, y `CURRENCY_NAME` para todos los países que pertenecen a la región "Central America". Esta consulta sirve como referencia para comparar y validar los resultados que serán generados por el procedimiento. La consulta SQL utilizada es la siguiente:

```
SELECT c.country_name, r.region_name, cu.currency_name
FROM WF_COUNTRIES c
JOIN WF_WORLD_REGIONS r ON c.region_id = r.region_id
JOIN WF_CURRENCIES cu ON c.currency_code = cu.currency_code
WHERE LOWER(r.region_name) = LOWER('Central America');
```

Una vez que hemos obtenido los resultados de esta consulta, procedemos a ejecutar el procedimiento `countries_in_same_region`, pasando **"Central America"** como *parámetro de entrada*. Este procedimiento recupera la lista de países, junto con sus respectivas regiones y monedas, y almacena esta información en un *arreglo asociativo* de tipo `countries_type`. Luego, utilizamos el procedimiento

`print_region_array` para imprimir el contenido de este arreglo en la consola. El bloque PL/SQL para realizar esta prueba es el siguiente:

```
DECLARE
    region_name VARCHAR2(50) := 'Central America';
    countries TRAVELER_ASSISTANCE_PACKAGE.countries_type;
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.COUNTRIES_IN_SAME_REGION(region_name,
countries);
    TRAVELER_ASSISTANCE_PACKAGE.PRINT_REGION_ARRAY(countries);
END;
```

Durante la ejecución de este bloque, el procedimiento `countries_in_same_region` debería recuperar todos los países asociados con la región **"Central America"** y almacenarlos en el arreglo `countries`. Luego, el procedimiento `print_region_array` imprimirá en la consola los valores de `COUNTRY_NAME`, `REGION_NAME`, y `CURRENCY_NAME` para cada país. Estos resultados deben coincidir con los que se obtuvieron en la consulta SQL directa realizada previamente, de ser así (que es así), entonces concluimos que la solución **es correcta**.

Prueba 4:

```
SELECT c.country_name, l.language_name, sl.official
FROM WF_COUNTRIES c
JOIN WF_SPOKEN_LANGUAGES sl ON c.country_id = sl.country_id
JOIN WF_LANGUAGES l ON sl.language_id = l.language_id
WHERE UPPER(c.country_name) = UPPER('Belize');

DECLARE
    country_name VARCHAR2(50) := 'Belize';
    country_langs TRAVELER_ASSISTANCE_PACKAGE.country_languages_type;
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.COUNTRY_LANGUAGES(country_name,
country_langs);
    TRAVELER_ASSISTANCE_PACKAGE.PRINT_LANGUAGE_ARRAY(country_langs);
END;
```

Para probar el procedimiento `country_languages` del paquete `traveler_assistance_package`, comenzamos realizando una **consulta directa** a las tablas `WF_COUNTRIES`, `WF_SPOKEN_LANGUAGES`, y `WF_LANGUAGES` para obtener los valores de `COUNTRY_NAME`, `LANGUAGE_NAME`, y `OFFICIAL` para todos los idiomas hablados en el país **"Belize"**. Esta consulta nos proporciona una referencia para validar los resultados que serán generados por el procedimiento. La consulta SQL utilizada es la siguiente:

```
SELECT c.country_name, l.language_name, sl.official
FROM WF_COUNTRIES c
JOIN WF_SPOKEN_LANGUAGES sl ON c.country_id = sl.country_id
```

```
JOIN WF_LANGUAGES l ON sl.language_id = l.language_id
WHERE UPPER(c.country_name) = UPPER('Belize');
```

Una vez que hemos obtenido los resultados de esta consulta, procedemos a ejecutar el procedimiento `country_languages`, pasando **"Belize"** como *parámetro de entrada*. Este procedimiento **recupera la lista** de idiomas hablados en el país, junto con una indicación de si cada idioma **es oficial o no**, y **almacena esta información en un arreglo asociativo** de tipo `country_languages_type`. Luego, utilizamos el procedimiento `print_language_array` para imprimir el contenido de este arreglo en la consola. El bloque PL/SQL para realizar esta prueba es el siguiente:

```
DECLARE
    country_name VARCHAR2(50) := 'Belize';
    country_langs TRAVELER_ASSISTANCE_PACKAGE.country_languages_type;
BEGIN
    TRAVELER_ASSISTANCE_PACKAGE.COUNTRY_LANGUAGES(country_name,
country_langs);
    TRAVELER_ASSISTANCE_PACKAGE.PRINT_LANGUAGE_ARRAY(country_langs);
END;
```

Durante la ejecución de este bloque, el procedimiento country_languages debería recuperar **todos los idiomas asociados** con **"Belize"** y almacenarlos en el arreglo `country_langs`. Luego, el procedimiento `print_language_array` *imprimirá* en la consola los valores de `COUNTRY_NAME`, `LANGUAGE_NAME`, y `OFFICIAL` para cada idioma. Estos resultados deben coincidir con los obtenidos en la consulta SQL directa realizada previamente.

Paquete **traveler_admin_package**

Para la segunda parte del proyecto, nos enfocamos en la **administración** del *sistema de viajeros* mediante la creación de un paquete denominado `traveler_admin_package`. Este paquete tiene como objetivo proporcionar **herramientas** y **procedimientos** útiles para el *mantenimiento* y *gestión* del sistema.

El paquete en cuestión cuenta con tres procedimientos:

1. display_disabled_triggers
2. all_dependent_objects
3. print_dependent_objects

1. **display_disabled_triggers**

El procedimiento `display_disabled_triggers` tiene como propósito *mostrar* una lista de todos los triggers que se encuentran **deshabilitados** en el esquema del usuario actual. Esta función es especialmente **útil** para los **administradores de bases de datos**, quienes necesitan *identificar rápidamente* los triggers que no están en funcionamiento, ya sea para revisarlos, reactivarlos, o simplemente para tener un control del estado de su esquema.

Para comenzar, definimos el procedimiento en el *package header* del paquete `traveler_admin_package` con la siguiente declaración:

```
PROCEDURE display_disabled_triggers IS
```

Dentro del *cuerpo del procedimiento*, primero se declara un cursor denominado `disabled_triggers`, que se encarga de seleccionar los nombres de todos los triggers cuyo estado es **DISABLED** en la tabla `user_triggers`. Esta tabla es parte del diccionario de datos de *Oracle* y contiene información sobre todos los triggers creados en el esquema del usuario. La consulta utilizada en el cursor es la siguiente:

```
CURSOR disabled_triggers IS
SELECT trigger_name
FROM user_triggers
WHERE status = 'DISABLED';
```

El procedimiento utiliza un **FOR LOOP** para iterar sobre cada registro devuelto por el cursor `disabled_triggers`. En cada iteración, el procedimiento recupera el nombre del trigger **deshabilitado** y lo imprime en la consola utilizando la función `DBMS_OUTPUT.PUT_LINE`. Esto permite al administrador visualizar de manera **clara** y **ordenada** todos los triggers que se encuentran *deshabilitados*:

```
FOR trigger_rec IN disabled_triggers LOOP
    DBMS_OUTPUT.PUT_LINE('Trigger deshabilitado: ' ||
trigger_rec.trigger_name);
END LOOP;
```

Este bucle asegura que **todos los triggers deshabilitados** se muestren en la salida, proporcionando al administrador una lista completa de aquellos que necesitan atención.

Finalmente, el procedimiento se completa **sin necesidad** de manejar excepciones explícitas, dado que simplemente lista los triggers deshabilitados y **no interactúa** con operaciones que podrían resultar en errores *complejos*.

2. **all_dependent_objects**

La función `all_dependent_objects` está diseñada para devolver todos los objetos que **dependen** de un objeto específico en el esquema de base de datos. Primero, definimos la función en el package header del paquete traveler_admin_package con la siguiente declaración:

```
FUNCTION all_dependent_objects(object_name VARCHAR2) RETURN obj_arr;
```

Dentro del cuerpo de la función, se *declara* un cursor llamado **dep_objects**, que selecciona los nombres (`name`), tipos (`type`), nombres referenciados (`referenced_name`), y tipos referenciados (`referenced_type`) de los objetos *dependientes* en la tabla `USER_DEPENDENCIES`. Esta tabla es parte del **diccionario de datos de Oracle** y almacena información sobre las dependencias entre objetos dentro del esquema del usuario. La consulta en el cursor es la siguiente:

```
CURSOR dep_objects IS
SELECT name, type, referenced_name, referenced_type
FROM USER_DEPENDENCIES
WHERE referenced_name = UPPER(object_name);
```

La función declara un arreglo de registros `v_objects` de tipo `obj_arr`, que se utilizará para **almacenar** los resultados obtenidos del cursor. Además, se inicializa un índice `idx` de tipo **PLS_INTEGER** con el valor **0**, que se usará para recorrer y llenar el arreglo:

```
v_objects obj_arr;  -- Arreglo para almacenar resultados
idx PLS_INTEGER := 0;    -- Índice inicial
```

La función utiliza un **FOR LOOP** para iterar sobre cada registro devuelto por el cursor `dep_objects`. En cada iteración, se incrementa el índice `idx` y se almacenan en `v_objects(idx)` los valores `name`, `type`, `referenced_name`, y `referenced_type` del **registro actual**:

```
FOR dep_rec IN dep_objects LOOP
    idx := idx + 1;
    v_objects(idx).name := dep_rec.name;
    v_objects(idx).type := dep_rec.type;
    v_objects(idx).referenced_name := dep_rec.referenced_name;
    v_objects(idx).referenced_type := dep_rec.referenced_type;
END LOOP;
```

Este bucle garantiza que todos los objetos dependientes del objeto especificado se almacenen en el arreglo `v_objects`.

Una vez que el bucle ha finalizado, la función **verifica** si el arreglo `v_objects` contiene algún elemento utilizando `v_objects.COUNT`. Si el arreglo está *vacío*, lo que indica que no se encontraron objetos dependientes, se lanza una excepción mediante `RAISE_APPLICATION_ERROR`:

```
IF v_objects.COUNT = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'No se encontraron objetos dependientes
para ' || object_name);
END IF;
```

Finalmente, si se encuentran dependencias, la función **retorna** el arreglo `v_objects` con todos los registros de los objetos dependientes almacenados en él:

```
RETURN v_objects;
```

3. **print_dependent_objects** El procedimiento `print_dependent_objects` está diseñado para **mostrar** el contenido de un arreglo de registros que contiene información sobre **objetos dependientes** dentro del esquema de base de datos. Para comenzar, definimos el procedimiento en el *package header* del paquete `traveler_admin_package` con la siguiente declaración:

```
PROCEDURE print_dependent_objects(objects IN obj_arr);
```

El procedimiento toma como *parámetro de entrada* un arreglo (`objects`) de tipo `obj_arr`, que contiene registros con información sobre los **nombres** y **tipos de objetos** dependientes, así como los nombres y tipos de los *objetos referenciados*.

Dentro del cuerpo del procedimiento, se utiliza un **FOR LOOP** para recorrer el arreglo `objects` desde el primer índice (`objects.FIRST`) hasta el último (`objects.LAST`). Este bucle asegura que **todos los registros** almacenados en el arreglo se procesen y se impriman:

```
FOR i IN objects.FIRST .. objects.LAST LOOP
```

Durante cada iteración del bucle, se **imprimen** en la consola los valores de `name`, `type`, `referenced_name`, y `referenced_type` correspondientes al registro almacenado en la posición `i` del arreglo. Estos valores se imprimen de manera **estructurada** y **legible** utilizando la función `DBMS_OUTPUT.PUT_LINE`:

```
DBMS_OUTPUT.PUT_LINE('Object Name        : ' || objects(i).name);
DBMS_OUTPUT.PUT_LINE('Object Type        : ' || objects(i).type);
DBMS_OUTPUT.PUT_LINE('Referenced Name    : ' ||
objects(i).referenced_name);
DBMS_OUTPUT.PUT_LINE('Referenced Type    : ' ||
objects(i).referenced_type);
```

Además, para mejorar la **legibilidad** y **separar** visualmente cada registro, se imprime una *línea de separación* utilizando una serie de guiones:

```
DBMS_OUTPUT.PUT_LINE('----------------------------------------');
```

Pruebas de funcionamiento

Prueba 1:

```
CREATE OR REPLACE TRIGGER trigger_prueba
BEFORE INSERT ON EMPLOYEES
FOR EACH ROW
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Trigger activado');
END;

-- Deshabilitar el trigger creado
ALTER TRIGGER trigger_prueba DISABLE;

-- Ejecutar el procedimiento para mostrar los triggers deshabilitados
BEGIN
    traveler_admin_package.display_disabled_triggers;
END;
```

Primero, se crea un trigger llamado `trigger_prueba`, que se activa antes de cualquier operación de inserción en la tabla `EMPLOYEES`. Este trigger es **simple** y su única acción es **imprimir un mensaje** en la consola cuando es activado:

```
CREATE OR REPLACE TRIGGER trigger_prueba
BEFORE INSERT ON EMPLOYEES
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Trigger activado');
END;
```

Una vez que el trigger `trigger_prueba` ha sido creado, el siguiente paso es **deshabilitarlo**. Esto se logra con la instrucción `ALTER TRIGGER`, como se muestra a continuación:

```
ALTER TRIGGER trigger_prueba DISABLE;
```

El propósito de **deshabilitar** el trigger es asegurarse de que el procedimiento `display_disabled_triggers` pueda *identificar* y **listar los triggers deshabilitados** en el esquema.

Finalmente, se ejecuta el procedimiento `display_disabled_triggers` para comprobar si el trigger deshabilitado (`trigger_prueba`) es correctamente **identificado** y **listado**. La ejecución del procedimiento se realiza con el siguiente bloque PL/SQL:

```
BEGIN
    traveler_admin_package.display_disabled_triggers;
END;
```

Prueba 2:

```
DECLARE
    dependent_objects traveler_admin_package.obj_arr;
BEGIN
    dependent_objects :=
```

```
traveler_admin_package.all_dependent_objects('EMPLOYEES');
    traveler_admin_package.print_dependent_objects(dependent_objects);
END;
```

Para validar el funcionamiento de las funciones `all_dependent_objects` y `print_dependent_objects` del paquete `traveler_admin_package`, se realiza la siguiente prueba.

Se comienza declarando una variable llamada `dependent_objects` de tipo `obj_arr`, que es el tipo de arreglo utilizado para *almacenar* la lista de **objetos dependientes**:

```
DECLARE dependent_objects traveler_admin_package.obj_arr;
```

A continuación, se llama a la función `all_dependent_objects`, pasando el nombre del objeto `EMPLOYEES` como *argumento*. Esta función retorna un **arreglo** con todos los objetos que dependen **directamente** del objeto especificado. El resultado se almacena en la variable `dependent_objects`:

```
BEGIN
    dependent_objects :=
traveler_admin_package.all_dependent_objects('EMPLOYEES');
```

Una vez que se ha obtenido la lista de objetos dependientes, se procede a **imprimir el contenido** de este arreglo utilizando el procedimiento `print_dependent_objects`. Este procedimiento recorre el arreglo `dependent_objects` y muestra en la consola los *detalles* de cada objeto dependiente, incluyendo su nombre, tipo, el nombre del objeto referenciado y el tipo de objeto referenciado:

```
    traveler_admin_package.print_dependent_objects(dependent_objects);
END;
```