Documento, segunda interacion proyecto hotel andes

Integrantes

d.zamorac 202113407

d.torresl 202110365

j.carrasquillag 202110183

c.reyv 202116752

Requerimiento número 1.

Primer requerimiento, “MOSTRAR EL DINERO RECOLECTADO POR SERVICIOS EN CADA HABITACIÓN EN EL ÚLTIMO AÑO CORRIDO” para el primer requerimiento se usó la sentencia sql.

@Query(value = "SELECT SUM(consumos.costo) FROM CONSUMOS WHERE consumos.habitacion_id=:id AND consumos.fecha_de_pago BETWEEN TRUNC(SYSDATE , 'Year') AND SYSDATE", nativeQuery = true)

Integer darIngreso(@Param("id") Integer id);

SELECT SUM(consumos.costo): Esta instrucción selecciona la suma de los valores contenidos en la columna costo dentro de la tabla CONSUMOS.

Explicación de la query MOSTRAR EL DINERO RECOLECTADO POR SERVICIOS EN CADA HABITACIÓN EN EL ÚLTIMO AÑO CORRIDO

FROM CONSUMOS: Indica la tabla de la base de datos de donde se están extrayendo los datos.

WHERE consumos.habitacion_id=:id: Filtra los registros para incluir solamente aquellos que corresponden a la habitación identificada por el parámetro :id.

AND consumos.fecha_de_pago BETWEEN TRUNC(SYSDATE , 'Year') AND SYSDATE: Define el intervalo de tiempo para el cálculo, limitando los datos a aquellos pagos efectuados desde el inicio del año en curso hasta el momento actual.

El resultado devuelto es un valor entero que representa el total de ingresos generados por los servicios utilizados en la habitación designada por el identificador id en el lapso desde el comienzo del año hasta la fecha presente.

Requerimiento número 2. MOSTRAR LOS 20 SERVICIOS MÁS POPULARES

@Query(value = "SELECT servicios_id, COUNT(*) as frecuencia FROM Consumos WHERE FECHA_DE_PAGO BETWEEN TO_DATE(:fechaInicio,'YYYY-MM-DD') AND TO_DATE(:fechaFin,'YYYY-MM-DD') GROUP BY servicios_id ORDER BY frecuencia DESC FETCH FIRST 20 ROWS ONLY", nativeQuery = true)

Collection<RespInfoConsumos> darServiciosPopulares(@Param("fechaInicio") String fechaInicio, @Param("fechaFin") String fechaFin);

ELECT servicios_id, COUNT(*) as frecuencia: Esta instrucción tiene como finalidad recopilar los identificadores únicos de los servicios y el número total de veces que cada uno ha sido contratado.

Explicación de la query

FROM Consumos: Señala la fuente de datos, en este caso, la tabla Consumos de la base de datos.

WHERE FECHA_DE_PAGO BETWEEN TO_DATE(:fechaInicio,'YYYY-MM-DD') AND TO_DATE(:fechaFin,'YYYY-MM-DD'): Restringe la selección a aquellos registros cuyas fechas de pago se encuentran dentro del rango definido por los parámetros :fechaInicio y :fechaFin.

GROUP BY servicios_id: Agrupa los resultados por el identificador de servicio, lo que permite el conteo de la frecuencia de cada servicio de forma individual.

ORDER BY frecuencia DESC: Ordena los servicios agrupados en orden descendente de frecuencia, poniendo primero los servicios más frecuentes.

FETCH FIRST 20 ROWS ONLY: Limita la salida de la consulta a los primeros veinte registros tras el ordenamiento, lo que equivale a los veinte servicios más populares.

Requerimiento número 3. MOSTRAR EL ÍNDICE DE OCUPACIÓN DE CADA UNA DE LAS HABITACIONES DEL HOTEL

@Query(value = "SELECT NVL((SELECT SUM(alojamiento.fecha_salida - alojamiento.fecha_ingreso) FROM alojamiento WHERE alojamiento.habitacion = habitaciones.id AND alojamiento.fecha_ingreso >= ADD_MONTHS(SYSDATE, -12) AND alojamiento.fecha_ingreso <= SYSDATE) / 365 * 100, 0) AS porcentaje_ocupacion FROM habitaciones WHERE habitaciones.id= :id", nativeQuery = true)

Float darOcupacionHabitacion(@Param("id") Integer id);

Explicación de la query

Se realiza una selección condicionada de la suma de intervalos de estancia, que es la diferencia entre las fechas de salida y de ingreso (alojamiento.fecha_salida - alojamiento.fecha_ingreso), de la tabla alojamiento.

La condición impuesta restringe la suma a los registros donde la habitación coincide con la especificada (alojamiento.habitacion = habitaciones.id) y donde la fecha de ingreso se encuentra dentro del último año respecto a la fecha actual del sistema (alojamiento.fecha_ingreso >= ADD_MONTHS(SYSDATE, -12) AND alojamiento.fecha_ingreso <= SYSDATE).

La suma resultante se divide por el total de días en un año (365), y el cociente se multiplica por 100 para obtener un porcentaje.

La función NVL se emplea para reemplazar cualquier resultado nulo con un cero, asegurando que se proporcione un valor por defecto en caso de que no existan datos de ocupación para la habitación en cuestión.

Este cálculo se etiqueta como porcentaje_ocupacion y se filtra para aplicarse a la habitación identificada por el parámetro suministrado :id.

El resultado es un valor de punto flotante que representa el porcentaje de días que la habitación ha estado ocupada en el transcurso del último año, proporcionando una métrica clave de rendimiento para la gestión hotelera.

Requerimiento número 4. MOSTRAR LOS SERVICIOS QUE CUMPLEN CON CIERTA CARACTERÍSTICA

```
@GetMapping("/serviciosConsumos")
    public String serviciosConsumos(Model model, String fechaInicio, String fechaFin, String tipo_servicio, String costo1, String costo2) {
        if (fechaInicio == null || fechaInicio.equals("")) {
            fechaInicio = "1000-12-12";
        }
        if (fechaFin == null || fechaFin.equals("")) {
            fechaFin = "5000-12-12";
        }
        if (costo1 == null || costo1.equals("")) {
            costo1 = "0";
        }
        if (costo2 == null || costo2.equals("")) {
            costo2 = "999999999";
        }
        model.addAttribute("filtrosServicios", servicioRepository.darServiciosCombinado(fechaInicio, fechaFin, Integer.parseInt(costo1), Integer.parseInt(costo2), tipo_servicio));


        return "serviciosFiltro";
    }
```

Explicación de la solución

El método serviciosConsumos es un controlador en un servicio web que maneja peticiones GET para la ruta /serviciosConsumos. Utiliza inyección de dependencias para incluir un modelo y recibe parámetros opcionales relacionados con fechas y costos. En caso de que los parámetros no estén presentes o estén vacíos, se asignan valores por defecto que establecen rangos amplios para asegurar la inclusión de todos los registros posibles. El método convierte

los parámetros de costo de tipo String a Integer, y luego invoca una consulta en servicioRepository pasando estos parámetros. Los resultados de la consulta se adjuntan al modelo que se pasará a la vista serviciosFiltro para su presentación al usuario. Este método sigue un patrón típico en aplicaciones que utilizan Spring MVC para manejar solicitudes web y presentar datos dinámicos al cliente.

Requerimiento número 5. MOSTRAR EL CONSUMO EN HOTELANDES POR UN USUARIO DADO, EN UN RANGO DE FECHAS INDICADO

@Query(value="SELECT SUM(COSTO) AS Consumo_Total FROM CONSUMOS WHERE ID_USUARIO = :id AND FECHA_DE_PAGO BETWEEN TO_DATE(:fechaInicio,'YYYY-MM-DD') AND TO_DATE(:fechaFin,'YYYY-MM-DD')", nativeQuery=true)

Integer darConsumosUsuario(@Param("id") Integer id, @Param("fechaInicio") String fechaInicio, @Param("fechaFin") String fechaFin);

Explicación de la query

SELECT SUM(COSTO) AS Consumo_Total: Esta expresión selecciona la suma de los valores en la columna COSTO, la cual representa el gasto asociado a los consumos realizados. El resultado se etiqueta como Consumo_Total para su uso en el resultado de la consulta.

FROM CONSUMOS: Indica que la fuente de los datos es la tabla CONSUMOS, la cual contiene los registros de los servicios o productos consumidos por los usuarios.

WHERE ID_USUARIO = :id: Especifica que solamente se deben considerar aquellos consumos realizados por el usuario con el identificador correspondiente al parámetro :id.

AND FECHA_DE_PAGO BETWEEN TO_DATE(:fechaInicio,'YYYY-MM-DD') AND TO_DATE(:fechaFin,'YYYY-MM-DD'): Delimita el cálculo del gasto al rango de fechas establecido entre :fechaInicio y :fechaFin, convertidas al formato de fecha estándar 'YYYY-MM-DD'.

Requerimiento número 6.

Requerimiento número 7. - ENCONTRAR LOS BUENOS CLIENTES

Consumo controller

```
@GetMapping("/buenosClientesConsumo")
   public String buenosClientesPorConsumo(Model model) {
      model.addAttribute("buenosClientes",
consumoRepository.darBuenosClientesPorConsumo());
      return "buenosClientesPorConsumo";
   }
@GetMapping("/serviciosBajaDemanda")
   public String serviciosBajaDemanda(Model model) {
```

```
        model.addAttribute("bajaDemanda",
consumoRepository.darServiciosDeBajaDemanda());
        return "serviciosBajaDemanda";
    }
```

Alojamieto controller

```
@GetMapping("/buenosClientesAlojamiento")
    public String buenosClientesAlojamiento(Model model) {
        model.addAttribute("clientes",
alojamientoRepository.darBuenosClientesPorAlojamiento());
        return "buenosClientesPorAlojamiento";
    }
```

Explicación querys

Controlador de Consumo:

El método buenosClientesPorConsumo en el controlador ConsumoController está asociado a la ruta /buenosClientesConsumo. Al invocarse, realiza una consulta mediante consumoRepository para obtener una lista de buenos clientes basada en su consumo. Esta lista se agrega al modelo con el nombre buenosClientes. Finalmente, se retorna la vista buenosClientesPorConsumo que se encargará de mostrar estos datos al usuario.

Controlador de Alojamiento:

De manera similar, el método buenosClientesAlojamiento en el controlador AlojamientoController, asignado a la ruta /buenosClientesAlojamiento, consulta a alojamientoRepository para obtener una lista de buenos clientes según su historial de alojamiento. Los datos obtenidos se añaden al modelo bajo la clave clientes y se devuelve la vista buenosClientesPorAlojamiento.

Estos métodos aplican el patrón Repositorio para desacoplar la lógica de negocio de la lógica de acceso a datos. Aunque los métodos específicos darBuenosClientesPorConsumo y darBuenosClientesPorAlojamiento no están explicados en detalle, se presume que ejecutan consultas para identificar a los clientes "buenos" basándose en criterios como la frecuencia de consumo o la fidelidad en el alojamiento.

Además, está el método:

serviciosBajaDemanda: Este maneja la ruta /serviciosBajaDemanda y recurre a consumoRepository para obtener una lista de servicios de baja demanda, sugiriendo aquellos

servicios que posiblemente no son muy utilizados por los clientes. Los resultados se incorporan al modelo y se muestra la vista serviciosBajaDemanda.

Requerimiento número 8. ENCONTRAR LOS SERVICIOS QUE NO TIENEN MUCHA DEMANDA

```
@Query(value = "SELECT servicios_id, COUNT(*) as FRECUENCIA FROM Consumos\r\n" + //
        "WHERE FECHA_DE_PAGO BETWEEN ADD_MONTHS(SYSDATE, -12) AND SYSDATE\r\n" + //
        "GROUP BY servicios_id HAVING COUNT(*) < 156", nativeQuery = true)
    Collection<RespInfoConsumos> darServiciosDeBajaDemanda();
```

Explicación de la query

Selección y Agregación: Se selecciona el servicios_id y se cuenta el número de veces que cada servicios_id aparece en la tabla Consumos, asignando a esta cuenta el alias FRECUENCIA.

Filtrado de Fecha: Los registros en la tabla Consumos se filtran para incluir solo aquellos cuya FECHA_DE_PAGO esté dentro del último año, utilizando SYSDATE para la fecha actual y ADD_MONTHS(SYSDATE, -12) para obtener la fecha de un año atrás.

Agrupación: Los resultados se agrupan por servicios_id, reiniciando la cuenta para cada identificador de servicio diferente.

Condición HAVING: Tras la agrupación, se aplica una condición que incluye solo aquellos grupos donde la cuenta es menor a 156. Esto indica que los servicios considerados de baja demanda son aquellos consumidos menos de 156 veces en el último año.

Resultado: La consulta devuelve una colección de objetos RespInfoConsumos, que probablemente es una clase de proyección diseñada para contener los campos servicios_id y FRECUENCIA.

Carga de datos

En el sistema de gestión de HotelAndes, se implementa un proceso automatizado de carga de datos para poblar la tabla `roles` en la base de datos Oracle. Este procedimiento se realiza a través de un script en Java, específicamente en el paquete `niandes.edu.co.hotelandes.Carga_de_datos.

El script establece una conexión con la base de datos utilizando JDBC (Java Database Connectivity), conectándose a través de una URL específica, junto con las credenciales de usuario y contraseña proporcionadas. Una vez establecida la conexión, se prepara una sentencia SQL de inserción (INSERT INTO) para agregar nuevos registros a la tabla `roles`.

El script genera automáticamente 10,000 entradas únicas. Para cada entrada, se asigna un id_rol incremental, empezando desde 1 hasta 10,000. Los roles son seleccionados de un conjunto predefinido que incluye roles como "cliente", "recepcionista", "empleado", "administrador" y "gerente". Un rol se asigna aleatoriamente a cada id_rol usando la clase `Random` de Java. Además, se genera una descripción básica para cada rol.

Finalmente, cada conjunto de d_rol, rol y descripcion se inserta en la base de datos mediante la ejecución de la sentencia SQL preparada. El proceso se repite en un bucle hasta que todas las entradas han sido cargadas. Al finalizar, el script cierra la sentencia preparada y la conexión con la base de datos para mantener la integridad y la seguridad de los datos.