

Manual de Backend – UV Alert



UNIVERSIDAD NACIONAL DE COLOMBIA

Integrantes:

Miguel Angel Rodriguez.

Indaliria Valentina Cardona.

Jhon Alex Riascos.

Docente: Néstor German Bolivar Pulgarin
Programación orientada a objetos

Contenido

Resumen.....	4
Introducción	4
Estado del arte.....	5
Tecnologías asociadas.....	5
Diseño del prototipo de software	6
Retos vigentes y análisis prospectivo.....	6
Conclusiones	7
Preservación y manejo de la información	7
Integración con la API de Weatherbit.io.....	11
Consideraciones para establecer conexión con el ESP32	14
Librerías utilizadas en el proyecto	16
Ejecución de la aplicación	17
Clases dentro de <i>Views</i>	18
Clase <i>AlertView</i>	18
Clase <i>ConfigurationView</i>	19
Clase <i>DashboardView</i>	24
Clase <i>HomepageView</i>	25
Clase <i>NotificationView</i>	25
Clase <i>RegisterView</i>	26
Clase <i>UserView</i>	27
Clase <i>LoginView</i>	27
Clase <i>ReportView</i>	28
Clases dentro de <i>controllers</i>	29
Clase <i>ApiService</i>	29
Clase <i>DatabaseConnector</i>	33
Clase <i>ExpositionDetailController</i>	36
Clase <i>SensorService</i>	41
Clase <i>UserConfigurationController</i>	44
Clase <i>UserController</i>	49
Clase <i>UserNotificationController</i>	53

Clases dentro de <i>models</i>	59
Clase ExpositionDetail	59
Clase User	62
Clase UserConfiguration	68
Clase UserNotification	72
Montaje del Proyecto en Otro Equipo	79
Módulos de Interacción y Funcionalidades	84
Lista de referencias	92

Resumen

El presente documento detalla el desarrollo de un sistema de alerta de exposición a rayos UV, diseñado para proporcionar recomendaciones personalizadas a los usuarios en función de su tipo de piel, condiciones climáticas y nivel de exposición a la radiación solar. Este sistema utiliza una estructura de software basada en el patrón de arquitectura Modelo-Vista-Controlador (MVC), implementado en Java, junto con una base de datos MySQL y un sensor de radiación UV. Se describen las tecnologías empleadas, los retos enfrentados durante el desarrollo, y se presentan posibles mejoras y futuras aplicaciones del sistema.

Introducción

La exposición a la radiación ultravioleta (UV) del sol es un factor de riesgo significativo para la salud, que puede causar desde quemaduras solares hasta cáncer de piel. Con el objetivo de mitigar estos riesgos, se ha desarrollado un sistema que monitorea la radiación UV en tiempo real y proporciona recomendaciones personalizadas a los usuarios. El sistema se basa en la integración de múltiples tecnologías, incluyendo sensores de radiación, APIs de servicios meteorológicos y un modelo de software robusto que sigue la arquitectura MVC. Este documento presenta el desarrollo del sistema, sus componentes principales, y los desafíos enfrentados durante su implementación.

Estado del arte

El monitoreo de la exposición a la radiación UV no es un concepto nuevo; existen diversas aplicaciones y dispositivos que buscan proteger a los usuarios de los efectos dañinos del sol. Sin embargo, la mayoría de estas soluciones se centran en alertar sobre la intensidad de la radiación sin tener en cuenta las características individuales del usuario, como el tipo de piel o la presencia de enfermedades cutáneas. Este proyecto se diferencia al ofrecer recomendaciones personalizadas, utilizando datos en tiempo real y configuraciones específicas para cada usuario.

Tecnologías asociadas

El desarrollo del sistema se apoyó en una combinación de tecnologías modernas y bien establecidas en la industria del software. Entre ellas, se destacan:

- **Java:** Lenguaje de programación utilizado para desarrollar la lógica del negocio, implementar el patrón MVC y manejar la interacción con la base de datos.
- **MySQL:** Sistema de gestión de bases de datos utilizado para almacenar la información del usuario, sus configuraciones y las mediciones de exposición.
- **API de OpenWeatherMap:** Proporciona datos meteorológicos en tiempo real, incluyendo el índice UV, que se integran en el sistema para generar recomendaciones precisas.
- **Sensor de radiación UV:** Dispositivo que captura los niveles de radiación en el entorno del usuario y los envía al sistema para su procesamiento.

Diseño del prototipo de software

El sistema está diseñado siguiendo el patrón de arquitectura MVC, lo que permite una separación clara entre la lógica del negocio, la interfaz de usuario y la gestión de datos. Este diseño facilita el mantenimiento y la escalabilidad del sistema. El modelo está compuesto por las clases que representan los datos y las operaciones relacionadas con la base de datos, mientras que las vistas están encargadas de la interfaz de usuario. Los controladores actúan como intermediarios, gestionando las interacciones entre el modelo y la vista, y procesando las solicitudes del usuario.

El sistema incluye varios módulos clave:

- **Controladores:** Gestionan la lógica del negocio y las interacciones con las APIs externas y la base de datos.
- **Modelos:** Representan los datos de la aplicación y las operaciones de manipulación de estos datos.
- **Vistas:** Proporcionan la interfaz de usuario y presentan la información de manera interactiva y comprensible.

Retos vigentes y análisis prospectivo

Uno de los principales desafíos de UV Alert es su dependencia de un dispositivo con sensor UV y conexión a internet, lo que puede limitar su accesibilidad. Además, al ser una aplicación de escritorio, su uso es menos flexible en comparación con aplicaciones móviles, lo que podría reducir su adopción. En el futuro, sería ideal desarrollar una versión móvil para mejorar la accesibilidad y conveniencia del usuario, permitiendo un mayor alcance y facilitando la integración con dispositivos portátiles para una monitorización más precisa y continua de la exposición UV.

Conclusiones

Este proyecto representa un paso significativo hacia la protección personalizada contra la radiación UV, combinando tecnologías avanzadas y un diseño de software robusto. El uso del patrón MVC ha demostrado ser eficaz para mantener la modularidad y facilitar el mantenimiento del sistema. A pesar de los desafíos, el sistema ha logrado cumplir con sus objetivos principales, y se prevé un desarrollo continuo para mejorar y expandir sus capacidades.

Preservación y manejo de la información

El proyecto ha sido diseñado para facilitar la gestión y manipulación de datos mediante la integración de bases de datos SQL desde el inicio. A lo largo del desarrollo, se han utilizado prácticas estándar para asegurar la integridad y disponibilidad de la información, incluyendo la implementación de métodos eficientes para la creación, lectura, actualización y eliminación de datos (CRUD).

Dado que el proyecto ya cuenta con todas las librerías y dependencias necesarias integradas, no es necesario realizar configuraciones adicionales. Esto permite una transición directa desde el desarrollo hasta la implementación sin complicaciones. La robustez de la arquitectura asegura que los datos se manejen de manera segura y que el sistema esté preparado para futuras actualizaciones o modificaciones sin requerir cambios significativos en la estructura del código base.

Base de Datos

La base de datos utilizada para esta aplicación se denomina **uv_alert** y está diseñada para gestionar la información relacionada con los usuarios, sus configuraciones personales, las notificaciones que reciben, y los detalles de su exposición a rayos UV. A continuación, se describe en detalle la estructura de las tablas principales que componen esta base de datos, así como su propósito y las relaciones entre ellas.

Estructura de la Base de Datos

La base de datos está compuesta por cuatro tablas principales:

1. Tabla users

- **Descripción:** Esta tabla almacena la información básica de los usuarios registrados en la aplicación. Cada registro en esta tabla corresponde a un usuario único.
- **Campos:**
 - **id:** Identificador único para cada usuario. Es de tipo **INT**, autoincremental, y sirve como clave primaria.
 - **age:** Almacena la edad del usuario. Es de tipo **INT**.
 - **name:** Almacena el nombre del usuario. Es de tipo **VARCHAR(255)**.
 - **last_name:** Almacena el apellido del usuario. Es de tipo **VARCHAR(255)**.
 - **email:** Almacena la dirección de correo electrónico del usuario. Es de tipo **VARCHAR(255)**, único y no nulo, lo que asegura que cada usuario tiene un correo electrónico distinto.
 - **password:** Almacena la contraseña del usuario. Es de

tipo **VARCHAR(255)** y no nulo.

- **Relaciones:** La tabla **users** se relaciona con las tablas **user_configuration**, **user_notifications**, y **exposition_detail** a través del campo **user_id**.

2. Tabla **user_configuration**

- **Descripción:** Esta tabla almacena las configuraciones personalizadas de los usuarios, tales como el tipo de piel, el tiempo máximo de exposición a los rayos UV, y si desean recibir notificaciones de la API.
- **Campos:**
 - **id:** Identificador único para cada configuración de usuario. Es de tipo **INT**, autoincremental, y sirve como clave primaria.
 - **user_id:** Referencia al identificador único del usuario en la tabla **users**. Es de tipo **INT** y no nulo.
 - **skin:** Almacena el tipo de piel del usuario, lo cual puede influir en la forma en que se gestionan las recomendaciones de exposición. Es de tipo **VARCHAR(255)**.
 - **time_exposition:** Almacena el tiempo máximo que el usuario puede estar expuesto a los rayos UV sin correr riesgo, expresado en minutos. Es de tipo **INT** y no nulo.
 - **disease:** Almacena información sobre enfermedades que el usuario pueda tener, lo cual puede afectar la sensibilidad a los rayos UV. Es de tipo **VARCHAR(255)** y no nulo.
 - **api_notification:** Almacena un valor de tipo **TINYINT(1)** que indica si el

usuario desea recibir notificaciones automáticas de la API. Un valor de **1** indica que las notificaciones están activadas, mientras que un valor de **0** indica que están desactivadas.

- **Relaciones:** Esta tabla se relaciona con la tabla **users** a través del campo **user_id**.

3. Tabla **user_notifications**

- **Descripción:** Esta tabla almacena las notificaciones que se envían a los usuarios, incluyendo mensajes sobre riesgos de exposición y otras alertas relevantes.
- **Campos:**
 - **id:** Identificador único para cada notificación. Es de tipo **INT**, autoincremental, y sirve como clave primaria.
 - **user_id:** Referencia al identificador único del usuario en la tabla **users**. Es de tipo **INT** y no nulo.
 - **date:** Almacena la fecha y hora en que se envió la notificación. Es de tipo **TIMESTAMP** y no nulo.
 - **message:** Almacena el contenido del mensaje de la notificación. Es de tipo **TEXT** y no nulo.
 - **state:** Almacena el estado de la notificación, como "leída", "no leída", o "enviada". Es de tipo **VARCHAR(255)**.
- **Relaciones:** Esta tabla se relaciona con la tabla **users** a través del campo **user_id**.

4. Tabla **exposition_detail**

- **Descripción:** Esta tabla registra los detalles de la exposición de los usuarios a los rayos UV, incluyendo el índice UV y la duración de la exposición.
- **Campos:**

- **id:** Identificador único para cada registro de exposición. Es de tipo **INT**, autoincremental, y sirve como clave primaria.
- **user_id:** Referencia al identificador único del usuario en la tabla **users**. Es de tipo **INT** y no nulo.
- **uv_data:** Almacena el valor del índice UV registrado durante la exposición. Es de tipo **DECIMAL(10, 2)** y no nulo.
- **time:** Almacena la duración de la exposición en minutos. Es de tipo **DECIMAL(10, 2)** y no nulo.
- **date:** Almacena la fecha y hora en que se registró la exposición. Es de tipo **TIMESTAMP** y no nulo.
- **Relaciones:** Esta tabla se relaciona con la tabla **users** a través del campo **user_id**.

Integración con la API de Weatherbit.io

Descripción General:

En este proyecto, se utiliza la API de [Weatherbit.io](https://weatherbit.io) para obtener datos meteorológicos en tiempo real, específicamente la temperatura y el índice UV para una ubicación geográfica determinada. Estos datos son de utilidad para que el usuario tome decisiones informadas basadas en la situación actual del clima.

Configuración de la API:

La clave de la API (**apiKey**) es un elemento esencial para la interacción con [Weatherbit.io](https://weatherbit.io). Esta clave permite autenticar las solicitudes realizadas a la API y debe manejarse con cuidado para evitar su exposición en el código fuente. En este proyecto, la clave API y las coordenadas

geográficas (latitud y longitud) se configuran directamente en la clase **ApiService**. Estas coordenadas determinan la ubicación específica para la cual se obtendrán los datos meteorológicos.

Código Ejemplo:

```
private static String apiKey = "YOUR_API_KEY";  
private static double longitude = -74.08175; // Cambiar según sea necesario  
private static double latitude = 4.60971; // Cambiar según sea necesario
```

Es recomendable extraer estos valores a un archivo de configuración o utilizar variables de entorno para mejorar la seguridad y la flexibilidad de la aplicación.

Métodos de Interacción con la API:

Método `getUvData`: Este método realiza una solicitud GET a la API de [Weatherbit.io](https://weatherbit.io) para obtener el índice UV actual en la ubicación especificada. La solicitud se realiza utilizando **HttpClient** para enviar la solicitud HTTP y **HttpResponse** para manejar la respuesta.

Código Ejemplo:

```
public static String getUvData() throws IOException, InterruptedException {  
    HttpClient client = HttpClient.newHttpClient();  
    HttpRequest request = HttpRequest.newBuilder()  
        .uri(URI.create("http://api.openweathermap.org/data/2.5/uv?lat=" + latitude + "&lon=" + longitude + "&appid=" + apiKey))  
        .GET()  
        .build();  
  
    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());  
    return response.body();  
}
```

Método `getWeatherData`: Este método es similar a `getUvData`, pero en lugar de obtener el índice UV, recupera datos meteorológicos actuales como la temperatura en grados Celsius.

Código Ejemplo:

```

public static String getWeatherData() throws IOException, InterruptedException {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("http://api.openweathermap.org/data/2.5/weather?lat=" + latitude + "&lon=" + longitude + "&appid=" + apiKey + "&units=metric"))
        .GET()
        .build();

    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
    return response.body();
}

```

Ambos métodos devuelven la respuesta de la API en formato JSON, que luego se procesa para extraer la información relevante.

Uso de la API en la Aplicación:

Clase DashboardView: En la clase **DashboardView**, la API se utiliza para obtener datos meteorológicos cada 10 segundos. Los datos obtenidos, como el índice UV y la temperatura, se muestran en la interfaz gráfica de usuario y se utilizan para generar alertas que informan al usuario sobre los riesgos potenciales de la exposición solar.

Código Ejemplo:

```

private void fetchDataApi() {
    while (runningApi) {
        try {
            ApiService.buildData();
            latestData.set(ApiService.getLatestData());
            String uvRays = Double.toString(latestData.get().getUvIndex());
            String temperature = Double.toString(latestData.get().getTemperature());
            String date = String.valueOf(latestData.get().getDate());
            this.uvRays.setText(uvRays);
            this.temperature.setText(temperature + "° C");
            String validate = userConfig.validateConfiguration(userId);
            if (validate != "no info") {
                this.checkNotification(Double.valueOf(uvRays));
            }

            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Api thread was interrupted.");
            break;
        } catch (Exception error) {
            System.err.println("Error: " + error.getMessage());
            break;
        }
    }
}

```

Este método se ejecuta en un hilo separado para no bloquear la interfaz de usuario, y actualiza

la información visualizada en la aplicación de manera continua. Además, verifica las configuraciones del usuario para determinar si se deben generar alertas basadas en los valores obtenidos.

Consideraciones para establecer conexión con el ESP32

1. Establecer la red wifi a la cual se va conectar el microcontrolador

1.1. Conexión para red abierta: se reemplaza la constante “ssid” por el nombre de la red abierta a la cual se va a conectar.

```
const char* ssid = "UNAL";           // Nombre de tu red Wi-Fi
//const char* password = "Shana2010"; // Contraseña de tu red Wi-Fi

void setup() {
  pinMode(sensorPin, INPUT);
  Serial.begin(115200);

  // Conectar a Wi-Fi
  WiFi.begin(ssid);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Conectando a Wi-Fi...");
  }
}
```

1.2. Conexión para red privada: se instancian las variables “ssid” y “password” con los datos de la red a la cual se va a conectar.

```
const char* ssid = "UNAL";           // Nombre de tu red Wi-Fi
const char* password = "Shana2010"; // Contraseña de tu red Wi-Fi

void setup() {
  pinMode(sensorPin, INPUT);
  Serial.begin(115200);

  // Conectar a Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Conectando a Wi-Fi...");
  }
}
```

2. Obtener dirección IP de la red con la que se estableció conexión: la línea de código “WiFi.localIP()” devuelve la IP relacionada con la red wifi y en la línea “server.on” se establece la ruta “/data” para subir la información en formato JSON.

```
Serial.println("Conectado a Wi-Fi");
Serial.println(WiFi.localIP()); // Imprimir la IP local asignada

// Iniciar el tiempo de recolección de datos
startMillis = millis();

// Definir la ruta para servir el JSON
server.on("/data", HTTP_GET, handleSendData);

// Iniciar el servidor
server.begin();
```

Ejemplo de serial.println: “192.168.10.27/data”

3. Conexión con el proyecto JAVA: se establece la dirección IP obtenida en el paso anterior en el constructor de la clase sensorService.java

```
public class SensorService {

    private static String esp32IP;
    private static String urlStr;

    public SensorService() {
        esp32IP = "192.168.10.27/data";
        urlStr = "http://" + esp32IP;
        getData();
    }
}
```

4. El equipo donde se va ejecutar el proyecto debe estar conectado a la misma red wifi establecida para el ESP32.

Librerías utilizadas en el proyecto

El proyecto utiliza varias librerías para facilitar la implementación de diferentes funcionalidades clave, tales como la creación de la interfaz gráfica de usuario, la conexión a bases de datos, y el manejo de datos en formato JSON. A continuación, se describen las principales librerías utilizadas:

- **Absolute Layout - AbsoluteLayout.jar**: Se utiliza para gestionar el diseño y la disposición de los componentes en la interfaz gráfica, permitiendo mayor control sobre la colocación de elementos en las ventanas.
- **FlatLaf (flatlaf-3.5.1.jar y flatlaf-intellij-themes-3.5.1.jar)**: Es una librería para crear interfaces gráficas modernas y personalizables con un estilo plano. FlatLaf también incluye temas inspirados en IntelliJ, proporcionando una apariencia limpia y profesional a la aplicación.
- **JSON (json-20240303.jar)**: Esta librería es utilizada para procesar datos en formato JSON, lo que es esencial para manejar la comunicación entre la aplicación y APIs externas como la API de [Weatherbit.io](https://weatherbit.io), que devuelve los datos meteorológicos en formato JSON.
- **MySQL Connector/J (mysql-connector-j-9.0.0.jar)**: Es el controlador JDBC para MySQL, que permite que la aplicación Java se conecte a la base de datos MySQL y ejecute consultas SQL para almacenar y recuperar datos de la aplicación.

Todas estas librerías se encuentran incluidas en el proyecto como archivos .jar y pueden ser descargadas desde el repositorio de GitHub del proyecto. Puedes encontrar todos los archivos .jar necesarios en el siguiente enlace:

https://github.com/valentinacardona07/Proyecto_UV_Alert/tree/main.

Ejecución de la aplicación

Después de haber importado todas las librerías necesarias y configurado los parámetros requeridos para la ejecución del proyecto, simplemente procede a ejecutar la aplicación. Una vez iniciada, podrás acceder a las diferentes funcionalidades como el registro, inicio de sesión, y los módulos específicos de interacción con los datos y servicios disponibles en el sistema. La aplicación está diseñada para ofrecer una experiencia intuitiva y eficiente, permitiendo a los usuarios obtener recomendaciones personalizadas basadas en datos de exposición a rayos UV y otros factores relevantes.

Clases dentro de *Views*

Clase **AlertView**

Descripción: La clase **AlertView** es una ventana gráfica (JFrame) que se utiliza para mostrar alertas al usuario dentro de la aplicación. Esta clase proporciona una interfaz sencilla para mostrar mensajes informativos o de advertencia, con un botón de "OK" para cerrar la alerta.

Métodos:

AlertView()

Descripción: La clase **AlertView** es una ventana gráfica (JFrame) utilizada para mostrar alertas al usuario dentro de la aplicación. Esta clase facilita la presentación de mensajes de advertencia o informativos con un botón "OK" para cerrarlas.

- **Propósito General:** Mostrar mensajes de alerta al usuario en diversas situaciones dentro de la aplicación.
- **Métodos Clave:**
 - **AlertView():** Constructor que inicializa la ventana de alerta.
 - **setMessage(String text):** Establece el mensaje a mostrar en la ventana.
 - **jButton1ActionPerformed(java.awt.event.ActionEvent evt):** Cierra la ventana cuando el usuario presiona "OK".

Uso en la Aplicación: Esta clase se utiliza principalmente en situaciones donde es necesario

informar al usuario de algún error o advertencia.

Clase **ConfigurationView**

Descripción: La clase **ConfigurationView** es un panel gráfico (JPanel) que permite al usuario configurar sus preferencias dentro de la aplicación, como el tiempo de exposición al sol, el fototipo de piel, las enfermedades relacionadas con la piel, y las notificaciones de radiación UV. La clase interactúa con controladores para recuperar y guardar la configuración del usuario.

Métodos:


ConfigurationView(int userId)

Descripción: Constructor de la clase que recibe el ID del usuario y lo almacena para cargar y mostrar su configuración personalizada.

Parámetros:

- **userId:** El ID del usuario que está configurando sus preferencias.

Código:



```
public ConfigurationView(int UserId) {  
    this.userId = UserId;  
    initComponents();  
    fillInfo();  
}
```

Detalles:

- Llama al método **initComponents()** para inicializar los componentes gráficos.
- Llama al método **fillInfo()** para cargar y mostrar la configuración actual del usuario.

fillInfo()

Descripción: Recupera y muestra la configuración existente del usuario en los componentes correspondientes.

Código:

```

public void fillInfo() {
    UserConfigurationController userConfig = new UserConfigurationController();
    List<Map<String, Object>> configData = new ArrayList<>();
    String validateConfig = userConfig.validateConfiguration(userId);

    if (!validateConfig.equals("no_info")) {
        configData = userConfig.getConfigurations(userId);
        if (configData != null) {
            for (Map<String, Object> config : configData) {
                String exposureTime = String.valueOf(config.get("time_exposition"));
                String skin = String.valueOf(config.get("skin"));
                String disease = String.valueOf(config.get("disease"));
                String apiNotification = String.valueOf(config.get("api_notification"));

                if (apiNotification.equals("1")) {
                    this.apiNotification.setSelected(true);
                }
                if (!disease.isEmpty()) {
                    fillDiseaseSelector(disease);
                }
                if (!skin.isEmpty()) {
                    fillSkinSelector(skin);
                }
                if (!exposureTime.isEmpty()) {
                    this.exposureTime.setText(exposureTime);
                }
            }
            this.saveButton.setEnabled(false);
        }
    }
}

```

Detalles:

- Utiliza **UserConfigurationController** para obtener la configuración actual del usuario.

- Actualiza los componentes gráficos (como botones de radio y campos de texto) para reflejar la configuración del usuario.

printAlert(String message)

Descripción: Muestra una ventana de alerta con un mensaje específico.

Parámetros:

- **message:** El mensaje que se desea mostrar en la alerta.

Código:

```
public void printAlert(String message) {  
    AlertView alert = new AlertView();  
    alert.setMessage(message);  
    alert.setModalExclusionType(Dialog.ModalExclusionType.APPLICATION_EXCLUDE); // Evita  
    alert.setAlwaysOnTop(true); // Asegura que la alerta esté siempre en primer plano  
    alert.setVisible(true);  
    alert.setLocationRelativeTo(this);  
}
```

Detalles:

- Crea y configura una instancia de **AlertView** para mostrar mensajes al usuario.


fillSkinSelector(String skin)

Descripción: Selecciona el fototipo de piel correspondiente en los botones de radio según el valor proporcionado.

Parámetros:

- **skin:** El tipo de piel seleccionado (e.g., "phototype_1").

Código:



```
public void fillSkinSelector(String skin) {  
    System.out.println(skin);  
    if (skin.equals("phototype_1")) {  
        this.phototype_1.setSelected(true);  
    } else if (skin.equals("phototype_2")) {  
        this.phototype_2.setSelected(true);  
    } else if (skin.equals("phototype_3")) {  
        this.phototype_3.setSelected(true);  
    } else if (skin.equals("phototype_4")) {  
        this.phototype_4.setSelected(true);  
    } else if (skin.equals("phototype_5")) {  
        this.phototype_5.setSelected(true);  
    } else if (skin.equals("phototype_6")) {  
        this.phototype_6.setSelected(true);  
    }  
}
```

Detalles:

- Asocia el valor de **skin** con el botón de radio correspondiente en la interfaz.


fillDiseaseSelector(String disease)

Descripción: Selecciona la enfermedad o condición correspondiente en los botones de radio según el valor proporcionado.

Parámetros:

- **disease:** La enfermedad o condición seleccionada (e.g., "cancer").

Código:



```

public void fillDiseaseSelector(String disease) {
    if (disease.equals("burn")) {
        this.burn.setSelected(true);
    } else if (disease.equals("cancer")) {
        this.cancer.setSelected(true);
    } else if (disease.equals("aging")) {
        this.aging.setSelected(true);
    } else if (disease.equals("dermatitis")) {
        this.dermatitis.setSelected(true);
    } else if (disease.equals("lupus")) {
        this.lupus.setSelected(true);
    } else if (disease.equals("dermatosis")) {
        this.dermatosis.setSelected(true);
    } else if (disease.equals("none")) {
        this.none.setSelected(true);
    }
}

```

Detalles:

- Asocia el valor de **disease** con el botón de radio correspondiente en la interfaz.

saveButtonActionPerformed(java.awt.event.ActionEvent evt)

Descripción: Guarda la configuración del usuario en la base de datos cuando se presiona el botón "Guardar".

Detalles:

- Recoge los valores seleccionados por el usuario y los guarda en la base de datos mediante **UserConfigurationController**.
- Maneja posibles excepciones como **NumberFormatException** si el tiempo de exposición no es un número entero.

Componentes Gráficos:

- **Botones de Radio (JRadioButton):** Utilizados para seleccionar el fototipo de piel y las enfermedades relacionadas.

- **Campo de Texto (JTextField):** Utilizado para ingresar el tiempo de exposición.
- **Botón (JButton):** Botón para guardar la configuración.
- **Grupos de Botones (ButtonGroup):** Organizan los botones de radio para asegurar que solo uno se pueda

Clase DashboardView

Descripción: **DashboardView** es la ventana principal de la aplicación que actúa como un panel de control. Maneja la visualización de información sobre la radiación UV, la temperatura, y permite la navegación entre diferentes secciones de la aplicación.

- **Propósito General:** Proveer un panel central donde el usuario pueda interactuar con las principales funcionalidades de la aplicación.
- **Métodos Clave:**
 - **startApi():** Inicia un hilo separado para obtener datos de la API de clima en intervalos regulares.
 - **fetchDataApi():** Obtiene datos de la API del clima y actualiza la interfaz con la radiación UV y la temperatura.
 - **checkNotification(double uvRays):** Verifica si es necesario enviar una notificación al usuario basado en los datos UV actuales.

- **SetDate():** Establece la fecha actual en el panel de control del dashboard.

Uso en la Aplicación: **DashboardView** es esencial para mostrar datos en tiempo real y permitir la navegación hacia otras funcionalidades como configuraciones y reportes.

Clase HomepageView

Descripción: **HomepageView** representa la pantalla principal o de bienvenida de la aplicación. Su objetivo es ofrecer una introducción general y guiar al usuario en las funcionalidades de la aplicación.

- **Propósito General:** Servir como pantalla de inicio, proporcionando un primer punto de interacción al usuario.
- **Método Clave:**
 - **HomepageView(int userId):** Constructor que inicializa la pantalla con los elementos gráficos necesarios.

Uso en la Aplicación: La clase se muestra al usuario cuando accede al tablero principal de la aplicación, brindando una introducción amigable y clara.

Clase NotificationView

Descripción: **NotificationView** es un panel gráfico que permite al usuario gestionar las notificaciones recibidas. Ofrece funcionalidades para visualizar, refrescar, y eliminar notificaciones.

- **Propósito General:** Facilitar la gestión de notificaciones generadas por la aplicación.
- **Métodos Clave:**
 - **fillTable():** Carga las notificaciones del usuario en la tabla gráfica.
 - **deleteSelectedRow():** Elimina la notificación seleccionada de la tabla y de la base de datos.
 - **deleteAllActionPerformed(java.awt.event.ActionEvent evt):** Elimina todas las notificaciones almacenadas en la base de datos.

Uso en la Aplicación: Se utiliza dentro del panel de notificaciones para ofrecer al usuario un control completo sobre las alertas y mensajes recibidos.

Clase RegisterView

- **Descripción:** Proporciona la interfaz para registrar un nuevo usuario.
- **Métodos:**
 - **doneActionPerformed(java.awt.event.ActionEvent evt):** Maneja el evento de clic del botón "Done".

- **closeModal(RegisterView register):** Cierra la ventana modal que contiene el formulario de registro.

Clase UserView

Descripción: Esta clase permite al usuario visualizar y actualizar su información personal dentro de la aplicación.

Métodos:

- **loadData(int user_id):** Este método carga los datos del usuario desde la base de datos y los muestra en los campos correspondientes del panel.
- **edit_userActionPerformed(java.awt.event.ActionEvent evt):** Permite que los campos del formulario sean editables.
- **cancelActionPerformed(java.awt.event.ActionEvent evt):** Cancela la edición y restablece los valores originales de los campos.

Clase LoginView

- **Descripción:** Proporciona la interfaz de inicio de sesión para los usuarios de la aplicación.
- **Métodos:**

- **LoginView():** Constructor que inicializa los componentes gráficos de la ventana y configura los manejadores de eventos.
- **jButton1ActionPerformed(java.awt.event.ActionEvent evt):** Maneja el evento de clic en el botón de "Ingresar".
- **registerActionPerformed(java.awt.event.ActionEvent evt):** Maneja el evento de clic en el botón de "Register".

Clase ReportView


Descripción: La clase **ReportView** es un panel gráfico (**JPanel**) que sirve como espacio dedicado para la generación de reportes dentro de la aplicación. Aunque actualmente esta clase contiene un componente básico, está preparada para ser expandida con funcionalidades adicionales relacionadas con la creación y visualización de reportes.

Métodos:

ReportView()

Descripción: Constructor de la clase que inicializa los componentes gráficos del panel.

Código:



```
public ReportView() {
    initComponents();
}
```

Detalles:

- Llama al método **initComponents()** para configurar los elementos de la interfaz gráfica.

Componentes Gráficos:

- **javax.swing.JLabel jLabel1**: Etiqueta que contiene el texto "Espacio para generar reportes". Este componente es un marcador de posición que indica dónde se añadirá la funcionalidad de generación de reportes en el futuro.

Uso en la Aplicación: La clase **ReportView** es parte de la interfaz de usuario de la aplicación y está diseñada para albergar futuras funcionalidades de generación y visualización de reportes.

Clases dentro de *controllers*

Clase ApiService

Descripción: La clase **ApiService** es responsable de interactuar con las APIs externas para obtener datos relacionados con la radiación UV y el clima. Utiliza el paquete **java.net.http** para enviar solicitudes HTTP y procesar las respuestas. Esta clase también gestiona la última instancia de los datos obtenidos a través de un objeto **AtomicReference** para garantizar que siempre se disponga de la información más reciente.

Atributos:

- **AtomicReference<UvDataTemplate> latestData:**

Una referencia atómica que almacena la instancia más reciente de los datos UV y de temperatura obtenidos de las APIs.

- **String apiKey:**

Clave API utilizada para autenticar las solicitudes realizadas a las APIs externas.

- **double longitude y double latitude:**

Coordenadas geográficas que determinan la ubicación para la cual se solicitarán los datos.

Métodos:

static void buildData()

Descripción: Método estático que realiza las llamadas a las APIs para obtener los datos UV y de clima, procesarlos y almacenarlos en la referencia atómica **latestData**.

Código:

```
public static void buildData() {
    try {
        String uvData = getUvData(); // obtiene los datos en formato json y los al
        String weatherData = getWeatherData();
        JSONObject uvJson = new JSONObject(uvData); // se crea un objeto json para
        JSONObject weatherJson = new JSONObject(weatherData);

        String date = uvJson.getString("date_iso");
        double uvIndex = uvJson.getDouble("value");
        double temperature = weatherJson.getJSONObject("main").getDouble("temp");

        UvDataTemplate data = new UvDataTemplate(date, uvIndex, temperature);
        latestData.set(data);
    } catch (IOException | InterruptedException ex) {
        System.err.println("Error: " + ex.getMessage());
    }
}
```

Detalles:

- Se obtienen los datos de UV y clima mediante los métodos **getUvData()** y **getWeatherData()**.
- Los datos se procesan y se almacenan en un objeto **UvDataTemplate** que se guarda en **latestData**.
- Maneja excepciones de IO e interrupciones durante la obtención de datos.

static UvDataTemplate getLatestData()

Descripción: Devuelve la instancia más reciente de los datos UV y de temperatura.

Código:

```
public static UvDataTemplate getLatestData() {
    //Funcion de tipo UvData, solo retorna dicho tipo.
    return latestData.get();
}
```

Detalles:

- Proporciona acceso a los datos más recientes almacenados en **latestData**.
- **static String getUvData()**

Descripción: Realiza una solicitud HTTP a la API de OpenWeatherMap para obtener los datos UV.

Código:

```
public static String getUvData() throws IOException, InterruptedException {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("http://api.openweathermap.org/data/2.5/uvi?lat=" + latitude + "&lon=" + longitude + "&appid=" + apiKey))
        .GET()
        .build();

    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
    return response.body();
}
```

Detalles:

- Construye y envía una solicitud HTTP para obtener datos UV.
- Retorna la respuesta en formato JSON como una cadena de texto.
- **static String getWeatherData()**

Descripción: Realiza una solicitud HTTP a la API de OpenWeatherMap para obtener datos del clima.

Código:

```
public static String getWeatherData() throws IOException, InterruptedException {  
    HttpClient client = HttpClient.newHttpClient();  
    HttpRequest request = HttpRequest.newBuilder()  
        .uri(URI.create("http://api.openweathermap.org/data/2.5/weather?lat=" + latitude + "&lon=" + longitude + "&appid=" + apiKey + "&units=metric"))  
        .GET()  
        .build();  
  
    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());  
    return response.body();  
}
```

Detalles:

- Similar a **getUvData()**, pero para datos de clima, incluyendo la temperatura.

Clase Interna: UvDataTemplate

- **Descripción:** Esta clase encapsula los datos de UV y temperatura en un solo objeto.
- **Atributos:**
 - **String date:** Fecha del reporte.
 - **double uvIndex:** Índice UV.

- **double temperature:** Temperatura en grados Celsius.
- **Métodos:**
 - **UvDataTemplate(String date, double uvIndex, double temperature):**
Constructor que inicializa los atributos.
 - Métodos **getDate()**, **getUvIndex()**, y **getTemperature()** para acceder a los datos encapsulados.

Uso en la Aplicación: **ApiService** actúa como el punto central para la obtención de datos UV y climáticos, proporcionando estos datos de manera encapsulada a otras partes de la aplicación, como los controladores o las vistas que requieran la información para tomar decisiones o mostrar advertencias al usuario.

Clase DatabaseConnector

Descripción: La clase **DatabaseConnector** es responsable de gestionar la conexión a la base de datos MySQL utilizada en la aplicación. Esta clase encapsula la lógica de conexión y proporciona un método para acceder a la conexión establecida, permitiendo que otras clases interactúen con la base de datos de manera sencilla y centralizada.

Atributos:

- **Connection connection:**

Variable estática que almacena la conexión activa a la base de datos. Al ser estática, se asegura que la conexión sea única y compartida entre todas las instancias de la clase.

- **String driver:**

Nombre del driver JDBC utilizado para conectarse a la base de datos MySQL (**com.mysql.cj.jdbc.Driver**).

- **String user:**

Nombre de usuario utilizado para autenticar la conexión con la base de datos (por defecto, "root").

- **String pass:**

Contraseña asociada al nombre de usuario para la autenticación (por defecto, una cadena vacía).

- **String url:**

URL de conexión a la base de datos, que incluye la dirección del servidor, el puerto y el nombre de la base de datos (**jdbc:mysql://localhost:3306/uv_alert**).

Métodos:

public DatabaseConnector()

Descripción: Constructor de la clase que establece la conexión a la base de datos cuando se crea una instancia de **DatabaseConnector**.

Código:

```
public DatabaseConnector() {  
    connection = null;  
    try {  
        Class.forName(driver);  
        // Nos conectamos al gestor de bd  
        connection = DriverManager.getConnection(url, user, pass);  
        // Si la conexion fue exitosa mostramos un mensaje de conexion exitosa  
        if (connection != null) {  
            System.out.println("Conexion establecida");  
        }  
        // Si la conexion NO fue exitosa mostramos un mensaje de error  
    } catch (ClassNotFoundException | SQLException e) {  
        System.out.println("Error de conexion" + e);  
    }  
}
```

Detalles:

- **Class.forName(driver):** Carga el driver JDBC necesario para la conexión a la base de datos.
- **DriverManager.getConnection(url, user, pass):** Establece la conexión a la base de datos usando la URL, usuario y contraseña proporcionados.
- Si la conexión es exitosa, se imprime un mensaje de confirmación en la consola.
- Si ocurre un error (por ejemplo, si el driver no se encuentra o hay un problema con la conexión), se captura y se imprime un mensaje de error.

public static Connection getConnection()

Descripción: Método estático que devuelve la conexión actual a la base de datos. Si no se ha establecido una conexión, este método retornará **null**.

Código:

```
public static Connection getConnection() {  
    return connection;  
}
```

Detalles:

- Este método permite que otras clases accedan a la conexión establecida sin necesidad de crear una nueva instancia de **DatabaseConnector**.
- Devuelve la conexión existente, lo que facilita la reutilización de la conexión en toda la aplicación.

Uso en la Aplicación: **DatabaseConnector** es fundamental para cualquier operación que requiera acceso a la base de datos. Proporciona una forma centralizada y controlada de gestionar la conexión, lo que ayuda a mantener la consistencia y facilita el manejo de errores relacionados con la base de datos. Las clases que requieren interactuar con la base de datos simplemente llaman a **DatabaseConnector.getConnection()** para obtener una referencia a la conexión actual.

Clase ExpositionDetailController

La **Descripción:** La clase **ExpositionDetailController** maneja la lógica relacionada con la exposición al sol del usuario. Hereda de la clase **ExpositionDetail** y se encarga de coordinar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) relacionadas con los detalles de la exposición del usuario a los rayos UV. Además, proporciona métodos para validar y calcular el

tiempo total de exposición en función de las configuraciones específicas del usuario.


Métodos:

String createExposition(Map<String, String> data)

Descripción:

Este método recibe un mapa de datos con la información necesaria para crear un nuevo registro de exposición. Establece una conexión a la base de datos utilizando la clase **DatabaseConnector**, y luego llama al método **create** de la clase base **ExpositionDetail** para guardar los datos en la base de datos.

Código:



```
public String createExposition(Map<String, String> data) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.create(connection, data);
    }
    return message;
}
```

Detalles:

- El método verifica que la conexión a la base de datos se haya establecido correctamente antes de intentar crear un nuevo registro.
- Devuelve un mensaje indicando el resultado de la operación (éxito o error).

List<Map<String, Object>> getExposition(int user_id, String date)

Descripción:

Recupera los registros de exposición de un usuario específico en una fecha determinada.

Establece una conexión a la base de datos y utiliza el método **getExposition** de la clase base para obtener los datos.

Código:

```
public List<Map<String, Object>> getExposition(int user_id, String date) {  
    List<Map<String, Object>> expositionData = new ArrayList<>();  
    DatabaseConnector connection_ = new DatabaseConnector();  
    Connection connection = connection_.getConnection();  
  
    if (connection != null) {  
        expositionData = super.getExposition(connection, user_id);  
    }  
    return expositionData;  
}
```

Detalles:

- Devuelve una lista de mapas, donde cada mapa representa un registro de exposición para el usuario en la fecha especificada.
- Si no se puede establecer la conexión a la base de datos, retorna una lista vacía.

boolean validateExposition(int user_id, String date)

Descripción:

Valida si el usuario ha excedido su tiempo máximo de exposición basado en su configuración.

Este método obtiene la configuración del usuario utilizando **UserConfigurationController** y compara el tiempo total de exposición con el límite configurado.

Código:

```

public boolean validateExposition(int user_id, String date) {
    UserConfigurationController userConfig = new UserConfigurationController();
    List<Map<String, Object>> configData = new ArrayList<>();
    int totalExposition = getTotalTimeExposition(user_id, date);
    int configExposition = 0;

    configData = userConfig.getConfiguration(user_id);

    if (totalExposition > 0 && configData != null) {
        for (Map<String, Object> config : configData) {
            configExposition = (int) config.get("time_exposition");
            if (totalExposition >= configExposition) {
                return true;
            }
        }
    }
    return false;
}

```

Detalles:

- Verifica la configuración del usuario para determinar el tiempo máximo de exposición permitido.
- Retorna **true** si el usuario ha excedido el tiempo permitido, **false** en caso contrario.

int getTotalTimeExposition(int user_id, String date)

Descripción:

Calcula el tiempo total de exposición de un usuario en una fecha específica sumando todos los registros de exposición para ese día.

Código:

```

public int getTotalTimeExposition(int user_id, String date) {
    List<Map<String, Object>> expositionData = new ArrayList<>();
    expositionData = getExposition(user_id, date);
    int totalTimeExposition = 0;

    if (expositionData != null) {
        for (Map<String, Object> exposition : expositionData) {
            totalTimeExposition += (int) exposition.get("time");
        }
    }
    return totalTimeExposition;
}

```

Detalles:

- Itera sobre todos los registros de exposición y suma los tiempos para obtener el total.
- Retorna el tiempo total de exposición en minutos.

double getTotalExposition(int user_id, String date)

Descripción:

Calcula la exposición promedio a los rayos UV para un usuario en una fecha específica.

Código:

```

public double getTotalExposition(int user_id, String date) {
    List<Map<String, Object>> expositionData = new ArrayList<>();
    expositionData = getExposition(user_id, date);
    double totalExposition = 0.0;
    int countExposition = 0;
    System.out.println("llega");
    if (expositionData != null) {
        for (Map<String, Object> exposition : expositionData) {
            totalExposition += (double) exposition.get("uv_data");
            countExposition += 1;
        }
    }
    return totalExposition / (double) countExposition;
}

```

Detalles:

- Calcula el promedio de los valores de UV registrados durante las exposiciones del usuario en la fecha especificada.
- Retorna el valor promedio de exposición UV.

Uso en la Aplicación: **ExpositionDetailController** se utiliza para gestionar y validar los registros de exposición del usuario. Se integra con otros controladores, como **UserConfigurationController**, para asegurarse de que el usuario no exceda los límites de exposición recomendados según su configuración personal. Esta clase es esencial para mantener la seguridad del usuario en términos de exposición a los rayos UV.

Clase SensorService

La **Descripción:** La clase **SensorService** es responsable de interactuar con un dispositivo ESP32 para obtener datos en tiempo real desde un sensor, presumiblemente relacionado con la medición de la radiación UV. Esta clase establece una conexión HTTP con el dispositivo, recupera los datos, y los procesa para generar notificaciones al usuario en caso de que los valores medidos superen ciertos umbrales de seguridad.

Atributos:

- **String esp32IP:**
Dirección IP del dispositivo ESP32 que está enviando los datos del sensor.

- **String urlStr:**

URL construida a partir de la IP del ESP32 para realizar la solicitud HTTP.

Constructores:

SensorService()

Descripción:

Constructor de la clase que inicializa la IP del ESP32 y construye la URL de solicitud. Llama automáticamente al método **getData()** para comenzar a obtener datos del sensor.

Código:

```
public SensorService() {  
    esp32IP = "192.168.43.133";  
    urlStr = "http://" + esp32IP + "/";  
    getData();  
}
```

Métodos:

void getData()

Descripción:

Método que inicia un bucle continuo para enviar solicitudes HTTP al ESP32 y obtener los datos del sensor. Procesa la respuesta obtenida y la pasa al método **readData()** para su interpretación.

Código:

```

public void getData() {
    while (true) {
        try {
            URL url = new URL(urlStr);
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.setConnectTimeout(5000); // 5 segundos de tiempo de espera para la conexión
            connection.setReadTimeout(5000); // 5 segundos de tiempo de espera para la lectura

            int responseCode = connection.getResponseCode();
            if (responseCode == 200) { // HTTP OK
                BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
                String inputLine;
                StringBuilder response = new StringBuilder();

                while ((inputLine = in.readLine()) != null) {
                    response.append(inputLine);
                }

                in.close();
                readData(response.toString());
            } else {
                System.out.println("Error en la conexión: " + responseCode);
            }
            break;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Detalles:

- Establece una conexión HTTP con el ESP32.
- Lee la respuesta si la conexión es exitosa y pasa los datos a **readData()**.
- Si la conexión falla, imprime el código de error.
- El método contiene un bucle infinito, pero se interrumpe tras la primera ejecución exitosa o fallo de la conexión.

void readData(String uv_read)

Descripción:

Método que interpreta los datos obtenidos del ESP32. Dependiendo del valor de radiación UV leído, genera una notificación para el usuario utilizando **UserNotificationController** y muestra una alerta mediante **AlertView**.

Código:

```

public void readData(String uv_read) {
    UserNotificationController userNotification = new UserNotificationController();
    Map<String, String> data_ = new HashMap<>();
    AlertView alert = new AlertView();
    float uv_read_float = 0;
    if (uv_read != null) {
        uv_read_float = Float.valueOf(uv_read);
        System.out.println(uv_read_float);
        if (uv_read_float >= 1.0) {
            data_.put("message", "<html>Riesgo minimo de daño por exposicion");
            data_.put("user_id", "5");
            data_.put("date", "2024-08-13");
            userNotification.createNotification(data_);

            alert.setMessage(data_.get("message")); // Obtener el valor de la clave "message"
            alert.show();
        }
    }
}

```

Detalles:

- Convierte la lectura UV en un valor flotante (**float**).
- Si el valor es mayor o igual a 1.0, genera un mensaje de advertencia.
- Crea una notificación utilizando **UserNotificationController** y muestra una alerta en la interfaz de usuario.

Uso en la Aplicación: **SensorService** está diseñado para funcionar de manera continua, monitoreando los niveles de radiación UV a través de un sensor conectado al ESP32. Si los niveles detectados son peligrosos, el servicio notifica al usuario a través de la interfaz y guarda un registro de la notificación en la base de datos.

Clase **UserConfigurationController**

Descripción: La clase **UserConfigurationController** extiende la funcionalidad del modelo **UserConfiguration** para manejar la lógica de negocio relacionada con las configuraciones del usuario en la aplicación. Esta clase proporciona métodos para crear, actualizar, y validar

configuraciones de usuario, así como para gestionar notificaciones API específicas.

Métodos:

String createConfiguration(Map<String, String> data)

Descripción: Este método permite crear una nueva configuración para un usuario en la base de datos utilizando la información proporcionada en un **Map**.

Código:

```

public String createConfiguration(Map<String, String> data) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();
    if (connection != null) {
        message = super.createConfigurationModel(connection, data);
    }
    return message;
}

```

Detalles:

- Establece una conexión con la base de datos utilizando **DatabaseConnector**.
- Llama al método **createConfigurationModel** del modelo **UserConfiguration** para insertar los datos en la base de datos.
- Retorna un mensaje que indica el resultado de la operación.
- **List<Map<String, Object>> getConfiguration(int userId)**

Descripción: Obtiene la configuración del usuario especificado por su **userId**.

Código:

```

public List<Map<String, Object>> getConfiguration(int userId) {
    List<Map<String, Object>> configData = new ArrayList<>();
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        configData = super.getConfiguration(connection, userId);
    }
    return configData;
}

```

Detalles:

- Establece una conexión con la base de datos y recupera la configuración del usuario.
- Devuelve una lista de mapas que contienen los datos de la configuración del usuario.

String updateConfiguration(Map<String, String> data)

Descripción: Actualiza la configuración de un usuario en la base de datos utilizando los datos proporcionados.

Código:

```

public String updateConfiguration(Map<String, String> data) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.updateConfiguration(connection, data);
    }
    return message;
}

```

Detalles:

- Establece una conexión con la base de datos.

- Llama al método **updateConfiguration** del modelo **UserConfiguration** para realizar la actualización.
- Retorna un mensaje que indica si la actualización fue exitosa o no.

String validateConfiguration(int userId)

Descripción: Valida la configuración actual del usuario en la base de datos.

Código:

```

public String validateConfiguration(int userId) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.validateConfiguration(connection, userId);
    }
    return message;
}

```

Detalles:

- Conecta a la base de datos para validar la configuración del usuario especificado.
- Utiliza el método **validateConfiguration** del modelo para realizar la validación.
- Devuelve un mensaje de validación.

boolean validateApiNotification(int userId)

Descripción: Valida si las notificaciones API están habilitadas para un usuario específico, basándose en la configuración del usuario.

Código:

```
public boolean validateApiNotification(int userId) {
    List<Map<String, Object>> configData = new ArrayList<>();
    boolean apiNotification = false;
    String apiNotification_ = "";
    configData = getConfiguration(userId);
    if (configData != null) {
        for (Map<String, Object> config : configData) {
            apiNotification_ = String.valueOf(config.get("api_notification"));
            if (apiNotification_.equals("1")) {
                apiNotification = true;
            }
        }
    }
    return apiNotification;
}
```

Detalles:

- Obtiene la configuración del usuario para verificar si las notificaciones API están habilitadas.
- Devuelve **true** si las notificaciones están habilitadas, **false** en caso contrario.

Uso en la Aplicación: La clase **UserConfigurationController** se encarga de gestionar las configuraciones del usuario en la aplicación. Proporciona la lógica necesaria para crear, actualizar, y validar configuraciones, asegurando que los datos se manejen de manera eficiente y segura a través de las interacciones con la base de datos. Esta clase también es crucial para manejar configuraciones que afectan el comportamiento de las notificaciones basadas en la API.

Clase UserController

Descripción: **UserController** es responsable de manejar la lógica relacionada con las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los usuarios en la aplicación. Hereda de la clase **User** del modelo, lo que le permite utilizar métodos predefinidos para interactuar con la base de datos a través de la clase **DatabaseConnector**.

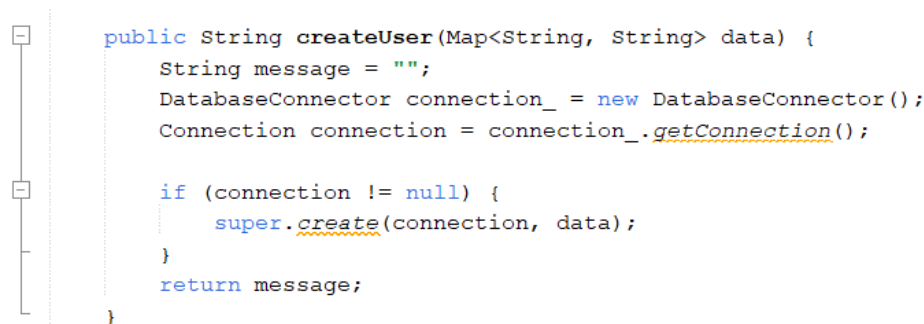
Métodos:

String createUser(Map<String, String> data)

Descripción:

Este método se encarga de crear un nuevo usuario en la base de datos. Toma un mapa de datos (**data**) como parámetro, que contiene los detalles del usuario a crear.

Código:



```
public String createUser(Map<String, String> data) {  
    String message = "";  
    DatabaseConnector connection_ = new DatabaseConnector();  
    Connection connection = connection_.getConnection();  
  
    if (connection != null) {  
        super.create(connection, data);  
    }  
    return message;  
}
```

Detalles:

- Se establece una conexión a la base de datos utilizando **DatabaseConnector**.
- Si la conexión es exitosa, llama al método **create()** de la clase **User** para insertar el nuevo usuario en la base de datos.


- Devuelve un mensaje indicando el resultado de la operación.

Map<String, Object> getUser(int user_id)

Descripción:

Este método recupera los datos de un usuario específico de la base de datos, basado en el **user_id** proporcionado.

Código:



```
public Map<String, Object> getUser(int user_id) {  
    Map<String, Object> userData = new HashMap<>();  
    DatabaseConnector connection_ = new DatabaseConnector();  
    Connection connection = connection_.getConnection();  
  
    if (connection != null) {  
        userData = super.getUser(connection, user_id);  
    }  
    return userData;  
}
```

Detalles:


- Se conecta a la base de datos y recupera los datos del usuario utilizando el método **getUser()** de la clase **User**.
- Devuelve un mapa con la información del usuario solicitado.

String updateUser(Map<String, String> data)

Descripción:

Actualiza los datos de un usuario existente en la base de datos. Recibe un mapa (**data**) que contiene la información actualizada del usuario.

Código:



```

public String updateUser(Map<String, String> data) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.update(connection, data);
    }
    return message;
}


```

Detalles:

- Se conecta a la base de datos y actualiza la información del usuario utilizando el método **update()** de la clase **User**.
- Devuelve un mensaje indicando el resultado de la operación.

String deleteUser(int user_id)**Descripción:**

Elimina un usuario de la base de datos basado en el **user_id** proporcionado.

Código:


```

public String deleteUser(int user_id) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.delete(connection, user_id);
    }
    return message;
}

```

Detalles:

- Se conecta a la base de datos y elimina el usuario utilizando el

método **delete()** de la clase **User**.


- Devuelve un mensaje indicando si la eliminación fue exitosa o no.

String validateLogin(String email, String password)

Descripción:

Valida las credenciales de un usuario (email y contraseña) durante el proceso de inicio de sesión.

Código:



```

public String validateLogin(String email, String password) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.validateLogin(connection, email, password);
    }
    return message;
}

```

Detalles:


- Se conecta a la base de datos y valida las credenciales del usuario utilizando el método **validateLogin()** de la clase **User**.
- Devuelve un mensaje indicando si las credenciales son correctas.

String validateUser(int user_id)

Descripción:

Valida si un usuario específico existe en la base de datos, basado en su **user_id**.

Código:



```

public String validateUser(int user_id) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.validateUser(connection, user_id);
    }
    return message;
}

```

Detalles:

- Se conecta a la base de datos y valida la existencia del usuario utilizando el método **validateUser()** de la clase **User**.
- Devuelve un mensaje indicando si el usuario existe.

Uso en la Aplicación: La clase **UserController** actúa como el intermediario entre las vistas o servicios que gestionan la interfaz del usuario y el modelo de datos de usuarios. Proporciona métodos que encapsulan la lógica de negocio relacionada con los usuarios, asegurando que las operaciones con la base de datos se manejen de manera eficiente y segura.

Clase **UserNotificationController**

Descripción: La clase **UserNotificationController** se encarga de gestionar las notificaciones de los usuarios dentro de la aplicación. Esta clase extiende la funcionalidad de **UserNotification** del paquete **models**, proporcionando métodos para crear, obtener y eliminar notificaciones en la base de datos. Además, implementa lógica personalizada para generar

recomendaciones basadas en las lecturas de radiación UV y el tiempo de exposición.

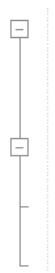
Métodos:

String createNotification(Map<String, String> data)

Descripción:

Este método permite crear una nueva notificación en la base de datos utilizando los datos proporcionados en el mapa **data**.

Código:



```
public String createNotification(Map<String, String> data) {  
    String message = "";  
    DatabaseConnector connection_ = new DatabaseConnector();  
    Connection connection = connection_.getConnection();  
    if (connection != null) {  
        message = super.createNotification(connection, data);  
    }  
    return message;  
}
```

Detalles:

- Establece una conexión con la base de datos utilizando **DatabaseConnector**.
- Llama al método **createNotification** de la clase **UserNotification** para insertar la notificación en la base de datos.
- Retorna un mensaje que puede contener información sobre el éxito o fallo de la operación.

List<Map<String, Object>> getNotification(int user_id)

Descripción:

Obtiene todas las notificaciones de un usuario específico según su ID.

Código:

```
public List<Map<String, Object>> getNotification(int user_id) {  
    List<Map<String, Object>> notificationData = new ArrayList<>();  
    DatabaseConnector connection_ = new DatabaseConnector();  
    Connection connection = connection_.getConnection();  
  
    if (connection != null) {  
        notificationData = super.getNotification(connection, user_id);  
    }  
    return notificationData;  
}
```

Detalles:

- Establece la conexión con la base de datos.
- Llama al método **getNotification** de **UserNotification** para recuperar las notificaciones asociadas con el **user_id**.
- Retorna una lista de mapas que contiene los datos de cada notificación.

String deleteNotification(int id)**Descripción:**

Elimina una notificación específica basada en su ID.

Código:

```

public String deleteNotification(int id) {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.deleteNotification(connection, id);
    }
    return message;
}

```

Detalles:

- Establece la conexión con la base de datos.
- Llama al método **deleteNotification** de **UserNotification** para eliminar la notificación.
- Retorna un mensaje indicando el resultado de la operación.

String deleteAllNotification()**Descripción:**

Elimina todas las notificaciones almacenadas en la base de datos.

Código:

```

public String deleteAllNotification() {
    String message = "";
    DatabaseConnector connection_ = new DatabaseConnector();
    Connection connection = connection_.getConnection();

    if (connection != null) {
        message = super.deleteAllNotification(connection);
    }
    return message;
}

```

Detalles:

- Establece la conexión con la base de datos.
- Llama al método **deleteAllNotification** de **UserNotification** para eliminar todas las notificaciones.
- Retorna un mensaje indicando el resultado de la operación.

String getRecommendation(double uvReading, int exposureTime, int userId)

Descripción:

Genera una recomendación personalizada para el usuario basada en la lectura de radiación UV y el tiempo de exposición. La recomendación también tiene en cuenta las condiciones médicas registradas en la configuración del usuario.

Código:

```

public String getRecommendation(double uvReading, int exposureTime, int userId) {
    UserConfigurationController userConfig = new UserConfigurationController();
    List<Map<String, Object>> configData = new ArrayList<>();
    configData = userConfig.getConfigurations(userId);
    String recommendation = "";
    String disease = "";

    if (configData != null) {
        for (Map<String, Object> config : configData) {
            disease = String.valueOf(config.get("disease"));
        }
    }

    //índice 6, exposición alta en todos los casos
    if (uvReading >= 6.0) {
        // máximo 20 min
        if (exposureTime >= 20 && disease.equals("burn")) { //buscar como ignorar mayúsculas y caracteres especiales
            recommendation = "burn";
        } //máximo 15 min
        else if (exposureTime >= 15 && disease.equals("cancer")) {
            recommendation = "cancer";
        } //máximo 25 min
        else if (exposureTime >= 25 && disease.equals("aging")) {
            recommendation = "aging";
        } //máximo 8 minutos
        else if (exposureTime >= 8 && disease.equals("dermatitis")) {
            recommendation = "dermatitis";
        } //máximo 4 minutos
        else if (exposureTime >= 4 && disease.equals("lupus")) {
            recommendation = "lupus";
        } //máximo 4 minutos
        else if (exposureTime >= 4 && disease.equals("dermatosis")) {
            recommendation = "dermatosis";
        } else {
            recommendation = "general";
        }
    }

    return recommendation;
}

```

Detalles:

- Recupera la configuración del usuario, incluida cualquier condición médica.
- Genera una recomendación basada en la combinación de la lectura de UV, el tiempo de exposición, y la condición médica del usuario.
- Retorna una cadena con la recomendación correspondiente.

String validateUv(Double uv_read, int userId)**Descripción:**

Valida el nivel de radiación UV en función del tipo de piel del usuario.

Código:

```

public String validateUv(Double uv_read, int userId) {
    /**
     *
     */
    UserConfigurationController userConfig = new UserConfigurationController();
    List<Map<String, Object>> configData = new ArrayList<>();
    configData = userConfig.getConfiguration(userId);
    String key = "";
    String skin = "";

    if (configData != null) {
        for (Map<String, Object> config : configData) {
            skin = String.valueOf(config.get("skin"));
        }
    }

    if (uv_read >= 10.0) {
        if (!skin.isEmpty()) {
            key = skin;
        } else {
            key = "high_level";
        }
    } else {
        key = "low_level";
    }

    return key;
}

```

Detalles:

- Recupera el tipo de piel del usuario desde la configuración.
- Evalúa el nivel de radiación UV y determina una clave asociada al riesgo, teniendo en cuenta el tipo de piel.
- Retorna una cadena que indica el nivel de riesgo (por ejemplo, **high_level** o **low_level**).

Uso en la Aplicación: **UserNotificationController** es esencial para la funcionalidad de notificaciones y recomendaciones dentro de la aplicación. Proporciona las herramientas necesarias para informar al usuario sobre su exposición al sol, riesgos potenciales, y para manejar notificaciones de forma eficiente en la base de datos.

Clases dentro de *models*

Clase **ExpositionDetail**

Descripción: La clase **ExpositionDetail** gestiona los detalles de exposición de un usuario a radiación UV en la base de datos. Proporciona métodos para crear un nuevo registro de

exposición y recuperar los datos almacenados, lo que permite un seguimiento detallado de la exposición de los usuarios a la radiación UV.

Métodos:

String create(Connection connection, Map<String, String> data)

Descripción:

Inserta un nuevo registro de exposición en la base de datos utilizando la información proporcionada en el mapa **data**.

Código:

```
public static String create(Connection connection, Map<String, String> data) {
    String message = "";
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
    String sql = "INSERT INTO exposition_detail (user_id, uv_data, date, time )"
        + "VALUES (?, ?, ?, ?)";

    try {
        int user_id = Integer.valueOf(data.get("user_id"));
        double uv_data = Double.valueOf(data.get("uv_data"));
        int time = Integer.valueOf(data.get("time"));
        LocalDate localDate = LocalDate.parse(data.get("date"), formatter);
        Date sqlDate = Date.valueOf(localDate);

        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        statement.setDouble(2, uv_data);
        statement.setDate(3, sqlDate);
        statement.setInt(4, time);
        statement.executeUpdate();
        message = "success";
    } catch (SQLException ex) {
        System.out.println(ex);
        message = "sql_query_error";
    }
    return message;
}
```

Detalles:

- **Conexión a la base de datos:** El método recibe una conexión **Connection** a la base de datos para ejecutar la consulta.

- **Datos requeridos:** Recibe un mapa **data** con las claves **user_id**, **uv_data**, **date**, y **time**.
- **Resultado:** Inserta los datos en la tabla **exposition_detail** y devuelve un mensaje que indica el éxito o el error de la operación.

List<Map<String, Object>> getExposition(Connection connection, int user_id)

Descripción:

Recupera todos los registros de exposición asociados a un usuario específico.

Código:

```
public static List<Map<String, Object>> getExposition(Connection connection, int user_id) {
    List<Map<String, Object>> expositionData = new ArrayList<>();
    String sql = "SELECT * FROM exposition_detail WHERE user_id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Map<String, Object> data = new HashMap<>();
            data.put("id", resultSet.getInt("id"));
            data.put("user_id", resultSet.getInt("user_id"));
            data.put("uv_data", resultSet.getDouble("uv_data"));
            data.put("date", resultSet.getDate("date"));
            data.put("time", resultSet.getInt("time"));
            expositionData.add(data);
        }
        return expositionData;
    } catch (SQLException ex) {
        return null;
    }
}
```

Detalles:

- **Consulta SQL:** Selecciona todos los registros de la tabla **exposition_detail** donde el **user_id** coincide con el ID proporcionado.
- **Resultado:** Retorna una lista de mapas donde cada mapa contiene los detalles de una exposición, incluyendo el ID del registro, el ID del usuario, los datos UV, la

fecha, y el tiempo de exposición.

Uso en la Aplicación: La clase **ExpositionDetail** es esencial para el manejo de los datos de exposición en la aplicación. Proporciona la funcionalidad necesaria para registrar las exposiciones de los usuarios y recuperar esta información para su análisis o presentación.

Clase User

Descripción: La clase **User** maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) relacionadas con los usuarios en la base de datos. Esta clase permite gestionar la información personal de los usuarios, como nombre, apellido, correo electrónico, contraseña, y edad.

Métodos:

String create(Connection connection, Map<String, String> data)

Descripción:

Inserta un nuevo registro de usuario en la base de datos utilizando la información proporcionada en el mapa **data**.

Código:

```

public static String create(Connection connection, Map<String, String> data)
{
    String message = "";
    String sql = "INSERT INTO users (age, name, last_name,"
        + " email, password) VALUES (?, ?, ?, ?, ?)";

    try {
        int age = Integer.valueOf(data.get("age"));
        String name = data.get("name");
        String last_name = data.get("last_name");
        String email = data.get("email");
        String password = data.get("password");

        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, age);
        statement.setString(2, name);
        statement.setString(3, last_name);
        statement.setString(4, email);
        statement.setString(5, password);
        statement.executeUpdate();
        message = "success_registry";
    } catch (SQLException ex) {
        System.out.println(ex);
        message = "sql_query_error";
    }

    return message;
}

```

Detalles:

- **Conexión a la base de datos:** El método recibe una conexión **Connection** a la base de datos para ejecutar la consulta.
- **Datos requeridos:** Recibe un mapa **data** con las claves **age**, **name**, **last_name**, **email**, y **password**.
- **Resultado:** Inserta los datos en la tabla **users** y devuelve un mensaje que indica el éxito o el error de la operación.

Map<String, Object> getUser(Connection connection, int user_id)

Descripción:

Recupera la información de un usuario específico de la base de datos, identificado por su **user_id**.

Código:

```

public static Map<String, Object> getUser(Connection connection, int user_id) {
    Map<String, Object> userData = new HashMap<>();
    String sql = "SELECT * FROM users WHERE id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        ResultSet resultSet = statement.executeQuery();

        while (resultSet.next()) {
            userData.put("id", resultSet.getInt("id"));
            userData.put("age", resultSet.getInt("age"));
            userData.put("name", resultSet.getString("name"));
            userData.put("last_name", resultSet.getString("last_name"));
            userData.put("email", resultSet.getString("email"));
            userData.put("password", resultSet.getBigDecimal("password"));
        }
        return userData;
    } catch (SQLException ex) {
        System.out.println(ex);
        return null;
    }
}

```

Detalles:

- **Consulta SQL:** Selecciona todos los campos del usuario donde el **id** coincide con el ID proporcionado.
- **Resultado:** Retorna un mapa con la información del usuario, incluyendo el ID, la edad, el nombre, el apellido, el correo electrónico, y la contraseña.

String update(Connection connection, Map<String, String> data)

Descripción:

Actualiza la información de un usuario existente en la base de datos.

Código:


```
public static String update(Connection connection, Map<String, String> data) {  
    String message = "";  
    String sql = "UPDATE users SET age = ?, name = ?, last_name = ?, "  
        + "password = ?, email = ? WHERE id = ?";  
  
    try {  
        int user_id = Integer.valueOf(data.get("user_id"));  
        int age = Integer.valueOf(data.get("age"));  
        String name = data.get("name");  
        String last_name = data.get("last_name");  
        String email = data.get("email");  
        String password = data.get("password");  
  
        PreparedStatement statement = connection.prepareStatement(sql);  
        statement.setInt(1, age);  
        statement.setString(2, name);  
        statement.setString(3, last_name);  
        statement.setString(4, password);  
        statement.setString(5, email);  
        statement.setInt(6, user_id);  
        statement.executeUpdate();  
        message = "success_update";  
    } catch (SQLException ex) {  
        message = "sql_query_error";  
    }  
    return message;  
}
```

Detalles:


- **Datos requeridos:** Recibe un mapa **data** con las claves **user_id**, **age**, **name**, **last_name**, **email**, y **password**.
- **Resultado:** Actualiza los datos del usuario en la tabla **users** y devuelve un mensaje que indica el éxito o el error de la operación.

String delete(Connection connection, int user_id)

Descripción:

Elimina un registro de usuario de la base de datos basado en el **user_id**.

Código:



```

public static String delete(Connection connection, int user_id) {
    String message = "";
    String sql = "DELETE FROM users WHERE id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        statement.executeUpdate();
        message = "success";
    } catch (SQLException ex) {
        message = "sql_query_error";
    }
    return message;
}


```

Detalles:

- **Consulta SQL:** Elimina el usuario cuyo **id** coincide con el **user_id** proporcionado.
- **Resultado:** Retorna un mensaje que indica el éxito o el error de la operación.

String validateLogin(Connection connection, String email, String password)**Descripción:**

Valida las credenciales de inicio de sesión de un usuario.

Código:


```

public static String validateLogin(Connection connection, String email, String password)
{
    String message = "no_info";
    String sql = "SELECT * FROM users WHERE email = ? AND password = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setString(1, email);
        statement.setString(2, password);
        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            message = String.valueOf(resultSet.getInt("id"));
        }
    } catch (SQLException ex) {
        message = "sql_query_error";
    }
    return message;
}

```

Detalles:

- **Validación de credenciales:** Verifica si el correo electrónico y la contraseña proporcionados coinciden con un registro en la base de datos.
- **Resultado:** Retorna el ID del usuario si las credenciales son correctas, o un mensaje de error si no lo son.

String validateUser(Connection connection, int user_id)

Descripción:

Verifica si un usuario existe en la base de datos basado en su **user_id**.

Código:

```

public static String validateUser(Connection connection, int user_id) {
    String message = "";
    String sql = "SELECT * FROM users WHERE id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        ResultSet resultSet = statement.executeQuery();
        message = (resultSet.next()) ? "success" : "no_info";
    } catch (SQLException ex) {
        message = "sql_query_error";
    }
    return message;
}

```

Detalles:

- **Consulta SQL:** Selecciona el usuario cuyo **id** coincide con el **user_id** proporcionado.

- **Resultado:** Retorna un mensaje de éxito si el usuario existe, o un mensaje de error si no existe.

Uso en la Aplicación: La clase **User** es fundamental para gestionar las operaciones relacionadas con los usuarios dentro de la aplicación. Proporciona una interfaz para crear, leer, actualizar y eliminar usuarios, así como para validar las credenciales de inicio de sesión.

Clase UserConfiguration

Descripción: La clase **UserConfiguration** maneja las operaciones CRUD relacionadas con la configuración personalizada de los usuarios en la aplicación. Esta configuración incluye el tiempo de exposición al sol, el tipo de piel, enfermedades relacionadas con la exposición al sol, y si se habilitan las notificaciones API.

Métodos:

String createConfigurationModel(Connection connection, Map<String, String> data)

Descripción:

Inserta una nueva configuración personalizada del usuario en la base de datos utilizando la información proporcionada en el mapa **data**.

Código:

```

public static String createConfigurationModel(Connection connection, Map<String, String> data) {
    String message = "";
    String sql = "INSERT INTO user_configuration (user_id, time_exposition, skin, disease, api_notification)"
        + "VALUES (?, ?, ?, ?, ?)";

    try {
        int user_id = Integer.valueOf(data.get("user_id"));
        int time_exposition = Integer.valueOf(data.get("time_exposition"));
        String skin = data.get("skin");
        String disease = data.get("disease");
        int api_notification = Integer.valueOf(data.get("api_notification"));
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        statement.setInt(2, time_exposition);
        statement.setString(3, skin);
        statement.setString(4, disease);
        statement.setInt(5, api_notification);

        statement.executeUpdate();
        message = "success_config";
    } catch (SQLException ex) {
        System.out.println(ex);
        message = "sql_query_error";
    }

    return message;
}

```

Detalles:

- **Conexión a la base de datos:** El método recibe una conexión **Connection** a la base de datos para ejecutar la consulta.
- **Datos requeridos:** Recibe un mapa **data** con las claves **user_id**, **time_exposition**, **skin**, **disease**, y **api_notification**.
- **Resultado:** Inserta los datos de configuración en la tabla **user_configuration** y devuelve un mensaje que indica el éxito o el error de la operación.

List<Map<String, Object>> getConfiguration(Connection connection, int user_id)

Descripción:

Recupera la configuración personalizada de un usuario específico de la base de datos, identificado por su **user_id**.

Código:

```

public static List<Map<String, Object>> getConfiguration(Connection connection, int user_id) {
    List<Map<String, Object>> configData = new ArrayList<>();
    String sql = "SELECT * FROM user_configuration WHERE user_id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Map<String, Object> data = new HashMap<>();
            data.put("id", resultSet.getInt("id"));
            data.put("user_id", resultSet.getInt("user_id"));
            data.put("time_exposition", resultSet.getInt("time_exposition"));
            data.put("skin", resultSet.getString("skin"));
            data.put("disease", resultSet.getString("disease"));
            data.put("api_notification", resultSet.getInt("api_notification"));
            configData.add(data);
        }
        return configData;
    } catch (SQLException ex) {
        System.out.println(ex);
        return null;
    }
}

```

Detalles:

- **Consulta SQL:** Selecciona todos los campos de la configuración donde el **user_id** coincide con el ID proporcionado.
- **Resultado:** Retorna una lista de mapas que contienen la configuración del usuario, incluyendo el ID de la configuración, el tiempo de exposición, el tipo de piel, la enfermedad relacionada, y si se habilitan las notificaciones API.

String updateConfiguration(Connection connection, Map<String, String> data)

Descripción:

Actualiza la configuración personalizada de un usuario existente en la base de datos.

Código:

```

public static String updateConfiguration(Connection connection, Map<String, String> data) {
    String message = "";
    String sql = "UPDATE user_configuration SET time_exposition = ?, "
        + "skin = ?, disease = ?, api_notification = ? WHERE user_id = ?";

    try {
        int time_exposition = Integer.valueOf(data.get("time_exposition"));
        int api_notification = Integer.valueOf(data.get("api_notification"));
        int user_id = Integer.valueOf(data.get("user_id"));
        String disease = data.get("disease");
        String skin = data.get("skin");

        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, time_exposition);
        statement.setString(2, skin);
        statement.setString(3, disease);
        statement.setInt(4, api_notification);
        statement.setInt(5, user_id);
        statement.executeUpdate();
        message = "success_update";
    } catch (SQLException ex) {
        message = "sql_query_error";
    }
    return message;
}

```

Detalles:

- **Datos requeridos:** Recibe un mapa **data** con las claves **user_id**, **time_exposition**, **skin**, **disease**, y **api_notification**.
- **Resultado:** Actualiza los datos de la configuración del usuario en la tabla **user_configuration** y devuelve un mensaje que indica el éxito o el error de la operación.

String validateConfiguration(Connection connection, int user_id)

Descripción:

Verifica si existe una configuración para un usuario en la base de datos basado en su **user_id**.

Código:

```

public static String validateConfiguration(Connection connection, int user_id) {
    String message = "";
    String sql = "SELECT * FROM user_configuration WHERE user_id = ?";

    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        ResultSet resultSet = statement.executeQuery();
        message = (resultSet.next()) ? "success" : "no_info";
    } catch (SQLException ex) {
        message = "sql_query_error";
    }
    return message;
}

```

Detalles:

- **Consulta SQL:** Selecciona la configuración del usuario cuyo **user_id** coincide con el ID proporcionado.
- **Resultado:** Retorna un mensaje de éxito si la configuración existe, o un mensaje de error si no existe.

Uso en la Aplicación: La clase **UserConfiguration** es crucial para permitir que los usuarios personalicen su experiencia en la aplicación según sus características y necesidades específicas. Proporciona una interfaz para crear, leer, actualizar y validar configuraciones personalizadas, lo que permite un manejo flexible y adaptable de las preferencias del usuario.

Clase UserNotification

Descripción: La clase **UserNotification** maneja las operaciones relacionadas con las notificaciones de usuario en la base de datos. Permite crear, leer, actualizar y eliminar notificaciones, así como recuperar todas las notificaciones para un usuario en particular.

Métodos:

String createNotification(Connection connection, Map<String, String> data)

Descripción:

Inserta una nueva notificación de usuario en la base de datos con la información proporcionada en el mapa **data**.

Código:

```
public static String createNotification(Connection connection, Map<String, String> data) {
    String message = "";

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
    String sql = "INSERT INTO user_notifications (user_id, date, message, state)"
        + "VALUES (?, ?, ?, ?)";

    try {
        int user_id = Integer.valueOf(data.get("user_id"));
        String dateStr = data.get("date");
        String state = data.get("state");
        LocalDate localDate = LocalDate.parse(dateStr, formatter);

        Date sqlDate = Date.valueOf(localDate);
        String text_notification = data.get("message");

        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        statement.setDate(2, sqlDate);
        statement.setString(3, text_notification);
        statement.setString(4, state);
        statement.executeUpdate();
        message = "success";
    } catch (SQLException ex) {
        ex.printStackTrace();
        message = "sql_query_error";
    }

    return message;
}
```

Detalles:

- **Conexión a la base de datos:** El método recibe una conexión **Connection** a la base de datos para ejecutar la consulta.
- **Datos requeridos:** Recibe un mapa **data** con las claves **user_id**, **date**, **message**, y **state**.
- **Resultado:** Inserta la notificación en la tabla **user_notifications** y devuelve un mensaje que indica el éxito o el error de la operación.

List<Map<String, Object>> getAllNotification(Connection connection, int user_id)

Descripción:

Recupera todas las notificaciones asociadas a un usuario específico de la base de datos, identificado por su **user_id**.

Código:

```
public static List<Map<String, Object>> getAllNotification(Connection connection, int user_id) {
    List<Map<String, Object>> notificationData = new ArrayList<>();
    String sql = "SELECT * FROM user_notifications WHERE user_id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, user_id);
        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Map<String, Object> data = new HashMap<>();
            data.put("id", resultSet.getInt("id"));
            data.put("user_id", resultSet.getInt("user_id"));
            data.put("date", resultSet.getDate("date"));
            data.put("message", resultSet.getString("message"));
            data.put("state", resultSet.getString("state"));
            notificationData.add(data);
        }
        return notificationData;
    } catch (SQLException ex) {
        ex.printStackTrace();
        return null;
    }
}
```

Detalles:

- **Consulta SQL:** Selecciona todas las notificaciones donde el **user_id** coincide con el ID proporcionado.
- **Resultado:** Retorna una lista de mapas, cada uno conteniendo los detalles de una notificación.

List<Map<String, Object>> getNotification(Connection connection, int id)

Descripción:

Recupera la notificación específica identificada por su **id** en la base de datos.

Código:

```

public static List<Map<String, Object>> getNotification(Connection connection, int id) {
    List<Map<String, Object>> notificationData = new ArrayList<>();
    String sql = "SELECT * FROM user_notifications WHERE id = ?";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, id);
        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Map<String, Object> data = new HashMap<>();
            data.put("id", resultSet.getInt("id"));
            data.put("user_id", resultSet.getInt("user_id"));
            data.put("date", resultSet.getDate("date"));
            data.put("message", resultSet.getString("message"));
            notificationData.add(data);
        }
        return notificationData;
    } catch (SQLException ex) {
        ex.printStackTrace();
        return null;
    }
}

```

Detalles:

- **Consulta SQL:** Selecciona la notificación donde el **id** coincide con el ID proporcionado.
- **Resultado:** Retorna una lista de mapas con los detalles de la notificación especificada.

String updateNotification(Connection connection, int id, String state)

Descripción:

Actualiza el estado de una notificación en la base de datos.

Código:

```

public static String updateNotification(Connection connection, int id, String state) {
    String message = "";
    String sql = "UPDATE user_notifications SET state = ? WHERE id = ?";

    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setString(1, state);
        statement.setInt(2, id);
        statement.executeUpdate();
        message = "success_update";
    } catch (SQLException ex) {
        message = "sql_query_error";
    }

    return message;
}

```

Detalles:

- **Datos requeridos:** Recibe el **id** de la notificación a actualizar y el nuevo **state**.
- **Resultado:** Actualiza el estado de la notificación en la tabla **user_notifications** y devuelve un mensaje que indica el éxito o el error de la operación.

String deleteNotification(Connection connection, int id)**Descripción:**

Elimina una notificación específica de la base de datos, identificada por su **id**.

Código:

```

public static String deleteNotification(Connection connection, int id) {
    String message = "";
    String sql = "DELETE FROM user_notifications WHERE id = ?";

    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setInt(1, id);
        statement.executeUpdate();
        message = "success_delete";
    } catch (SQLException ex) {
        ex.printStackTrace();
        message = "sql_query_error";
    }

    return message;
}

```

Detalles:

- **Consulta SQL:** Elimina la notificación donde el **id** coincide con el ID proporcionado.
- **Resultado:** Retorna un mensaje que indica el éxito o el error de la operación.
- **String deleteAllNotification(Connection connection)**

Descripción:

Elimina todas las notificaciones de la tabla **user_notifications**.

Código:

```

public static String deleteAllNotification(Connection connection) {
    String message = "";
    String sql = "DELETE FROM user_notifications";
    try {
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.executeUpdate();
        message = "success_delete_all";
    } catch (SQLException ex) {
        ex.printStackTrace();
        message = "sql_query_error";
    }
    return message;
}

```

Detalles:

- **Consulta SQL:** Elimina todas las filas de la tabla **user_notifications**.
- **Resultado:** Retorna un mensaje que indica el éxito o el error de la operación.

Uso en la Aplicación: La clase **UserNotification** es crucial para manejar las notificaciones de usuario dentro de la aplicación. Permite la creación, actualización, eliminación y consulta de notificaciones, asegurando que los usuarios reciban la información relevante en el momento

adecuado.

Montaje del Proyecto en Otro Equipo

1. Requisitos Previos

Antes de iniciar el proceso de montaje del proyecto en un nuevo equipo, asegúrate de cumplir con los siguientes requisitos:

- **Git:** Herramienta de control de versiones para clonar el repositorio del proyecto. [Descargar Git](#).
- **NetBeans IDE:** Entorno de desarrollo integrado (IDE) para correr y modificar el proyecto. [Descargar NetBeans](#).
- **XAMPP:** Paquete de software que incluye MySQL, necesario para manejar la base de datos. [Descargar XAMPP](#).

2. Clonación del Repositorio

Para obtener una copia local del proyecto, sigue estos pasos:

1. Abre una terminal o línea de comandos en tu sistema.
2. Navega al directorio donde deseas clonar el proyecto.
3. Ejecuta el siguiente comando para clonar el repositorio:

```
git clone https://github.com/valentinacardona07/Proyecto_UV_Alert.git
```

4. Una vez clonado, abre el proyecto en NetBeans IDE.

3. Configuración de la Base de Datos

Para configurar la base de datos utilizando la línea de comandos (cmd):

1. Instalar y Configurar XAMPP:

- Inicia XAMPP y asegúrate de que el servidor MySQL esté funcionando.

2. Acceder a MySQL desde la Línea de Comandos:

- Abre la terminal de comandos (cmd) en tu sistema.
- Navega a la carpeta donde tienes instalado XAMPP (por ejemplo, **C:\xampp\mysql\bin**).
- Inicia el cliente de MySQL con el siguiente comando:

```
mysql -u root -p
```

- Si se te solicita, ingresa la contraseña de root (por defecto, puede estar en blanco).

3. Crear la Base de Datos:

- Una vez dentro del cliente de MySQL, crea la base de datos **uv_alert** con el siguiente comando:

```
CREATE DATABASE uv_alert;
```

- Usa la base de datos recién creada:

```
USE uv_alert;
```


4. Crear las Tablas:

- Con la base de datos seleccionada, ejecuta los siguientes comandos para crear las tablas necesarias:

```
CREATE TABLE users (  
  
    id INT AUTO_INCREMENT PRIMARY KEY,  
  
    age INT,  
  
    name VARCHAR(255),  
  
    last_name VARCHAR(255),  
  
    email VARCHAR(255) UNIQUE NOT NULL,  
  
    password VARCHAR(255) NOT NULL  
  
);
```

```
CREATE TABLE user_configuration (  
  
    id INT AUTO_INCREMENT PRIMARY KEY,  
  
    user_id INT NOT NULL,  
  
    skin VARCHAR(255),  
  
    time_exposition INT NOT NULL,  
  
    disease VARCHAR(255) NOT NULL,
```

api_notification TINYINT(1) NOT NULL

);

CREATE TABLE user_notifications (

id INT AUTO_INCREMENT PRIMARY KEY,

user_id INT NOT NULL,

date TIMESTAMP NOT NULL,

message TEXT NOT NULL,

state VARCHAR(255)

);

CREATE TABLE exposition_detail (

id INT AUTO_INCREMENT PRIMARY KEY,

user_id INT NOT NULL,

uv_data DECIMAL(10, 2) NOT NULL,

time DECIMAL(10, 2) NOT NULL,

date TIMESTAMP NOT NULL

);

4. Configuración en NetBeans

1. Abre el proyecto en NetBeans.
2. Verifica que las configuraciones de conexión a la base de datos estén correctas:
 - Asegúrate de que los parámetros de conexión (host, puerto, nombre de la base de datos, usuario y contraseña) en los archivos de configuración del proyecto coincidan con los de tu entorno local.

5. Ejecución del Proyecto

Una vez configurado todo, sigue estos pasos para ejecutar el proyecto:

1. Abre NetBeans y selecciona el proyecto.
2. Haz clic en el botón "Run" para iniciar la aplicación.
3. La aplicación debería conectarse automáticamente a la base de datos y estar lista para su uso.

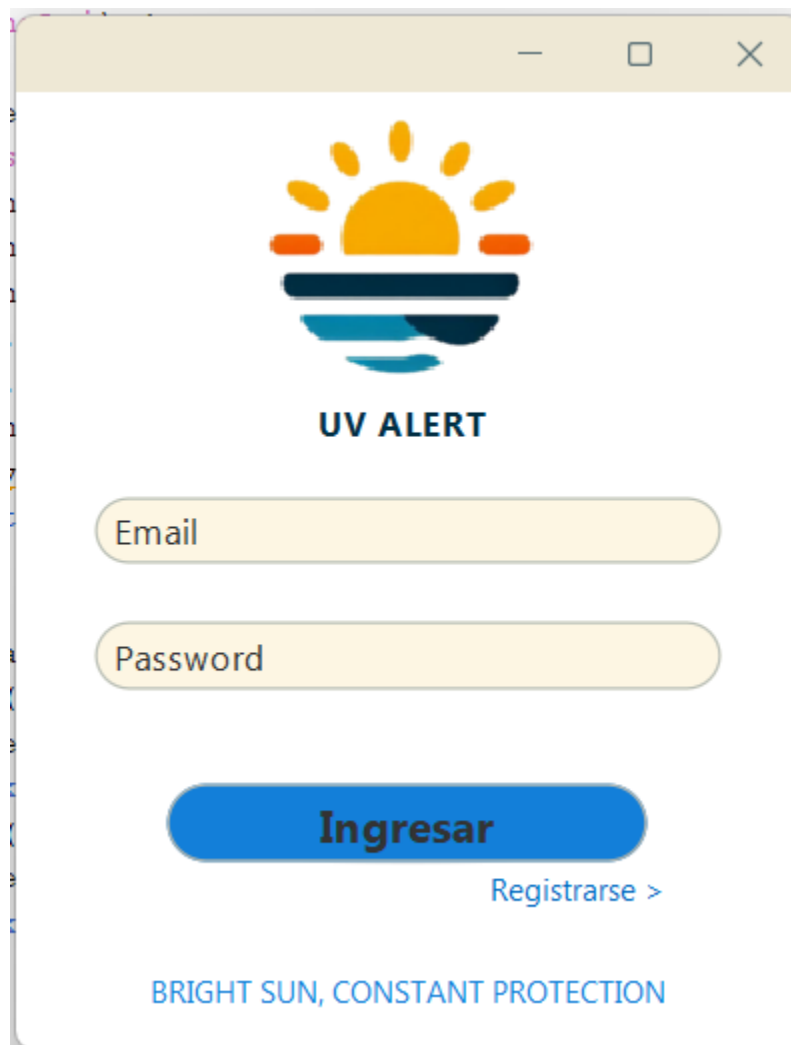
6. Consideraciones Adicionales

- **Configuración del Firewall:** Asegúrate de que el firewall no esté bloqueando el puerto de MySQL (3306 por defecto) si estás trabajando en una red compartida.
- **Backups:** Realiza copias de seguridad regulares tanto del código como de la base de datos antes de realizar cambios importantes.

Módulos de Interacción y Funcionalidades

Pantalla de Inicio de Sesión

Al ejecutar la aplicación, el usuario es recibido por una pantalla de inicio de sesión (**LoginView**). Aquí, el usuario debe ingresar su correo electrónico y contraseña para acceder a su cuenta. Si el correo o la contraseña no cumplen con los requisitos, o si no coinciden, se mostrará un mensaje de error en una ventana de alerta, guiando al usuario a corregir la información ingresada.



UV ALERT

Email

Password

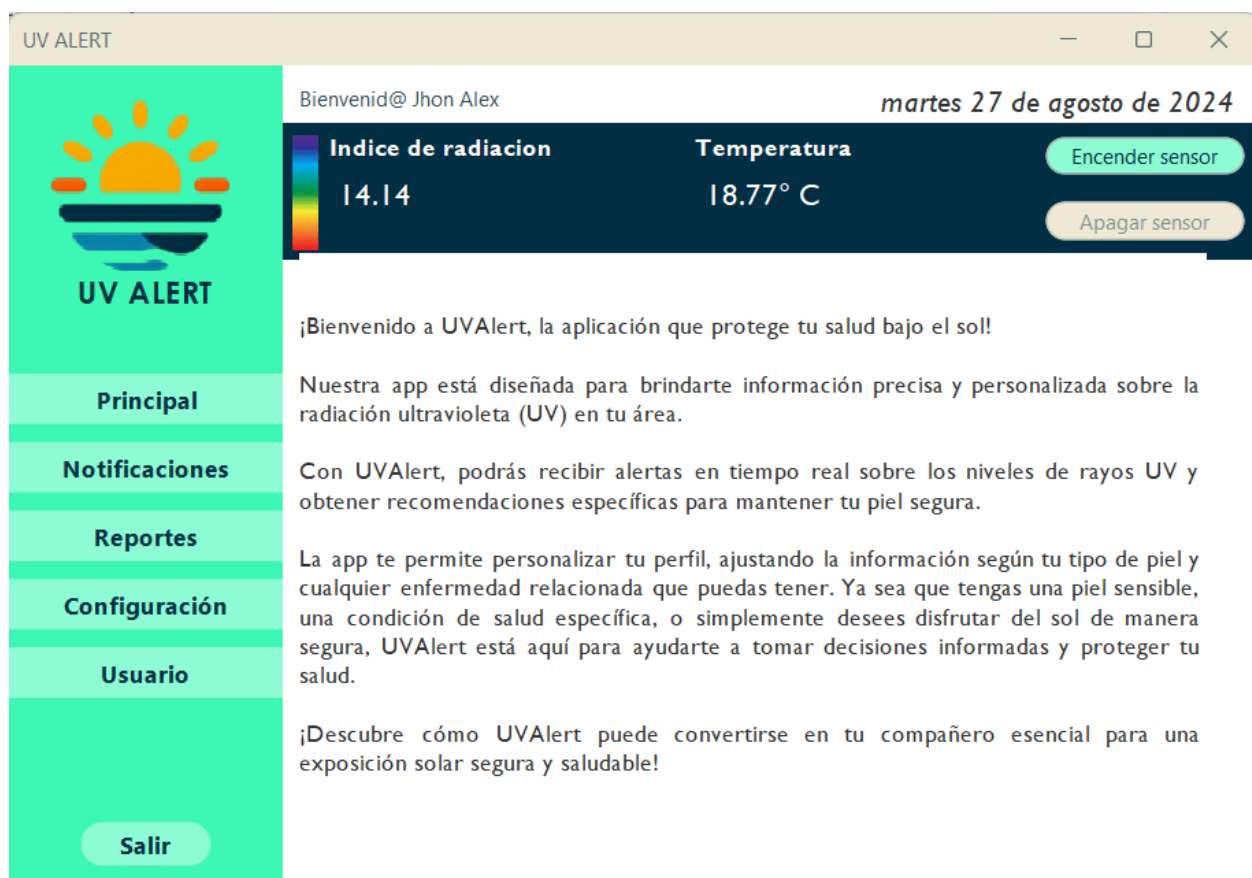
Ingresar

[Registrarse >](#)

BRIGHT SUN, CONSTANT PROTECTION

Panel de Control (DashboardView)

Una vez que el usuario ha iniciado sesión, es dirigido al panel de control (**DashboardView**). Esta vista muestra la fecha actual en la esquina superior derecha y un mensaje de bienvenida personalizado en la esquina superior izquierda. La parte central de la pantalla contiene información relevante sobre los niveles actuales de radiación UV y la temperatura. Los usuarios pueden optar por encender o apagar el sensor de radiación UV utilizando los botones proporcionados en la parte derecha del panel.



Notificaciones Personalizadas

La aplicación revisa continuamente los niveles de radiación UV y compara esta información con la configuración del usuario, como su tipo de piel y cualquier condición de salud existente. Si se detecta un nivel de radiación UV que supera el límite seguro, el usuario recibirá una notificación personalizada. Esta notificación aparecerá en una ventana emergente (**AlertView**), y también se registrará en el panel de notificaciones (**NotificationView**), al que se puede acceder desde el menú lateral.

UV ALERT

Bienvenid@ Jhon Alex martes 27 de agosto de 2024

Indice de radiacion 14.14 **Temperatura** 18.77° C

Encender sensor Apagar sensor

Id	Fecha	Mensaje	Estado	Acciones
1	2024-08-27	Alerta por	leido	Acción

Principal Notificaciones Reportes Configuración Usuario Salir

REFRESCAR ELIMINAR ELIMINAR TODO

Configuración del Usuario (ConfigurationView)

Los usuarios pueden personalizar su configuración accediendo a la vista de configuración (**ConfigurationView**). Aquí, pueden ajustar su tipo de piel, cualquier condición de salud que pueda afectar su exposición al sol, y la cantidad de tiempo que planean pasar al aire libre. También pueden activar o desactivar las notificaciones automáticas basadas en la API del clima.



The screenshot shows the 'ConfigurationView' of the 'UV ALERT' application. The interface is divided into a sidebar on the left and a main content area on the right. The sidebar contains a logo with a sun and the text 'UV ALERT', and a list of menu items: 'Principal', 'Notificaciones', 'Reportes', 'Configuración' (highlighted), and 'Usuario'. At the bottom of the sidebar is a 'Salir' button. The main content area has a header with the user's name 'Bienvenid@ Jhon Alex' and the date 'martes 27 de agosto de 2024'. Below the header, there are two sections: 'Indice de radiacion' showing '14.14' with a color scale bar, and 'Temperatura' showing '18.77° C'. To the right of these sections are two buttons: 'Encender sensor' and 'Apagar sensor'. Below these, there are two sections: 'Tiempo de exposición (min)' with a text input field, and 'Notificaciones' with a radio button for 'Radiacion general'. Below these, there is a section 'Enfermedades o condiciones' with three columns of radio buttons: 'Ninguna', 'Cáncer de piel', 'Fotoenvejecimiento', 'Dermatitis actínica crónica', 'Lupus Eritematoso Cutáneo', 'Fotodermatosis', and 'Quemaduras'. Below this, there is a section 'Fototipo de Piel' with two columns of radio buttons: 'Fototipo I', 'Fototipo II', 'Fototipo III', 'Fototipo IV', 'Fototipo V', and 'Fototipo VI'. At the bottom right of the main content area is a large blue 'GUARDAR' button.

Reportes de Exposición UV (ReportView)

La aplicación también permite a los usuarios visualizar un historial de su exposición a los rayos UV a través de la vista de reportes (**ReportView**). Esta vista muestra gráficamente cómo han

fluctuado los niveles de radiación UV durante un período de tiempo específico, lo que ayuda al usuario a tomar decisiones informadas sobre su exposición al sol.

Actualización de Datos de Usuario (UserView)

En la sección de "Usuario", los usuarios tienen la opción de actualizar su información personal, incluyendo su nombre, apellido, correo electrónico, y contraseña. La vista muestra un formulario donde el usuario puede modificar estos datos de manera sencilla. Los campos incluyen:

- **Nombre y Apellido:** El usuario puede actualizar su nombre y apellido para reflejar cualquier cambio necesario.
- **Correo Electrónico:** Es posible cambiar el correo electrónico asociado a la cuenta del usuario. Este cambio requiere confirmación ingresando nuevamente el correo electrónico en el campo de "Confirmar correo".
- **Contraseña:** Para mantener la seguridad, cualquier cambio en la contraseña también requiere que se confirme la nueva contraseña en un campo separado.

Una vez que el usuario ha ingresado los datos deseados, puede optar por guardar los cambios, o cancelar la operación si decide no modificar la información. Los botones correspondientes para estas acciones son "Editar", "Guardar", y "Cancelar".

Esta sección es esencial para mantener la información del usuario actualizada y segura, asegurando que solo el usuario autorizado tenga acceso a su cuenta y a las recomendaciones

personalizadas que ofrece la aplicación.

The screenshot displays the UV ALERT application interface. On the left is a green sidebar menu with the following options: Principal, Notificaciones, Reportes, Configuración, Usuario, and a Salir button at the bottom. The main content area has a dark blue header with a sun icon and the text 'UV ALERT'. Below the header, the user is logged in as 'Bienvenid@ Jhon Alex' on 'martes 27 de agosto de 2024'. The header also displays 'Indice de radiacion' as 14.14 and 'Temperatura' as 18.77° C, with buttons for 'Encender sensor' and 'Apagar sensor'. The main section is titled 'Actualizar Datos' and contains a form with the following fields: 'Nombre *' (Jhon Alex), 'Apellido *' (Riascos Borja), 'Contraseña *' (masked with dots), 'Confirmar contraseña *' (masked with dots), 'Correo *' (usa@gmail.com), and 'Confirmar correo *' (usa@gmail.com). At the bottom of the form are three buttons: 'EDITAR' (blue), 'GUARDAR' (yellow), and 'CANCELAR' (yellow).

Salir de la Aplicación

Para cerrar la aplicación, el usuario puede seleccionar la opción "Salir" en el menú lateral, lo que detendrá cualquier proceso en ejecución, como el monitoreo de los sensores, y cerrará la sesión del usuario.

Lista de referencias

* *Java Platform SE 7*. (2000). API JAVA. <https://docs.oracle.com/javase/7/docs/api/>