

Criterio C: Desarrollo

Estructura del Proyecto

La aplicación *FoodMatch* está organizada en múltiples archivos Java distribuidos según su funcionalidad, respetando los principios de programación orientada a objetos (POO).

Interfaz gráfica de usuario

Toda la interfaz se ha implementado con **Java Swing**, separando cada ventana en una clase independiente. Por ejemplo, *PantallaPrincipal.java* contiene la ventana principal desde donde el usuario accede a todas las funciones. *InicioDeSesion.java* y *Registro.java* controlan el acceso al sistema, mientras que *Perfil.java* y *Encuesta.java* permiten configurar preferencias alimentarias.

Otras ventanas destacadas son *Buscar.java* y *BuscadorRecetas.java*, que gestionan la búsqueda de recetas, *VentanaReceta.java* para visualizar su contenido y *AñadirReceta.java* para crear nuevas. Esta organización permite un diseño modular y claro.

Lógica y control de la aplicación

El control de flujo se gestiona desde clases como *BusquedaInteligente.java*, que permite sugerir recetas según los ingredientes disponibles, y *BuscadorRecetas.java*, que ofrece búsqueda por texto. *Sesion.java* mantiene la sesión activa del usuario, y *BaseDeDatos.java* centraliza el acceso a la base de datos, separando lógica de presentación y almacenamiento.

Clases orientadas a datos y estructura del modelo

El diseño orientado a objetos se implementa en clases como *ClaseUsuario.java*, *Usuario.java*, *ClaseReceta.java*, *Receta.java*, *Alimento.java* y *AlimentoDisponible.java*, que encapsulan los elementos principales del sistema. Por ejemplo, *Receta.java* gestiona ingredientes, dificultad, tiempo, imagen y valores nutricionales. *AlimentoDisponible.java* representa los ingredientes disponibles en casa, clave para el filtrado inteligente. Este modelo modular facilita la escalabilidad y reutilización del código.



Figura 1: Archivos de FoodMatch

Conexión con base de datos (MySQL)

La persistencia se realiza mediante una base de datos MySQL. La clase BaseDeDatos.java contiene los métodos de conexión y consulta, utilizando la librería mysql-connector-j-9.1.0.jar y JDBC con consultas preparadas para garantizar eficiencia y seguridad.

Las funciones incluyen el registro de usuarios, almacenamiento de alimentos prohibidos, consultas filtradas, valoraciones y recuperación de imágenes.

Inserción y gestión de imágenes

El sistema permite asociar imágenes a recetas. Las clases Cargalmagen.java e InsertarImágenesRecetas.java procesan imágenes locales y las vinculan a recetas en la base de datos y la interfaz.

Las imágenes se almacenan en carpetas específicas, lo que permite su visualización posterior y reutilización eficiente.

Carpetas del proyecto

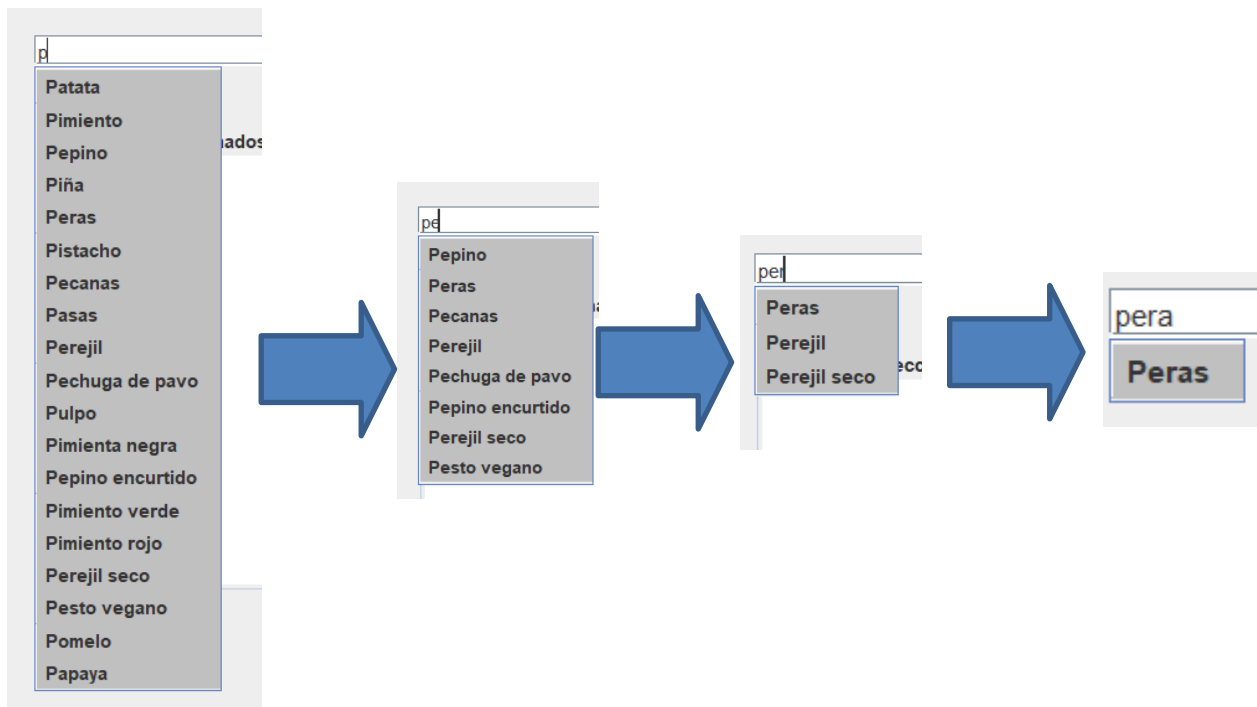
El proyecto incluye:

- imagen_pantallaprincipal: imágenes estáticas de la interfaz inicial.
- imagen_receta: imágenes asociadas a recetas de usuarios.
- .vscode: configuraciones del entorno de desarrollo.

Técnicas utilizadas en el proyecto

Técnica 1: Sugerencias dinámicas mientras se escribe

La aplicación incorpora un sistema de sugerencias automáticas que se actualizan en tiempo real mientras el usuario escribe en un campo de búsqueda. Esta funcionalidad, implementada en AlimentosProhibidos.java, permite al usuario introducir alimentos que desea excluir de su dieta. A medida que se escribe, se despliega un menú contextual con coincidencias filtradas.



El sistema utiliza un `DocumentListener` asociado al `JTextField`, que detecta cualquier cambio en el contenido (escritura, borrado o reemplazo) y activa el método `actualizarSugerencias()`.

```
//escucha cambios en el campo de búsqueda
buscarField.getDocument().addDocumentListener(new javax.swing.event.DocumentListener() {
    @Override public void insertUpdate(javax.swing.event.DocumentEvent e) { actualizarSugerencias(); }
    @Override public void removeUpdate(javax.swing.event.DocumentEvent e) { actualizarSugerencias(); }
    @Override public void changedUpdate(javax.swing.event.DocumentEvent e) { actualizarSugerencias(); }
});
```

Este método recorre la lista de alimentos disponibles y filtra aquellos cuyo nombre comienza con el texto introducido. Además, aplica condiciones adicionales:

```
//muestra sugerencias en el menú desplegable
private void actualizarSugerencias() {
    String texto = buscarField.getText().toLowerCase();
    sugerenciasMenu.setVisible(b:false);
    sugerenciasMenu.removeAll();

    if (!texto.isEmpty() && !texto.equals(anObject:"Introducir alimento")) {
        for (Alimento alimento : alimentos) {
            String nombre = alimento.getNombre().toLowerCase();

            //evitar duplicados
            boolean yaSeleccionado = false;
            for (Alimento s : seleccionados) {
                if (s.getNombre().equalsIgnoreCase(alimento.getNombre())) {
                    yaSeleccionado = true;
                    break;
                }
            }

            //filtrar según dieta del usuario
            boolean incompatible =
                (usuario.isVegano() && !alimento.isVegano()) ||
                (usuario.isVegetariano() && !alimento.isVegetariano()) ||
                (usuario.isCeliaco() && !alimento.isCeliaco());

            //añadir sugerencia válida al menú
            if (nombre.startsWith(texto) && !yaSeleccionado && !incompatible) {
                JMenuItem alimentoSugerido = new JMenuItem(alimento.getNombre());
                alimentoSugerido.setBackground(Color.LIGHT_GRAY);
                alimentoSugerido.addActionListener(e -> agregarAlimento(alimento));
                sugerenciasMenu.add(alimentoSugerido);
            }
        }

        if (sugerenciasMenu.getComponentCount() > 0) {
            Point location = buscarField.getLocationOnScreen();
            sugerenciasMenu.show(this,
                location.x - this.getLocationOnScreen().x,
                location.y - this.getLocationOnScreen().y + buscarField.getHeight());
            buscarField.requestFocusInWindow();
        }
    }
}
```

- Evita mostrar alimentos ya seleccionados.
- Excluye alimentos incompatibles con las restricciones alimentarias del usuario (vegano, vegetariano o celíaco).
- Muestra los resultados en un JPopupMenu con elementos JMenuItem seleccionables, que permiten añadir rápidamente el alimento a la selección.

Técnica 2: Uso de SwingWorker para evitar bloqueos en la interfaz (carga asincrónica de imágenes)

Para evitar bloqueos en la interfaz gráfica durante tareas pesadas, la aplicación utiliza SwingWorker para cargar imágenes desde la base de datos en segundo plano. Esta técnica está implementada en CargImagen.java.

```
//deja la etiqueta vacía mientras se carga la imagen
label.setIcon(icon:null);

//usa un swingworker para cargar la imagen sin bloquear la interfaz
new SwingWorker<ImageIcon, Void>() {
    @Override
    protected ImageIcon doInBackground() {
        try {
            //obtiene la imagen desde la base de datos como array de bytes
            byte[] imagenBytes = claseReceta.obtenerImagenReceta(recetaId);
            if (imagenBytes != null) {
                //escala la imagen al tamaño deseado y la devuelve como icono
                Image img = new ImageIcon(imagenBytes).getImage()
                    .getScaledInstance(ancho, alto, Image.SCALE_SMOOTH);
                return new ImageIcon(img);
            }
        } catch (Exception e) {
        }
        return null;
    }

    @Override
    protected void done() {
        try {
            //cuando la imagen ya está lista, se muestra en la etiqueta
            ImageIcon icon = get();
            if (icon != null) {
                label.setIcon(icon);
            }
        } catch (Exception e) {
        }
    }
}.execute(); //ejecuta el swingworker
```

El método `doInBackground()` recupera y procesa la imagen (escalado y conversión a `ImageIcon`) fuera del hilo principal de la interfaz. Una vez completada, el método `done()` actualiza la etiqueta (`JLabel`) correspondiente desde el Event Dispatch Thread.

Este enfoque evita congelamientos visuales al mostrar imágenes y garantiza una experiencia de usuario fluida y receptiva.

Técnica 3: Filtro inteligente de recetas a partir de ingredientes disponibles

Esta técnica permite mostrar solo aquellas recetas que el usuario puede preparar con los ingredientes que tiene en casa, considerando además las cantidades, las unidades de medida y posibles sustituciones como la unidad “a ojo”. El método central que implementa esta funcionalidad es `buscarRecetasConResultado()`, ubicado en la clase `ClaseReceta`.

A continuación se muestra el método completo, comentado y justificado paso a paso:

```
//método que devuelve recetas que pueden prepararse con los ingredientes disponibles del usuario
public List<ResultadoBusqueda> buscarRecetasConResultado(List<AlimentoDisponible> disponibles) throws SQLException {
    List<ResultadoBusqueda> resultados = new ArrayList<>(); //lista que almacenará las recetas válidas
    List<Receta> todas = buscarRecetasPorNombreOCreador(texto:""); //obtiene todas las recetas desde la base de datos
```

Se inicializan las listas que almacenarán las recetas compatibles. Se consultan todas las recetas existentes en la base de datos.

```
for (Receta r : todas) {
    List<AlimentoDisponible> ingredientesNecesarios = obtenerIngredientesDeReceta(r.getId()); //ingredientes requeridos para esta receta
    boolean cantidadesSuficientes = true; //bandera para saber si se cumplen todos los requisitos
```

Se recorren las recetas una por una y se recuperan sus ingredientes. Se usa una bandera `cantidadesSuficientes` para controlar si esa receta es válida.

```
for (AlimentoDisponible ingNecesario : ingredientesNecesarios) {
    String nombreReq = ingNecesario.getAlimento().getNombre().toLowerCase().trim(); //nombre del ingrediente requerido
    double cantidadReq = ingNecesario.getCantidad(); //cantidad requerida
    String unidadReq = ingNecesario.getUnidad().toLowerCase().trim(); //unidad requerida
    boolean encontrado = false; //bandera para saber si el usuario tiene este ingrediente
```

Se recorren los ingredientes requeridos y se preparan variables auxiliares para comparar nombre, cantidad y unidad.

```
for (AlimentoDisponible ad : disponibles) {
    String nombreUsuario = ad.getAlimento().getNombre().toLowerCase().trim(); //nombre del ingrediente disponible

    if (nombreReq.equals(nombreUsuario)) { //coincidencia por nombre
        double cantidadUsuario = ad.getCantidad(); //cantidad disponible
        String unidadUsuario = ad.getUnidad().toLowerCase().trim(); //unidad disponible
```

Se compara si el usuario tiene un ingrediente con el mismo nombre. Si es así, se analiza si su cantidad y unidad son adecuadas.

```

boolean unidadRecetaEsComodin = unidadReq.contains(s:"ojo"); //la receta acepta unidad "a ojo"
boolean unidadUsuarioEsComodin = unidadUsuario.contains(s:"ojo"); //el usuario ha indicado unidad "a ojo"

//si la unidad de la receta es "a ojo", automáticamente se acepta
if (unidadRecetaEsComodin) {
    encontrado = true;
    break;
}

```

Se permite que la unidad "a ojo" actúe como comodín y se acepte sin validación de cantidad.

```

//verifica compatibilidad entre unidades
boolean unidadCompatible =
    unidadUsuario.equals(unidadReq) //mismas unidades
    || (unidadUsuario.equals(anObject:"unidad") && (unidadReq.equals(anObject:"g") || unidadReq.equals(anObject:"ml"))) //usu
    || ((unidadUsuario.equals(anObject:"g") || unidadUsuario.equals(anObject:"ml")) && unidadReq.equals(anObject:"unidad")); /

//si no son compatibles y no es "a ojo", se ignora este ingrediente disponible
if (!unidadCompatible && !unidadUsuarioEsComodin) {
    continue;
}

```

Se evalúa si las unidades son compatibles (por ejemplo, 1 unidad ≈ 150g). Si no lo son, se descarta.

```

//ajusta las cantidades si las unidades no coinciden
double cantidadUsuarioConvertida = cantidadUsuario;
double cantidadReqConvertida = cantidadReq;

//conversión de "unidad" a gramos/ml (estimado como 150)
if (unidadUsuario.equals(anObject:"unidad") && (unidadReq.equals(anObject:"g") || unidadReq.equals(anObject:"ml"))) {
    cantidadUsuarioConvertida = cantidadUsuario * 150;
} else if ((unidadUsuario.equals(anObject:"g") || unidadUsuario.equals(anObject:"ml")) && unidadReq.equals(anObject:"unidad")) {
    cantidadReqConvertida = cantidadReq * 150;
}

```

Se convierten las cantidades entre unidades distintas utilizando una equivalencia aproximada de 150 gramos por unidad.

```

//si el usuario no indicó "a ojo", comparamos cantidades
if (!unidadUsuarioEsComodin) {
    if (cantidadUsuarioConvertida < cantidadReqConvertida) {
        cantidadesSuficientes = false; //no hay suficiente cantidad
    }
}

encontrado = true; //se encontró un ingrediente compatible
break;

//si el ingrediente no fue encontrado, no se puede preparar la receta
if (!encontrado) {
    cantidadesSuficientes = false;
    break;
}
}

```

Se establece si la receta puede ser cocinada con los ingredientes disponibles. Si falta alguno, se descarta completamente.

```

        //si todos los ingredientes están presentes y en cantidad suficiente, se añade la receta al resultado
        if (cantidadesSuficientes) {
            resultados.add(new ResultadoBusqueda(r, completa:true)); //la receta es válida para preparar
        }
    }

    return resultados; //devuelve la lista de resultados de búsqueda
}

```

Finalmente, se devuelven solo las recetas aptas.

Apoyo estructural

El proceso se apoya en funciones auxiliares como obtenerIngredientesDeReceta(), que recupera ingredientes y cantidades desde la base de datos, y la clase ResultadoBusqueda, que encapsula el resultado final indicando si la receta es apta.

```

//clase que representa el resultado de una búsqueda inteligente
public class ResultadoBusqueda {
    private Receta receta;
    private boolean completa;

    //constructor que recibe la receta encontrada y si está completa
    public ResultadoBusqueda(Receta receta, boolean completa) {
        this.receta = receta;
        this.completa = completa;
    }

    //devuelve la receta asociada al resultado
    public Receta getReceta() {
        return receta;
    }

    //indica si se puede preparar la receta con los ingredientes disponibles
    public boolean isCompleta() {
        return completa;
    }
}

```

Además, se incorpora una validación adicional mediante el método esAptaPara(...), que comprueba:

- La compatibilidad de los atributos dietéticos (vegano, vegetariano, celíaco).
- La ausencia de ingredientes prohibidos definidos por el usuario.

El siguiente código muestra el método completo:

```
//método que verifica si una receta es apta para un usuario según su dieta y alimentos prohibidos
public boolean esAptaPara(Receta receta, Usuario usuario, ClaseAlimento claseAlimento) throws SQLException {
    //verifica restricciones dietéticas del usuario
    if (usuario.isVegano() && !receta.isVegano()) return false;
    if (usuario.isVegetariano() && !receta.isVegetariano()) return false;
    if (usuario.isCeliaco() && !receta.isCeliaco()) return false;

    //obtiene los ingredientes de la receta
    List<AlimentoDisponible> ingredientes = obtenerIngredientesDeReceta(receta.getId());
    //obtiene la lista de alimentos prohibidos para el usuario
    List<Alimento> prohibidos = claseAlimento.obtenerAlimentosProhibidos(usuario.getId());

    Set<String> nombresProhibidos = new HashSet<>();
    for (Alimento a : prohibidos) {
        nombresProhibidos.add(a.getNombre().toLowerCase().trim()); //normaliza nombres para comparación
    }

    for (AlimentoDisponible ing : ingredientes) {
        String nombre = ing.getAlimento().getNombre().toLowerCase().trim();
        if (nombresProhibidos.contains(nombre)) return false; //si hay un ingrediente prohibido, no es apta
    }

    return true; //si pasó todas las comprobaciones, es apta
}
```

Este enfoque garantiza que solo se muestren recetas personalizadas, viables y alineadas con las preferencias y restricciones del usuario.

Técnica 4: Inserción automatizada de imágenes en la base de datos:

Para enriquecer visualmente cada receta, la aplicación incorpora un sistema que permite asociar imágenes a las recetas directamente desde una carpeta local. Esto se gestiona a través de la clase InsertarImágenesRecetas.java.

```
public class InsertarImágenesRecetas {

    private static final String CARPETA_IMAGENES = "imagen_receta"; //carpeta que contiene las imágenes .png

    public static void insertarImágenes(Connection connection) {
        try {

            //ajusta el tamaño máximo de paquete para permitir imágenes grandes
            try (Statement stmt = connection.createStatement()) {
                stmt.execute(sql:"SET GLOBAL max_allowed_packet = 33554432");
            } catch (SQLException e) {
            }

            File carpeta = new File(CARPETA_IMAGENES); //accede a la carpeta de imágenes
            if (!carpeta.exists() || !carpeta.isDirectory()) {
                return; //sale si la carpeta no existe o no es válida
            }

            File[] archivos = carpeta.listFiles((dir, name) -> name.toLowerCase().endsWith(suffix:".png")); //filtra solo archivos .png
            if (archivos == null || archivos.length == 0) {
                return; //sale si no hay imágenes
            }

            //recorre cada imagen en la carpeta
            for (File imagen : archivos) {
                String nombreArchivo = imagen.getName().replace(target:".png", replacement:""); //elimina extensión
                String nombreReceta = nombreArchivo.replace(target:"_", replacement:" ").toLowerCase(); //formatea nombre para coincidir con bd

                byte[] imagenBytes = Files.readAllBytes(imagen.toPath()); //lee la imagen como bytes

                String sql = "UPDATE RECETA SET imagen = ? WHERE LOWER(nombre) = ?";
                try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                    stmt.setBytes(parameterIndex:1, imagenBytes); //asigna imagen como parámetro
                    stmt.setString(parameterIndex:2, nombreReceta); //asigna nombre formateado como parámetro
                    stmt.executeUpdate(); //ejecuta la actualización
                } catch (SQLException ignored) {} //ignora si alguna imagen no se puede insertar
            }
        } catch (Exception ignored) {} //ignora cualquier excepción general
    }
}
```

El método insertarImágenes() se encarga de recorrer automáticamente una carpeta del sistema llamada imagen_receta, localizar archivos .png, leerlos como arrays de bytes (byte[]), y guardarlos en la base de datos como blobs binarios.

```
String sql = "UPDATE RECETA SET imagen = ? WHERE LOWER(nombre) = ?";
try (PreparedStatement stmt = connection.prepareStatement(sql)) {
    stmt.setBytes(parameterIndex:1, imagenBytes); //asigna imagen como parámetro
    stmt.setString(parameterIndex:2, nombreReceta); //asigna nombre formateado como parámetro
    stmt.executeUpdate(); //ejecuta la actualización
} catch (SQLException ignored) {} //ignora si alguna imagen no se puede insertar
```

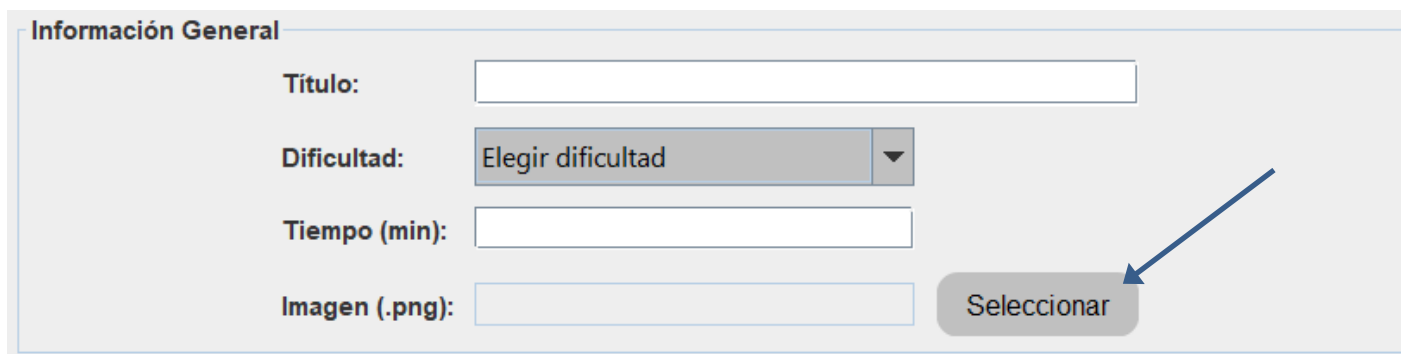
Para evitar errores, también se ajusta el tamaño máximo de paquete de MySQL:

```
//ajusta el tamaño máximo de paquete para permitir imágenes grandes
try (Statement stmt = connection.createStatement()) {
    stmt.execute(sql:"SET GLOBAL max_allowed_packet = 33554432");
} catch (SQLException e) {
}
```

Este sistema automatiza el vínculo entre imágenes locales y recetas, eliminando la carga manual.

Inclusión de imagen de una nueva receta

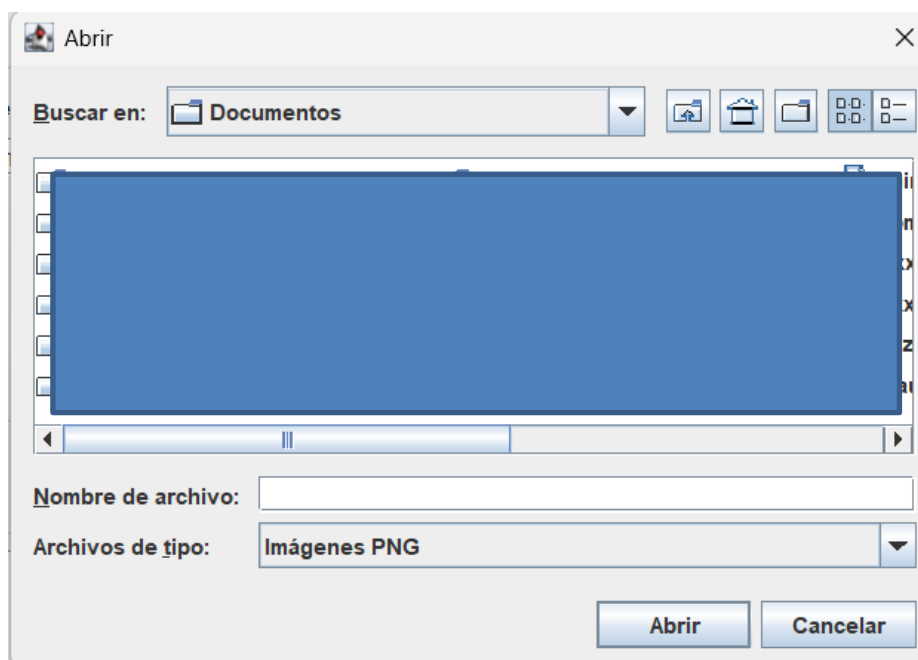
Al crear una receta se puede ver claramente como añadir una imagen y se abre un FileChooser con un filtro automático de PNG. Se oobtiene la ruta de imagen y se pasa a Bytes de la misma forma.



Formulario de Información General:

- Título:
- Dificultad:
- Tiempo (min):
- Imagen (.png):

Una flecha azul apunta al botón "Seleccionar".



Herramientas y recursos técnicos utilizados

Durante el desarrollo de la aplicación, se han utilizado diversas herramientas y librerías que permiten una estructura funcional, modular y escalable.

- **Entorno de desarrollo:** El proyecto se desarrolló íntegramente en Visual Studio Code, que facilitó la edición simultánea de múltiples clases Java y su organización en carpetas.
- **Base de datos:** Se utilizó una base de datos relacional **MySQL**, gestionada con **phpMyAdmin**. La conexión con Java se estableció mediante **JDBC** usando la librería `mysql-connector-j-9.1.0.jar`.
- **Inicialización de la conexión:**
La clase `BasesDeDatos.java` gestiona la configuración y apertura de la conexión. Desde `InicioDeSesion.java`, el sistema comprueba si ya existe una conexión mediante la clase `Sesion`. Si no, se crea una nueva.

```
//constructor
public InicioDeSesion(String inTitul) {
    super(inTitul);
    try {
        this.inicialConnection = DriverManager.getConnection(url:"jdbc:mysql://localhost:3306/", user:"root", password:"");
        try (Statement stmt = inicialConnection.createStatement()) {
            stmt.execute(sql:"SET GLOBAL max_allowed_packet = 33554432");
        } catch (SQLException e) {
        }

        if (!baseDeDatosCreada) {
            new BasesDeDatos(inicialConnection); //solo se crea bd si no se ha creado aún
            baseDeDatosCreada = true;
        }

        this.connection = DriverManager.getConnection(url:"jdbc:mysql://localhost:3306/FoodMatch", user:"root", password:"");

        if (!imagenesInsertadas) {
            InsertarImagenesRecetas.insertarImagenes(connection);
            imagenesInsertadas = true;
        }

        this.claseUsuario = new ClaseUsuario(this.inicialConnection);
    } catch (SQLException e) {
    }

    crear();
}
```

Esta conexión se reutiliza en todas las clases a través de `Sesion.java`, evitando duplicaciones y mejorando la eficiencia.

- **Gestión de imágenes:**
La clase `InsertarImagenesRecetas.java` automatiza la carga de imágenes desde la carpeta `/imagen_receta/`, convirtiéndolas a `byte[]` para almacenarlas como BLOB en la base de datos mediante consultas UPDATE.
- **Persistencia de sesión:**
La clase `Sesion` almacena tanto la conexión como el usuario activo, lo que permite navegar entre pantallas sin necesidad de reconexión o reidentificación.

```

import java.sql.Connection;

//clase que mantiene el estado del usuario autenticado y la conexión activa
public class Sesion {
    private Usuario usuario;
    private Connection connection;

    //constructor que recibe el usuario actual y la conexión a la base de datos
    public Sesion(Usuario usuario, Connection connection) {
        this.usuario = usuario;
        this.connection = connection;
    }

    //devuelve el usuario autenticado
    public Usuario getUsuario() {
        return usuario;
    }

    //devuelve la conexión activa a la base de datos
    public Connection getConnection() {
        return connection;
    }

    //actualiza el usuario actual de la sesión
    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
    }

    //actualiza la conexión asociada a la sesión
    public void setConnection(Connection connection) {
        this.connection = connection;
    }
}

```

- **Gestión modular del sistema:**

Cada clase cumple una función concreta. Por ejemplo, ClaseReceta gestiona el CRUD de recetas; ClaseUsuario gestiona el login y los datos del usuario; ClaseAlimento accede a la tabla de ingredientes. Esto permite una arquitectura mantenible y reutilizable.

Núm de palabras: 1197