Estándar de codificación

Guía de Codificación para la Aplicación Cliente EduShare (Android – Java)

Elaborado por:

Juan Eduardo Cumplido Negrete

Erick Abdiel Atzin Olarte

Christian Alberto Vásquez Cruz

Equipo: CAPA 8

Fecha de emisión: 10 de mayo de 2025

Contenido

1. Introducción

Durante el desarrollo del sistema EduShare, correspondiente a las experiencias educativas de Desarrollo de Sistemas en Red y Desarrollo de Aplicaciones, se ha definido el uso del lenguaje de programación Java dentro del entorno de desarrollo Android Studio 2024.

Este documento establece las normas y buenas prácticas de codificación para la aplicación móvil Android cliente del sistema EduShare. Estas normas están orientadas al uso del patrón de arquitectura MVVM (Model-View-ViewModel), buscando una separación clara de responsabilidades, facilitando la escalabilidad, mantenibilidad y pruebas del sistema.

El código se escribirá en idioma español, conservando coherencia semántica en nombres de variables, métodos y clases, sin comprometer la legibilidad o comprensión por parte de otros desarrolladores.

El estándar de codificación está estructurado en las siguientes secciones:

- Propósito del documento.
- Reglas de nombrado para clases, variables, constantes, métodos, paquetes y recursos.
- Estilo de código, incluyendo sangría, uso de llaves, espacios, y longitud de línea.
- Convenciones del equipo, incluyendo reglas internas que puedan establecerse en el transcurso del desarrollo.
- Directrices sobre comentarios, tanto en línea como a nivel de documentación para clases, métodos y lógica compleja.
- Buenas prácticas en control de flujo, manejo de excepciones, seguridad y uso de componentes específicos de Android (LiveData, ViewModel, Activity, Fragment, etc.).

Este documento es vivo y sujeto a cambios conforme avance el desarrollo. Cualquier mejora, modificación o adición será documentada explícitamente en este archivo con el fin de mantener una guía clara, uniforme y que fortalezca el trabajo colaborativo y el mantenimiento a largo plazo del sistema.

2. Propósito

El propósito de este estándar de codificación es establecer directrices claras y coherentes que garanticen la calidad, mantenibilidad y legibilidad del código. Al seguir estas pautas, buscamos:

1. **Uniformidad:** Asegurar que todo el código fuente se adhiera a un formato y estilo consistente, lo que facilita la comprensión y colaboración entre los miembros del equipo.
2. **Calidad del Código:** Promover la escritura de código limpio y eficiente, que cumpla con las mejores prácticas de desarrollo y reduzca la posibilidad de errores y defectos.
3. **Mantenibilidad:** Facilitar la modificación y extensión del código en el futuro mediante una estructura y nomenclatura bien definidas, lo que permite que el código sea fácilmente comprensible y modificable por otros desarrolladores.
4. **Documentación:** Garantizar que el código esté adecuadamente documentado, proporcionando información clara sobre su propósito, uso y funcionamiento, para mejorar la comprensión y la facilidad de mantenimiento.
5. **Estandarización:** Fomentar la adopción de una metodología común que pueda ser aplicada en todos los proyectos, promoviendo la coherencia y la integración entre diferentes módulos y equipos de trabajo.
6. **Seguridad:** Incorporar prácticas que aseguren la protección de los datos y la prevención de vulnerabilidades, contribuyendo a la creación de aplicaciones seguras y confiables.

Al adherirse a este estándar, nos aseguramos de que nuestro código sea de alta calidad, fácil de mantener y que cumpla con los requisitos funcionales y no funcionales del proyecto.

3. Reglas de nombrado o convención de nombres

1. Utilizar **nombres significativos y descriptivos** que comuniquen claramente la intención del elemento (variable, clase, método).
2. Priorizar **claridad sobre brevedad**.
3. Evitar el uso de abreviaciones, excepto aquellas **ampliamente reconocidas** (por ejemplo, `URL`, `ID`, `API`).
4. El código debe escribirse en **idioma español**, pero conservando la claridad y coherencia con nombres técnicos en inglés si es necesario (por ejemplo: `usuarioList`, `cargarDatosViewModel`).

3.1 Variables

Reglas:

1. Declarar variables lo más cerca posible de su uso.
2. Evitar declarar múltiples variables en una sola línea.

Variables locales y parámetros de métodos: Utilizar *camelCase*.

Ejemplo correcto

```
int montoTotal;
string primerApellido;
bool registrado;
```

Ejemplo incorrecto

```
int Montototal;
string primerpellido;
bool Registrado;
```

Campos privados: Usar camelCase y el modificador private.

Ejemplo correcto

```
private int contador;
private UsuarioViewModel usuarioViewModel;
```

Ejemplo incorrecto

```
private int _contador;
```

3.2 Constantes

Regla de Nomenclatura: Utilizar static final para declarar constantes.

Convención: Se recomienda el uso de mayúsculas con guiones bajos (SCREAMING_SNAKE_CASE), siguiendo las convenciones de Java y Android.

Ejemplo correcto

```
public static final int TIEMPO_ESPERA_SEGUNDOS = 30;
public static final String CLAVE_USUARIO = "CLAVE_USUARIO";
```

Ejemplo incorrecto

```
public static final int TiempoEsperaSegundos = 30;
```

Justificación:
Nombrar las constantes con *SCREAMING_SNAKE_CASE* ayuda a diferenciarlas visualmente de las variables, métodos y clases. Este enfoque estandarizado permite que las constantes sean fácilmente identificables en el código, lo que es crucial para la comprensión y mantenimiento del mismo.

3.3 Métodos

Regla de Nomenclatura: Se debe utilizar camelCase para nombrar métodos.

Convención: Los nombres de los métodos deben ser verbos o verbos seguidos de sustantivos, comenzando la segunda palabra con mayúscula.

Ejemplo correcto

```
void cargarDatosUsuario();
boolean esCorreoValido(String correo);
```

Ejemplo incorrecto

```
void Datos();
void setnombre();
```

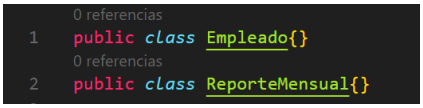
Justificación:
El uso de *camel Case* para los métodos ayuda a diferenciarlos de variables y constantes, reforzando la claridad del código. La convención de nombrar métodos como verbos o frases verbales también es importante, ya que los métodos generalmente representan acciones o procesos. Este enfoque no solo mejora la legibilidad del código, sino que también facilita la comprensión de la funcionalidad del método sin necesidad de profundizar en su implementación. Al mantener una nomenclatura consistente y descriptiva, se reduce la curva de aprendizaje para los nuevos desarrolladores en el proyecto y se mejora la colaboración dentro del equipo.

3.4 Clases

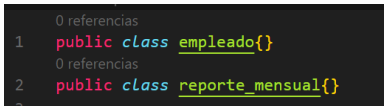
Regla de Nomenclatura: Se debe utilizar *Pascal Case* para nombrar clases.

Convención: Los nombres de las clases deben ser sustantivos o frases de sustantivos, y cada palabra debe comenzar con mayúscula.

Ejemplo correcto



Ejemplo incorrecto

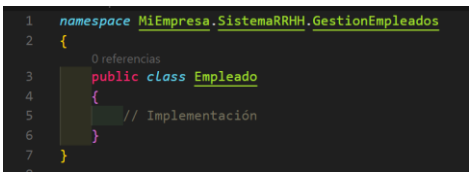


Justificación:
Las clases representan entidades o conceptos clave dentro del código, y el uso de sustantivos o frases de sustantivos como nombres refuerza esta idea. Esta convención también facilita la identificación de las clases al revisar o navegar por el código, lo que es esencial en proyectos grandes o de larga duración. Mantener esta consistencia en la nomenclatura de clases contribuye a una estructura de código bien organizada y a una mayor facilidad de mantenimiento.

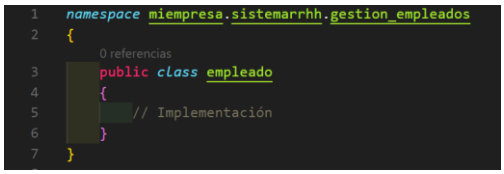
3.5 Espacio de Nombres (Namespace)

1. Usar **todo en minúsculas**, separado por puntos.
2. Reflejar la **estructura lógica del proyecto**.
3. Seguir la convención `com.nombreproyecto.modulo`.

Ejemplo correcto



Ejemplo incorrecto



Justificación:
Los nombres de los espacios de nombres en minúsculas separados por puntos proporcionan una estructura clara y jerárquica, que facilita la organización del código en módulos lógicos. Al reflejar la jerarquía del proyecto, los namespaces permiten una navegación más eficiente y comprensible del código, especialmente en aplicaciones grandes o en desarrollo continuo. Esta convención también ayuda a evitar conflictos de nombres, asegurando que cada namespace sea único dentro del proyecto. Al seguir estas directrices, los desarrolladores pueden mantener una estructura de código limpia y bien organizada, lo que es vital para la colaboración en equipo y la escalabilidad del proyecto.

3.1 Interfaces

Regla de Nomenclatura: Se debe utilizar *Pascal Case* para nombrar interfaces, no se recomienda prefijar los nombres con la letra `I`.

Convención: Los nombres de las interfaces deben ser sustantivos y estar en *Pascal Case*.

Ejemplo correcto

```
interface OnUsuarioClickListener;
interface RepositorioUsuarios;
```

Ejemplo incorrecto

```
interface IUsuario;
```

Justificación:
Utilizar *Pascal Case* ayuda a distinguirlas claramente de las clases y otros tipos en el código. Este enfoque proporciona una forma rápida y consistente de identificar las interfaces, lo cual es crucial en la implementación de patrones de diseño como la inyección de dependencias y la programación orientada a interfaces. Al seguir esta convención, se facilita la comprensión del código y se mejora la colaboración en equipos grandes al asegurar una identificación clara y uniforme de las interfaces a lo largo del proyecto.

3.2 Enums

Regla de Nomenclatura: Se debe utilizar *Pascal Case* para nombrar enums y sus valores.

Convención: Los nombres de los enums deben ser sustantivos en Pascal Case, y los valores dentro del enum también deben seguir Pascal Case.

Ejemplo correcto

```
0 referencias
1  public enum EstadoOrden;
0 referencias
2  public enum TipoEmpleado;
3
```

Ejemplo incorrecto

```
0 referencias
1  public enum estado_Orden;
0 referencias
2  public enum Tipo_empleado;
```

Justificación:
Utilizar *Pascal Case* para los enums y sus valores garantiza la claridad y la coherencia en la representación de conjuntos de valores predefinidos. Esto facilita la lectura del código y la comprensión del propósito de cada valor en el contexto de la enumeración. Al seguir una convención uniforme, se reduce la confusión y se mejora la mantenibilidad del código. Los nombres descriptivos y bien estructurados para enums permiten una mejor comunicación de las intenciones del código y simplifican la implementación de lógica condicional basada en los valores de la enumeración.

4 Estilo de código

Dentro de este apartado, se establecen las normas para poder escribir un código de manera clara y consistente. Tomando en cuenta reglas sobre los niveles de indentación, espacios en blanco, uso de llaves y modificadores de acceso.

4.1 Indentación

La indentación del código será de cuatro espacios por nivel, permitiendo un máximo de hasta cinco niveles de indentación en los métodos, a excepción de casos en específico los cuales deberán tener una justificación de por qué se realizaron de esa manera.

Justificación:

La indentación de código es fundamental para ayudarnos a tener mejor legibilidad y comprensión de código, facilitando la visión de la estructura lógica con la que cuenta el programa, así como de las estructuras de control.

Ejemplo sobre el uso aceptable de Indentación:

```
public void ProcesarDatos()
{
    if (esValido)
    {
        foreach (var item in lista)
        {
            Console.WriteLine(item);
        }
    }
}
```

Ejemplo sobre el uso incorrecto de Indentación:

```
public void Procesar()
{
    if (condicion1)
    {
        if (condicion2)
        {
            while (condicion3)
            {
                if (condicion4)
                {
                    for (int i = 0; i < 10; i++)
                    {
                        if (condicion5)
                        {
                            Console.WriteLine("Demasiado anidado");
                        }
                    }
                }
            }
        }
    }
}
```

4.2 Líneas en blanco

Para cada uno de estos casos, se deberá usar una línea de código en blanco:

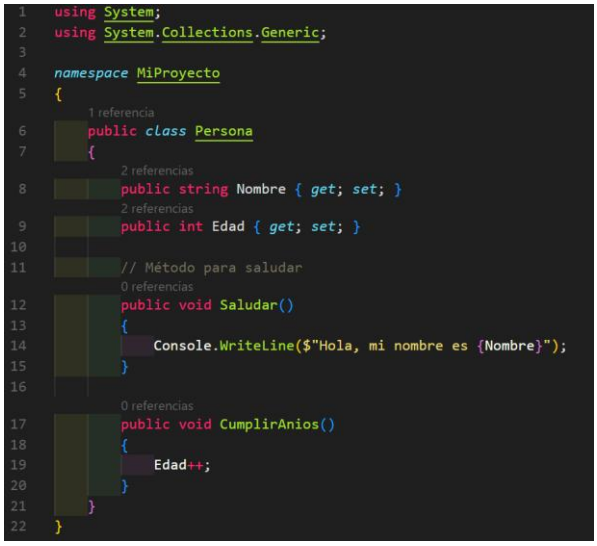
- Entre métodos
- Atributos de una clase y su primer método

- Después de un comentario
- Después de la declaración de los paquetes o librerías a usar (Using)

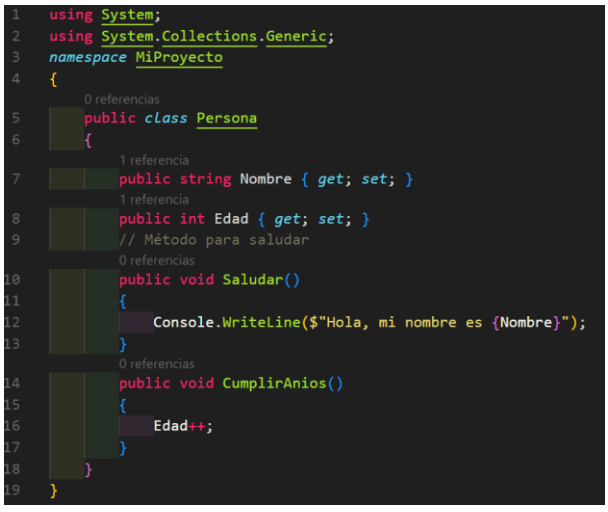
Justificación:

Las reglas dadas sobre el uso de las líneas en blanco, nos facilitarán la lectura de código, identificando de manera más rápida donde se usan las estructuras de control, la separación entre métodos, así como los atributos de una clase y de los comentarios de código que se lleguen a realizar.

Ejemplo correcto



Ejemplo incorrecto



4.3 Uso de llaves

Dentro del uso de llaves, cada vez que se abra una llave, deberá hacerse seguido de la línea de código en donde se pondría, pero colocando la llave de cierre una línea por debajo de la última línea de código del método a la hora de cerrarse una llave. En otras palabras, se

asignará una línea de código especifica y única para cada llave de cierre de declaración de una clase, namespace o estructura de control.

En caso de que exista una estructura de control que solo contenga una línea de código, deberá usarse su perspectiva llave de inicio y cierre, siguiendo la regla especificada.

Justificación:

El buen manejo y uso de apertura y cierre de llaves, nos permite tener una mejor comprensión del código, así como de estética, donde podremos apreciar de forma clara que llave de cierre pertenece a una llave de apertura y viceversa.

Ejemplo correcto de uso de las llaves:

```
if (condición) {
    // cuerpo
}
```

Ejemplo incorrecto del uso de las llaves:

```
12      public void Saludar()
13      {
14          Console.WriteLine($"Hola, mi nombre es {Nombre}");
15      }
16
```

5 Comentarios

5.1 Comentarios de Código

Regla de Comentarios: Los comentarios de código deben ser claros, concisos y relevantes. Deben explicar la lógica compleja o cualquier decisión de diseño inusual, pero no repetir lo que el código ya comunica claramente.
Convención: Utilizar // para comentarios de una sola línea y /* */ para comentarios de múltiples líneas.

Ejemplo correcto

```
1   // Verifica si el usuario tiene permisos antes de continuar
2   if (usuario.TienePermiso("Admin"))
3   {
4       EjecutarOperacionCritica();
5   }
6
7   /*
8       Este algoritmo recorre una lista de productos y agrupa
9       los que tienen el mismo código, sumando sus cantidades.
10  */
11  AgruparProductosSimilares(listaProductos);
```

Ejemplo incorrecto

```
16  // Verifica si el usuario tiene permisos
17  if (usuario.TienePermiso("Admin")) // Llama a EjecutarOperacionCritica si tiene permisos
18  {
19      EjecutarOperacionCritica(); // Ejecutar
20  }
21
22  // Este es un comentario muy largo que intenta explicar algo simple
23  // y que debería estar en un lugar más conciso o dentro de documentación
```

Justificación:

En Android y Java, los comentarios deben evitar la redundancia y enfocarse en lo que no es obvio. Esto mejora la mantenibilidad del código y la colaboración entre desarrolladores. Sin embargo, es importante que los comentarios no repitan lo que ya es evidente en el propio código, ya que esto puede llevar a redundancia y desorden. Un comentario bien colocado puede ahorrar tiempo a otros desarrolladores y a uno mismo cuando se revisa el código después de un tiempo. Por lo tanto, los comentarios deben ser precisos, evitando ambigüedades y redundancias innecesarias, contribuyendo así a la mantenibilidad del código.

6 Estructuras de control

6.1 If / else if

Regla de Formato:

- Las instrucciones if deben estar seguidas de llaves ({}) incluso si el bloque de código tiene una sola línea.
- El bloque de código dentro del if, else if, y else debe estar indentado a 4 espacios.
- La condición del if debe ir entre paréntesis (), y el bloque de código debe estar entre llaves {}.

Convención:

- Utilizar llaves {} para los bloques de código de if, else if, y else, incluso si son de una sola línea.
- Alinear el código dentro de los bloques para mejorar la legibilidad.

Ejemplo correcto

```
if (usuario != null) {
    mostrarPerfil(usuario);
} else {
    redirigirALogin();
}
```

Ejemplo incorrecto

```
if (usuario != null)
    mostrarPerfil(usuario);
else
    redirigirALogin();
```

Justificación:
El uso de llaves incluso para bloques de una sola línea previene errores sutiles cuando se agregan más líneas de código en el futuro. La estructura clara con llaves asegura que el código sea más fácil de entender y mantener. El ejemplo que no cumple puede llevar a errores si se agregan más líneas en el futuro, ya que no se garantiza que todas las líneas se incluyan en el bloque condicional.

6.2 Switch

Regla de Formato:

- La palabra clave switch debe estar seguida de un paréntesis que contiene la expresión de control.
- Cada caso dentro del switch debe estar precedido por la palabra clave case y seguido de dos puntos :.
- Los bloques de código asociados a cada case deben estar encerrados entre llaves {} y deben estar correctamente indentados.
- Utilizar break; para terminar cada bloque de código de un case para evitar fallos en cascada.
- El bloque default debe estar al final del switch y debe estar precedido por la palabra clave default.

Convención:

- Incluir una declaración break en cada case, a menos que se utilicen otras técnicas para evitar el flujo continuo, como return o throw.
- Alinear los bloques de código para mejorar la claridad y organización.

Ejemplo correcto

```
switch (opcion) {
    case 1: {
        iniciarActividad();
        break;
    }
    case 2: {
        mostrarDialogo();
        break;
    }
    default: {
        mostrarError();
        break;
    }
}
```

Ejemplo incorrecto

```
22  switch (opcion)
23  {
24      case 1:
25          EjecutarOpcionUno();
26      case 2:
27          EjecutarOpcionDos();
28          break;
29      default:
30          MostrarMensajeError();
31  }
```

Justificación:
El formato adecuado para la instrucción switch asegura que cada caso sea claramente separado y el flujo de control sea evidente. El uso de break evita errores sutiles de flujo continuo, y el default proporciona una manera de manejar valores no esperados. El ejemplo que no cumple puede llevar a comportamientos inesperados debido a la falta de break, causando que varios casos se ejecuten en secuencia.

6.3 For

Regla de Formato:

- La instrucción for debe estar seguida de paréntesis () que contengan la inicialización, condición y actualización.
- El bloque de código dentro del for debe estar entre llaves {} y debe estar indentado a 4 espacios.

Convención:

- Utilizar llaves {} para los bloques de código del for para mantener una estructura clara.

Ejemplo correcto

```
for (int i = 0; i < lista.size(); i++) {
    Log.d("Debug", "Elemento: " + lista.get(i));
}
```

Ejemplo incorrecto

```
for (int i = 0; i < lista.size(); i++)
    Log.d("Debug", "Elemento: " + lista.get(i));
```

Justificación:
El uso de llaves en los bucles for mejora la claridad y evita errores asociados con la falta de llaves, como la inclusión no intencionada de más líneas de código en el bucle. La estructura y la indentación uniforme facilitan la lectura y comprensión del bucle, lo que ayuda en la depuración y mantenimiento del código. El ejemplo que no cumple puede ser problemático si se agregan más líneas de código dentro del bucle en el futuro.

6.4 Bucle while y do-while

Regla de Formato:

- La instrucción while debe estar seguida de un paréntesis () que contenga la condición.
- La instrucción do debe estar seguida de un bloque de código entre llaves {}.
- El bloque de código asociado a while y do-while debe estar indentado a 4 espacios.
- La instrucción while debe aparecer después del bloque do en un bucle do-while.

Convención:

- Utilizar llaves {} para los bloques de código dentro de while y do-while.
- Alinear el código dentro de los bloques para mantener la legibilidad.

Ejemplo correcto

```
while (cursor.moveToNext()) {
    procesarDato(cursor);
}

do {
    leerEntrada();
} while (!entradaValida);
```

Ejemplo incorrecto

```
while (cursor.moveToNext())
    procesarDato(cursor);

do
    leerEntrada();
while (!entradaValida);
```

Justificación:
Los bucles while y do-while deben estar estructurados con llaves y una indentación adecuada para garantizar que el flujo de control sea claro. Utilizar llaves evita errores de ejecución y facilita la adición o modificación del código dentro del bucle. La consistencia en la estructura de los bucles mejora la legibilidad y facilita el mantenimiento del código. El ejemplo que no cumple puede llevar a errores debido a la falta de llaves, resultando en un comportamiento no deseado.

7 Manejo de excepciones

Las excepciones deben ser manejadas localmente en la capa donde se originan. Cada componente (Repositorio, ViewModel, etc.) es responsable de gestionar sus errores, en lugar de propagar la excepción hasta la capa de la interfaz de usuario. Esto permite mantener una arquitectura desacoplada y robusta.

Para el control de excepciones, seguiremos utilizando la estructura try-catch, registrando la causa de la excepción en la bitácora correspondiente utlizando log.e, log.w o log.d y asignándole una clasificación según el tipo de excepción. Además, en lugar de propagar las excepciones a capas superiores, se enviarán valores de retorno correspondientes que permitan manejar los errores de manera contextual.

En cuanto a las clases validadoras, si los valores no cumplen con los requisitos de las expresiones regulares, se lanzará una excepción local que será manejada inmediatamente, evitando la propagación innecesaria de errores.

Ejemplo sobre el buen uso del manejo de excepciones:

```
public Usuario obtenerUsuarioPorId(int id) {
    try {
        // Supongamos que accede a una base de datos local o remota
        return usuarioDao.obtenerPorId(id);
    } catch (SQLException e) {
        Log.e("UsuarioRepo", "Error al obtener usuario por ID", e);
        return null; // o una respuesta controlada
    }
}
```

Ejemplo sobre el uso incorrecto del manejo de excepciones:

```
public Usuario obtenerUsuarioPorId(int id) throws SQLException {
    return usuarioDao.obtenerPorId(id); // Propaga la excepción
}
```

Justificación:

El manejo adecuado de excepciones permite evitar caídas innecesarias de la aplicación, y ofrece una experiencia de usuario más robusta. Además, facilita la depuración al registrar adecuadamente los errores y permite responder de forma contextual a cada caso, sin comprometer la estabilidad del sistema.

8 Interfaz de usuario

Este apartado establece un estándar para el nombrado de los elementos de interfaz gráfica en Android. El uso de prefijos ayuda a identificar rápidamente el tipo de componente, facilita la lectura del código, y mejora el mantenimiento de las vistas, especialmente en proyectos grandes o colaborativos.

8.1 Prefijos para componentes de interfaz gráfica

Cada componente de UI debe tener un prefijo descriptivo en su id, seguido de un nombre significativo que indique su función o contenido. El prefijo se escribe en minúsculas, separado por guion bajo (_) del nombre del componente.

Interfaz de usuario	Prefijo	Correcto	Incorrecto
Button	**btn_**	btn_login, btn_submit	botonLogin, buttonSubmit
TextView	**txt_**	txt_username, txt_title	textViewUser, tvTitle
EditText	**edt_**	edt_email, edt_password	inputEmail, editTextPass
ImageView	**img_**	img_logo, img_profile	imageView1, logoImage
CheckBox	**chk_**	chk_terms, chk_newsletter	checkTerms, cbNewsletter
RadioButton	**rbtn_**	rbtn_male, rbtn_female	radioButtonMale, rb_female
RadioGroup	**rg_**	rg_gender, rg_options	groupGender, radioGroup
Switch	**swt_**	swt_notifications, swt_mode	switch1, switchMode
ToggleButton	**tgl_**	tgl_theme, tgl_password	togglePassword, toggle1
Spinner	**spn_**	spn_country, spn_category	spinnerCategory, spin1
ListView	**lv_**	lv_contacts, lv_messages	listViewContact, list1
RecyclerView	**rv_**	rv_posts, rv_results	recyclerView, resultList
ProgressBar	**pb_**	pb_loading, pb_upload	progressBar, barLoading
SeekBar	**skb_**	skb_volume, skb_brightness	seekbarVol, seekBrightness
RatingBar	**rtb_**	rtb_score, rtb_feedback	ratingBar, ratingScore
WebView	**wv_**	wv_terms, wv_privacy	webView1, termsWeb
ViewPager	**vp_**	vp_images, vp_pages	pagerImages, viewPager
TabLayout	**tab_**	tab_menu, tab_navigation	tabs, tabBar
ConstraintLayout	**cl_**	cl_main, cl_container	constraintLayout, mainLayout
LinearLayout	**ll_**	ll_header, ll_footer	linearLayout, headerLayout

RelativeLayout	**rl_**	rl_login, rl_profile	relativeLayout, loginRel
FrameLayout	**fl_**	fl_content, fl_placeholder	frameLayout, contentFrame
ScrollView	**sv_**	sv_form, sv_list	scrollView, formScroll
CardView	**cv_**	cv_product, cv_article	cardView, productCard
FloatingActionButton	**fab_**	fab_add, fab_chat	addFab, floatingButton
TextInputLayout	**til_**	til_email, til_password	inputLayoutEmail, textInputPassword
TextInputEditText	**tie_**	tie_email, tie_password	editTextInput, inputEditText

9 PRÁCTICAS DE CONSTRUCCIÓN SEGURA

Dentro de este apartado del estándar de codificación, se explicará las diferentes prácticas de programación que permitirán la integridad de los datos que se manipulen, así como la validación de entrada de datos, con el fin de mantener un código seguro y que no sea susceptible a inyección de código o cualquier otra vulnerabilidad.

9.1 Validación de datos

Para la validación de entradas de datos por el usuario dentro de la capa de interfaces de usuario, se contarán con clases validadoras que contendrán expresiones regulares y métodos con el algoritmo de detección en caso de que el dato ingresado no cumpla con las reglas de valores aceptados, asegurando que no se exceda la inserción de aquellos tipos de datos que no sean permitidos al igual que poder controlar el flujo de caracteres sin rebasar los límites establecidos dentro de la base de datos. Destacando que esta práctica de seguridad ayude a mitigar la inyección de código.

Justificación:

Es importante poner en práctica la validación de datos que el usuario intente ingresar, creando una brecha de seguridad que ayude a mantener íntegros los datos con los que se cuentan dentro de la base de datos, así como el código fuente; previniendo ataques que atenten contra la integridad del sistema.

9.3 Hasheo de secretos

Para el manejo de secretos dentro del proyecto, en todo momento serán hasheados a la hora de guardarlos dentro de la base de datos, así como de recuperarlos, con el fin de mantener la seguridad en aquella información que pueda ser demasiado sensible y pueda provocar un gran problema de seguridad su mala protección.

Justificación:

El hashing de secretos como contraseñas o tokens, es una práctica esencial en cualquier proyecto que manipule información sensible. El uso de esta práctica mantiene la seguridad de la información sensible que es objetivo de los atacantes al sistema, además de que se cumplen estándares de seguridad, reduciendo el impacto de las filtraciones de datos.

9.4 Política de contraseñas

Para la protección de cuentas de usuario dentro del sistema, con el fin de mantener contraseñas suficientemente robustas y seguras, su longitud mínima deberá estar entre los 12 y 23 caracteres incluyendo como mínimo una letra mayúscula, una letra minúscula, un número y un carácter especial. Evitando el uso de contraseñas sensibles y que puedan ser fáciles de adivinar.

Justificación:

La implementación de una política de contraseñas es fundamental para garantizar la seguridad y protección de datos sensibles, así como para mantener la integridad y confiabilidad del sistema, manteniendo protección contra accesos no autorizados y mitigando riesgos de seguridad.