Estándar de codificación

Guía de Codificación para el servicio APIRest

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 3
2. Propósito 4
3. Reglas de nombrado o convención de nombres 5
   3.1 Variables 5
   3.2 Constantes 5
   3.3 Métodos 5
   3.4 Clases 6
   3.5 Espacio de Nombres (Namespace) ..... **¡Error! Marcador no definido.**
   3.6 Interfaces ..... **¡Error! Marcador no definido.**
   3.7 Enums 7
4 Estilo de código 8
   4.1 Indentación 8
   4.2 Líneas en blanco 8
   4.3 Uso de llaves 9
5 Comentarios 10
   5.1 Comentarios de Código 10
6 Estructuras de control 11
   6.1 If / else if 11
   6.2 Switch 12
   6.3 For 12
   6.4 Bucle while y do-while 13
7 Manejo de excepciones 14
8 Interfaz de usuario ..... **¡Error! Marcador no definido.**
   8.1 Prefijos para componentes de interfaz gráfica ..... **¡Error! Marcador no definido.**
9 PRÁCTICAS DE CONSTRUCCIÓN SEGURA 16
   9.1 Validación de datos 16
   9.3 Hasheo de secretos 16
   9.4 Política de contraseñas 17

1. Introducción

Durante el desarrollo del proyecto correspondiente a la Experiencia Educativa *Desarrollo de Sistemas en Red y Desarrollo de Aplicaciones*, se eligió como lenguaje principal de programación para el backend JavaScript, utilizando el entorno de ejecución Node.js basado en el motor V8 de Google.

Este documento establece las normas y buenas prácticas de codificación exclusivamente para la aplicación "Edushare API-REST" del sistema EduShare, la cual ha sido desarrollada utilizando el framework Express en Node.js, se programará en español.

El estándar de codificación está compuesto por diversas secciones, entre ellas:

- Propósito del documento.
- Reglas de nombrado para variables, constantes, métodos, clases, y propiedades.
- Patrones para manejar operaciones asíncronas sobre el hilo de ejecución de Node.js
- Estilo de código, incluyendo la indentación y el uso de llaves.
- Convenciones internas del equipo que puedan surgir y acordarse durante el desarrollo del proyecto.
- Directrices sobre comentarios, tanto para anotaciones en el código como para documentación de clases que definan servicios del sistema.
- Buenas prácticas en estructuras de control, manejo de excepciones y programación segura.

Es importante destacar que, a lo largo del ciclo de desarrollo, este estándar podrá ser modificado conforme se presenten avances o se identifiquen nuevas necesidades. Cualquier propuesta de mejora o adición será debidamente documentada dentro de este mismo archivo, con el objetivo de mantener una guía clara y coherente que facilite la colaboración del equipo y el mantenimiento futuro del proyecto. De este modo, se busca promover un entorno de trabajo bien estructurado, limpio y documentado.

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 para variables, métodos y clases.
2. Preferir la claridad a la brevedad.
3. Evitar utilizar abreviaturas o acrónimos en los nombres, excepto las abreviaturas ampliamente conocidas y aceptadas.

3.1 Variables

Reglas:

1. Declarar variables lo más cerca posible de su uso.
2. Evitar declarar múltiples variables en una sola línea.
3. Inicializar todas las variables como const o let

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

Ejemplo correcto

```
let resultadoInsercion
let configuracionConexion
let conexion;
```

Ejemplo incorrecto

```
let ResultadoInsercion
let ConfiguracionConexion
let Conexion;
```

3.2 Constantes

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

Convención: Las constantes deben comenzar con una letra mayúscula y cada palabra subsiguiente también debe estar en mayúscula. Se recomienda utilizar el modificador const o readonly según corresponda.

Ejemplo correcto

```
const ConfiguracionConexion = RetornarTipoDeConexion();
const Pi = 3.1416
```

Ejemplo incorrecto

```
const Configuracion_Conexion = RetornarTipoDeConexion();
const _Pi = 3.1416
```

Justificación:
Nombrar las constantes con *Pascal 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 Pascal Case para nombrar métodos.

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

Ejemplo correcto

```
static async InsertarNuevaCuenta({datos}){

}

static async EliminarCuenta(idCuenta){

}
```

Ejemplo incorrecto

```
3  static async Insertar_Nueva_Cuenta({datos}){
4
5  }
6
7  static async eliminarcuenta(idCuenta){
8
9  }
```

Justificación:
El uso de *Pascal 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

```
2  export class ModeloAcceso{
3
4    }
5
6  export class ModeloPublicacion{
7
8    }
```

Ejemplo incorrecto

```
2  export class modeloAcceso{
3
4  }
5
6  export class modelo_Publicacion{
7
8  }
```

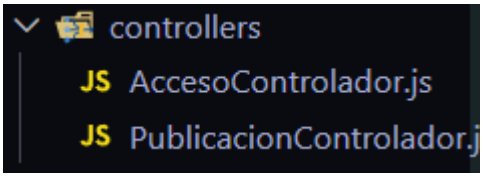
Justificación:
Las clases representan entidades o conceptos clave dentro del código, y el uso de sustantivos o frases de sustantivos como nombres refuerza esta idea. Esta convención también facilita la identificación de las clases al revisar o navegar por el código, lo que es esencial en proyectos grandes 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 Nombre de archivos

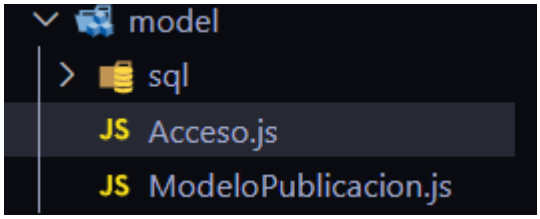
Regla de Nomenclatura: Se debe utilizar *Pascal Case* para nombrar archivos y la extensión .js

Convención: Los espacios de nombres deben reflejar la jerarquía lógica del proyecto, comenzando con el nombre de la compañía u organización, seguido por el nombre del proyecto y las áreas específicas.

Ejemplo correcto



Ejemplo incorrecto



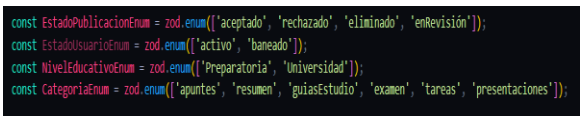
Justificación:
Los nombres de los archivos **Pascal Case** proporcionan una estructura clara y jerárquica, que facilita la organización del código en módulos lógicos. Al reflejar la jerarquía del proyecto, permite una navegación más eficiente y comprensible del código, especialmente en aplicaciones grandes o en desarrollo continuo. Es importante también respetar la nomenclatura de prefijos, siendo controladores, parte del modelo o similares.

3.6 Enums

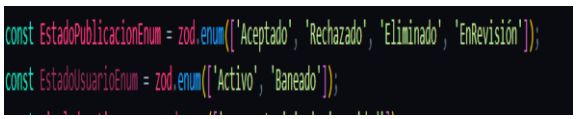
Regla de Nomenclatura: Se debe utilizar *Pascal Case* para nombrar enums y sus valores.

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



Ejemplo incorrecto



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 blando, 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:

```
static async ProcesarDatos(){
        if (esValido){
                foreach (var item in lista){
                        console.log(item)
                }
        }
}
```

Ejemplo sobre el uso incorrecto de Indentación:

```
static async ProcesarDatos(){
        if (condicion1){
                if (condicion2){
                        while (condicion3){
                                if (condicion4){
                                        for (int i=0; i<0; i++){
                                                if (condicion5){
                                                        console.log("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

```
import express, { json, urlencoded } from 'express';
import { CrearRutaAcceso } from './api_rest/routes/Acceso.js';
import { CrearRutaPublicacion } from './api_rest/routes/Publicaciones.js';
import { DocumentoSwagger } from './api_rest/utilidades/swagger.js';
import swaggerUI from 'swagger-ui-express';
import dotenv from 'dotenv';
import cors from 'cors';

export const CrearServidor = ({ModeloAcceso, ModeloPublicaciones}) => {
    const app = express();
    dotenv.config();
    app.use(json({limit:'100mb'}));
    app.use(urlencoded({limit:'100mb', extended:true}))
    app.use(cors());
    app.disable('x-powered-by');

    app.use('/edushare/acceso', CrearRutaAcceso({ModeloAcceso}));

    app.use('/edushare/publicaciones', CrearRutaPublicacion({ModeloPublicaciones}));

    app.use('/edushare/doc',swaggerUI.serve, swaggerUI.setup(DocumentoSwagger));

    const PUERTO = process.env.PUERTO;

    app.listen(PUERTO,()=>{
        console.log(`Servidor activo en la siguiente ruta http://localhost:${PUERTO}`);
    })
}
```

Ejemplo incorrecto

```
import express, { json, urlencoded } from 'express';

import { CrearRutaAcceso } from './api_rest/routes/Acceso.js';

import { CrearRutaPublicacion } from './api_rest/routes/Publicaciones.js';

import { DocumentoSwagger } from './api_rest/utilidades/swagger.js';

import swaggerUI from 'swagger-ui-express';

import dotenv from 'dotenv';

import cors from 'cors';

export const CrearServidor = ({ModeloAcceso, ModeloPublicaciones}) => {
    const app = express();
    dotenv.config();
    app.use(json({limit:'100mb'}));
    app.use(urlencoded({limit:'100mb', extended:true}))
    app.use(cors());
    app.disable('x-powered-by');
    app.use('/edushare/acceso', CrearRutaAcceso({ModeloAcceso}));
    app.use('/edushare/publicaciones', CrearRutaPublicacion({ModeloPublicaciones}));
    app.use('/edushare/doc',swaggerUI.serve, swaggerUI.setup(DocumentoSwagger));
    const PUERTO = process.env.PUERTO;
    app.listen(PUERTO,()=>{
        console.log(`Servidor activo en la siguiente ruta http://localhost:${PUERTO}`);
    })
}
```

4.3 Uso de llaves

Dentro del uso de llaves, cada vez que se abra una llave, deberá hacerse con "K&R Style" (Kernighan & Ritchie Style). En otras palabras, se colocará sobre la declaración de la clase, namespace o estructura de control la llave de inicio y en la línea siguiente, al final, la llave de cierre.

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:

```
1   static async saludar(){
2           console.log($"Hola, mi nombre es{nombre}")
3   }
4
```

Ejemplo incorrecto del uso de las llaves:

```
static async saludar()
{
        console.log($"Hola, mi nombre es{nombre}")
}
```

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

```
//Verificar si el usuario tiene permisos
if (usuario.TienePermisos("admin")){ //Llama a EjecutarOperacionCritica si tiene permisos
        EjecutarOperacionCritica(); //Ejeecutar
}
```

Justificación:

Los comentarios de código son fundamentales para aclarar partes complejas o justificaciones de diseño que podrían no ser obvias al leer el código. 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 una tabulación con la configuración por defecto de VSC.
- 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 (usuarioEsValido){
        ProcesarUsuario();
} else if (usuarioEstaInactivo){
        MostrarMensajeDeError();
} else {
        RegistrarIntentoFallido();
}
```

Ejemplo incorrecto

```
if (usuarioEsValido){
ProcesarUsuario();
} else if (usuarioEstaInactivo){
MostrarMensajeDeError();
} else {
RegistrarIntentoFallido();
}
```

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 (statusCode) {
  case 200:
    console.log("Éxito");
    break;
  case 404:
    console.log("No encontrado");
    break;
  default:
    console.log("Error desconocido");
}
```

Ejemplo incorrecto

```
switch (statusCode) {
  case 200:
    console.log("Éxito");
  case 404:
    console.log("No encontrado");
  default:
    console.log("Error desconocido");
}
```

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 (let i = 0; i < 10; i++) {
    console.log(`índice ${i}`)
}
```

Ejemplo incorrecto

```
for (let i = 0; i < 10; i++)
    console.log(`índice ${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 (contador < 10) {
    console.log(`Contador: ${contador}`);
    contador++;
}

do {

} while (contador<10)
```

Ejemplo incorrecto

```
while (contador < 10)
{
    console.log(`Contador: ${contador}`);
    contador++;
}

do {

}

while (contador<10)
```

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

En esta sección se hablará sobre el manejo de las excepciones. Excepciones se manejarán en el lugar donde se generen, en lugar de propagarlas hasta la capa de controladores de la interfaz de usuario. De esta forma, cada componente será responsable de controlar y manejar sus propias excepciones, asegurando que se tomen las medidas adecuadas en el punto exacto de falla.

Para el control de excepciones, seguiremos utilizando la estructura try-catch, registrando la causa de la excepción en la bitácora correspondiente utlizando log4net y asignándole una clasificación según el tipo de excepción (Error, Fatal, Information). Además, en lugar de propagar las excepciones a capas superiores, se enviarán valores de retorno correspondientes que permitan manejar los errores de manera contextual.

Al ser una API, se priorizará el retorno de códigos HTTP y mensajes claros frente a excepciones crudas. Pueden atraparse las excepciones directamente en el código que maneje la solicitud, como en los middlewares previos del endpoint.

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:

```
app.post('/login', async (req, res) => {
  try {
    validateEmail(req.body.email);

    const user = await UserService.login(req.body);
    res.json(user);

  } catch (error) {
    if (error instanceof ValidatorError) {
      res.status(400).json({ error: error.message });
    } else {
      logger.error('Error en /login:', error);
      res.status(500).json({ error: 'Error interno' });
    }
  }
});
```

Ejemplo sobre el uso incorrecto del manejo de excepciones:

```
app.post('/login', async (req, res) => {
  try {
    validateEmail(req.body.email);

    const user = await UserService.login(req.body);
    res.json(user);

  } catch (error) {
    if (error instanceof ValidatorError) {
      res.status(200).json({});
    } else {
      logger.error('Error en /login:', error);
      res.status(200).json({});
    }
  }
});
```

Justificación:

Además de que el manejo de excepciones es una buena práctica de programación y ayuda a tener un código de alta calidad. Nos permite poder continuar con la funcionalidad del sistema sin tener que pausarlo o reiniciarlo, además nos ayuda a detectar situaciones indeseables que traten de alterar el comportamiento correcto del sistema, con el fin de ayudar en la corrección de áreas de código o conexión que lleguen a fallar en un momento dado sin empeorar la experiencia de usuario.

8 ES6 Modules

Se utilizarán las palabras clave “import” y “export” con el fin de modularizar el código, mejorar su legibilidad, optimizar el código, garantizar una mayor compatibilidad y cumplir con estándar actual de javascript. Se permite el uso de CommonJS sólo en módulos que no soportan ES6.

Ejemplo válido de importación

```
import sql from 'mssql';
import path from 'path';
import { promises as fs } from 'fs';
import { RetornarTipoDeConexion } from './sql/connection/ConfiguracionConexion.js';
import { MensajeDeRetornoBaseDeDatosAcceso, MensajeDeRetornoBaseDeDatosInfoAdicional } from '../utilidades/Constantes.js';
```

Ejemplo no válido (si el módulo es compatible con ES6):

```
const sql = require('mssql');
const path = require('path');
const fs = require('fs').promises;
const { RetornarTipoDeConexion } = require('./sql/connection/ConfiguracionConexion');
const {
  MensajeDeRetornoBaseDeDatosAcceso,
  MensajeDeRetornoBaseDeDatosInfoAdicional
} = require('../utilidades/Constantes');
```

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.