UNIVERSIDAD PRIVADA DE TACNA

FACULTAD DE INGENIERIA

Escuela Profesional de Ingeniería de Sistemas

Estándar de Programación

Curso: *Construcción de Software II*

Docente: *Mag. Ricardo Eduardo Valcárcel Alvarado*

Integrantes:

Arenas Paz Soldan, Miguel Jesus ***(2017059282)***
Christian Alexander, Céspedes Medina ***(2010036257)***

Tacna – Perú
2025

Sistema *Implementación de un Aplicativo web para la Gestión de Acervos Digitales en el Gobierno Regional de Tacna*
Documento Estándar de Programación del Proyecto

CONTROL DE VERSIONES					
Versión	Hecha por	Revisada por	Aprobada por	Fecha	Motivo
1.0	MAPS	CACM	CACM	17/04/2025	Versión Original

INDICE GENERAL

ESTÁNDAR DE PROGRAMACIÓN BACKEND

1. Introducción

1.1. Contexto General del Proyecto

Este documento tiene como finalidad establecer un conjunto de normas y buenas prácticas para el desarrollo de proyectos en ASP.NET Core, con el objetivo de asegurar que el código sea claro, mantenible, escalable y seguro. Implementar estas pautas mejora la colaboración entre los desarrolladores, eleva la calidad del software y disminuye la posibilidad de errores en ambientes de producción.

El desarrollo de soluciones en .NET Core requiere la aplicación de patrones estructurados, una modularización adecuada, un correcto manejo de errores y un refuerzo en la seguridad. Por esta razón, este documento actuará como una guía de referencia para el equipo, garantizando que todos sigan un estilo uniforme en la escritura y organización del código.

Adoptando estas buenas prácticas, el equipo será capaz de:

- Mantener la uniformidad en la codificación, facilitando su lectura y revisión.
- Minimizar la deuda técnica, evitando soluciones apresuradas que comprometan el desempeño y la escalabilidad.
- Promover el uso de prácticas óptimas en seguridad, en la estructura del proyecto y en el control de errores.
- Mejorar el trabajo colaborativo, asegurando que todos los miembros comprendan fácilmente el flujo del desarrollo de código.

2. Configuración de Entorno

Las credenciales y configuraciones se almacenan en el archivo **appsettings.json** con variantes para diferentes entornos:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"server=localhost;database=SGTD_db;Trusted_Connection=True;TrustServerCertific
ate=True"
  },
  "Jwt": {
    "Key":
"JODcs3UuNZIrhTzkvzeLujZ7ZbipCxv4ZuNWah8xkq1WXuEbrMFBk7j0sT5PmYlJ",
    "Issuer": "jwt_test_issuer",
    "Audience": "www.example.com"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Encryption": {
    "Key": "w7Klp2L5QEqCXV0/qz0Kh6HgJQYAz5FdKpOMKxIY+8Y="
  },
  "Countries": {
    "CountriesApiURL": "https://country.io"
  },
  "FilesPath": {
    "BasePath": "D:\\Descargas\\Users\\Files\\"
  },
  "AllowedHosts": "*"
}
```

Directrices importantes:

- **.NET Core:** Se recomienda utilizar la versión LTS estable.
- **Gestor de paquetes:** Se debe usar Nuget de manera consistente en todo el equipo.
- **Variables de entorno**: Los valores sensibles deben transferirse a variables de entorno en producción.
- **Configuración por entorno:** Utilizar **appsettings.Development.json** y **appsentings.Production.json** para configuraciones específicas.

3. Nomenclatura y Estilo de Código

El Código sigue las siguientes convenciones de nomenclatura:

Elemento	Convención	Ejemplo Correcto	Ejemplo Incorrecto
Modelos (DTOs)	PascalCase + Dto	PersonDto.cs	personDto.cs, Person_Dto.cs
Parámetros de solicitud	PascalCase + RequestParams	PersonRequestParams.cs	personRequestParams.cs
Controladores	PascalCase + Controller	PersonController.cs	personController.cs
Servicios (Interfaces)	IPascalCase	IPersonService.cs	PersonService.cs
Servicios (Implementación)	PascalCase + Service	PersonService.cs	personServices.cs
Entidades	PascalCase	Person.cs	Person.cs
Variables Locales	camelCase	**var userToken = “”;**	**Var Usertoken = “”;**
Constantes	SCREAMING_SNAKE_CASE	**const string API_URL = “…”;**	**const string apiUrl = “…”;**
Funciones y Métodos	PascalCase	**GetUserById(id)**	**getUserById(id)**
Tablas en Base de Datos	PascalCase, plural	People, Projects	Person, Project
Campos en Base de Datos	PascalCase	**{ FirstName: “Christian” }**	**{ firstName: “Christian” }**
Rutas API (RESTfull)	Kebab-case, recursos en plural	GET /api/users, POST /api/tasks	GET /api/User, POST /api/new/task

4. Estructura del Proyecto

El proyecto sigue una estructura modular y organizada para facilitar el mantenimiento y la escalabilidad. A continuación, se describen las carpetas y archivos principales:

Carpeta/Archivo	Descripción
Controllers.cs	Contiene los controladores que exponen los endpoints de la API
Models/	Define los modelos de datos utlizados para transferencia (DTOs) y parámetros de solicitud
DbModel/	Contiene las entidades de la base de datos y el contexto
DbModel/Context/	Configuración del contexto de la base de datos
DbModel/Entities	Define las entidades que se mapean a tablas en la base de datos
Services/	Contiene interfaces e implementaciones que manejan la lógica del negocio
Services/Implementation/	Implementación concreta de los servicios
Middlewars/	Contiene middlewares para la seguridad, validaciones y manejo de errores
Extensions/	Métodos de extensión y configuraciones para el startup
Appsettings.json	Configuraciones generales de la aplicación
Program.cs	Punto de entrada de la aplicación y configuración del hosting
Startup.cs	Configuración de servicios y middleware del pipeline HTTP

5. Modelos de Datos

5.1. DTOs (Data Transfer Objects)

Los DTOs se utilizan para transferir datos entre capas, siguiendo estas convenciones:

```
public class PersonDto
{
 public int? Id { get; set; }

[Required]
[StringLength(100)]
 public string FirstName { get; set; }

[Required]
[StringLength(100)]
public string LastName { get; set; }

[StringLength(20)]
public string Phone { get; set; }

[StringLength(2)]
public string NationalityCode { get; set; }

[StringLength(20)]
public string DocumentNumber { get; set; }
public bool Gender { get; set; }
}
```

5.2. Parámetros de Solicitud

Los parámetros de solicitud se utilizan para recibir datos en las operaciones:

```
public class PersonRequestParams
{
public int Id { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
public string Phone { get; set; }
public string NationalityCode { get; set; }
public string DocumentNumber { get; set; }
public bool Gender { get; set; }
}
```

5.3. Entidades de Base de Datos

Las entidades deben incluir propiedades de auditoría:

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Phone { get; set; }
    public string NationalityCode { get; set; }
    public string DocumentNumber { get; set; }
    public bool Gender { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime? UpdatedAt { get; set; }
}
```

6. Servicios

6.1. Definición de Interfaces

Cada servicio debe tener una interfaz clara que defina sus operaciones:

```
public interface IPersonService
{
    Task CreateAsync(PersonRequestParams requestParams);
    Task UpdateAsync(PersonRequestParams requestParams);
    Task<List<PersonDto>> GetAllAsync();
    Task<PersonDto> GetByIdAsync(int id);
    Task DeleteByIdAsync(int id);
}
```

6.2. Implementación de Servicios

La implementación de servicios debe seguir el patrón de inyección de dependencias:

```
public class PersonService : IPersonService
{
    private readonly DatabaseContext _context;
    public PersonService(DatabaseContext context)
    {
        _context = context;
    }
    // Implementación de los métodos...
}
```

7. Manejo de Errores

El backend implementa un sistema de manejo de errores robusto:

7.1. Errores de Validación

- Usar atributos de validación en los DTOs
- Realizar validaciones explícitas en los servicios

7.2. Excepciones Específicas

```
// Validación de existencia
var personExists = await _context.People
   .Where(q => q.Phone.Equals(requestParams.Phone) ||
q.DocumentNumber.Equals(requestParams.DocumentNumber))
   .AnyAsync();

if (personExists)
{
   throw new ValidationException("Ya existe una persona con el mismo
número de teléfono o DNI");
}

// Validación de recursos
var person = await _context.People.FirstOrDefaultAsync(p => p.Id == id);
if (person == null)
throw new KeyNotFoundException("Person not found.");
```

7.3. Middleware Global

Implementar un middleware para capturar errores y devolver respuestas estandarizadas:

```
app.UseExceptionHandler(appError =>
{
   appError.Run(async context =>
   {
      context.Response.StatusCode =
(int)HttpStatusCode.InternalServerError;
      context.Response.ContentType = "application/json";

      var contextFeature =
context.Features.Get<IExceptionHandlerFeature>();
      if (contextFeature != null)
      {
         // Personalizar respuesta según tipo de excepción
         if (contextFeature.Error is ValidationException)
            context.Response.StatusCode =
(int)HttpStatusCode.BadRequest;
         else if (contextFeature.Error is KeyNotFoundException)
            context.Response.StatusCode = (int)HttpStatusCode.NotFound;

         await context.Response.WriteAsync(new ErrorDetails
         {
            StatusCode = context.Response.StatusCode,
            Message = contextFeature.Error.Message
         }.ToString());
      }
   });
});
```

8. Conexión a la Base de Datos

La conexión a la base de datos es un paso fundamental en el desarrollo de aplicaciones backend. En este proyecto se utiliza SQL Server como base de datos relacional y Entity Framework Core como ORM para interactuar con ella.

Aspecto	Descripción
Archivo	DbModel/Context/DatabaseContext.cs
Base de Datos	SQL SERVER
ORM	Entity Framework Core
Manejo de Errores	Se usa try/catch para manejar errores de conexión
Variables de Entorno	Se almacena la conexión en appsettings.json (ConntecionStrings.DefaultConnection)

```
public class DatabaseContext : DbContext
{
   public DatabaseContext(DbContextOptions<DatabaseContext> options) :
base(options)
   {
   }

   public DbSet<Person> People { get; set; }
   // Otros DbSets...

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
      base.OnModelCreating(modelBuilder);

      // Configuraciones específicas de entidades...
   }
}
```

Configuración en **Program.cs** o **startup.cs**:

```
services.AddDbContext<DatabaseContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
")));
```

9. Seguridad

9.1. Autenticación JWT

Implementar autenticación JWT utilizando la configuración del archivo appsettings.json:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Jwt:Issuer"],
            ValidAudience = Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))
        };
    });
```

9.2. Cifrado de Datos Sensibles

Utilizar la clave de cifrado para datos sensibles:

```
        public static class EncryptionHelper
        {
            public static string Encrypt(string text, string key)
            {
                // Implementación del cifrado usando la clave de
        Configuration["Encryption:Key"]
            }
            public static string Decrypt(string cipherText, string key)
            {
                // Implementación del descifrado
            }}
```

9.3. Protección de Endpoints

Asegurar los endpoints con tributos de autorización:

```
[Authorize]
[HttpGet]
public async Task<ActionResult<IEnumerable<PersonDto>>>
GetAllPersons()
{
    var persons = await _personService.GetAllAsync();
    return Ok(persons);
}
```

10. Asincronía

10.1. Programación Asíncrona

Todos los métodos que interactúan con recursos externos (base de datos, APIs, etc.) deben ser asíncronos:

```
public async Task<List<PersonDto>> GetAllAsync()
{
    return await _context.People
        .Select(p => new PersonDto
        {
            Id = p.Id,
            FirstName = p.FirstName,
            LastName = p.LastName,
            // Otras propiedades...
        })
        .ToListAsync();
}
```

10.2. Convenciones de Nombres

Los métodos asícronos deben terminar con el sufijo “Async”:

- **CreateAsync**
- **GetByIdAsync**
- **UpdateAsync**
- **DeleteByIdAsync**

11. Documentación

11.1. Comentarios de Código

Documentar funcionalidades complejas o no evidentes usando comentarios XML:

```
/// <summary>
/// Obtiene una persona por su identificador.
/// </summary>
/// <param name="id">Identificador de la persona</param>
/// <returns>DTO de la persona encontrada</returns>
/// <exception cref="KeyNotFoundException">Si no existe la persona</exception>
public async Task<PersonDto> GetByIdAsync(int id)
{
    // Implementación...
}
```

11.2. Documentación de API

Configurar Swagger/OpenAPI para documentar endpoints:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "SGTD API", Version = "v1" });

    // Incluir comentarios XML
    var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);

    // Configurar autenticación
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Description = "JWT Authorization header using the Bearer scheme.",
        Name = "Authorization",
        In = ParameterLocation.Header,
```

```
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });

    c.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            new string[] {}
        }
    });
});
```

12. Conclusión y Recomendación

12.1. Conclusión

La adopción del estándar de programación en el proyecto SGTD basado en ASP.NET refleja un compromiso claro con la calidad, la sostenibilidad del código y la excelencia técnica. A lo largo del documento se han establecido lineamientos sobre estructura, seguridad, nomenclatura, manejo de errores y documentación, contribuyendo a un sistema más robusto, mantenible y seguro. La aplicación consistente de estas prácticas permitirá mejorar la colaboración, facilitar el mantenimiento del software, reducir errores y optimizar la incorporación de nuevos integrantes, fortaleciendo así todo el ciclo de vida del desarrollo.

12.2. Recomendación

Para garantizar la implementación efectiva del estándar, se recomienda integrar herramientas de automatización como analizadores de código, configuración de reglas EditorConfig y hooks de pre-commit. A nivel de procesos, es clave fomentar revisiones de código regulares, incluir el estándar en el proceso de incorporación de nuevos miembros y actualizar periódicamente las directrices. Además, se sugiere fortalecer el sistema técnico mediante la adopción de prácticas como el registro estructurado de logs, uso de FluentValidation, manejo más preciso de excepciones, implementación de patrones de diseño y establecimiento de pruebas automatizadas, elevando así la calidad y eficiencia del proyecto.

ESTÁNDAR DE PROGRAMACIÓN FRONTEND

1. Introducción

El propósito de este documento es definir un conjunto de estándares y mejores prácticas para el desarrollo frontend del proyecto basado en Angular, garantizando que el código sea legible, mantenible, escalable y seguro. La aplicación de estas convenciones facilita la colaboración entre desarrolladores, mejora la calidad del software y reduce la probabilidad de errores en producción.

El desarrollo de aplicaciones en Angular implica el uso de componentes estructurados, servicios, modelos de datos y una arquitectura modular. Por lo tanto, este documento servirá como una guía de referencia para el equipo, asegurando que todos sigan una metodología coherente en la escritura y organización del código.

Al seguir estas convenciones, el equipo podrá:

- Asegurar la consistencia en la codificación, facilitando la lectura y revisión del código.
- Reducir la deuda técnica, evitando soluciones improvisadas que puedan afectar el rendimiento y la escalabilidad.
- Fomentar las mejores prácticas en seguridad, estructuración del proyecto y manejo de errores.
- Optimizar la colaboración dentro del equipo, asegurando que cada miembro entienda fácilmente el flujo de trabajo del código.

2. Configuración de Entorno

2.1. Angular y TypeScript

- **Version de Angular:** Se recomienda utilizar la versión LTS estable.
- **TypeScript:** Usar la versión compatible con la versión de Angular seleccionada.
- **Gestor de paquetes:** Usar npm o yarn de manera consistente en todo el equipo.

2.2. Configuración de Entorno

Las configuraciones específicas para cada entorno se almacenan en archivos dedicados:

```
// environments/environment.ts (desarrollo)
export const environment = {
  production: false,
  apiUrl: 'http://localhost:5000/api'
};

// environments/environment.prod.ts (producción)
export const environment = {
  production: true,
  apiUrl: 'https://api.example.com/api'
};
```

2.3. Librería de Componentes

El proyecto utiliza PrimeNG como librería de componentes UI, asegurando una interfaz consistente y moderna:

- Importar solo los módulos necesarios para cada componente
- Personalizar los estilos según las necesidades del proyecto, manteniendo la coherencia visual.

3. Nomenclatura y Estilo de Código

El Código sigue las siguientes convenciones de nomenclatura:

Elemento	Convención	Ejemplo Correcto	Ejemplo Incorrecto
Componentes	PascalCase + Component	AreaListComponent	areaListComponent
Servicios	PascalCase + Service	AreaService	areaService
Modelos	PascalCase + Model	AreaModel	areaModel
Interfaces	IPascalCase o PascalCase	IAreaModel o AreaModel	IAreaModel
Variables Locales	camelCase	**const userToken = ‘’;**	**const UserToken = ‘’;**
Constantes	SCREAMING_SNAKE_CASE	**const ApiURL = ‘…’;**	**const apiURL = ‘…’;**
Funciones y Métodos	camelCase	**getAreaById(id)**	**GetAreaById(id)**
Clases	PascalCase	**class AreaValidator**	**class areaValidator**
Eventos	on + PascalCase	**(onDialogClosed)=”method()”**	**(dialogClosed)=”method()”**
Propiedades de entrada	camelCase	**@Input() areaData**	**@Input() AreaData**
Propiedades de salida	camelCase	**@Output() dialogClosed**	**@Output() DialogClosed**
Archivos de componentes	kebab-case.component.ts	area-list.component.ts	areaList.component.ts
Archivos de servicios	kebab-case.service.ts	area.service.ts	AreaService.ts
Archivos de modelos	kebab-case.model.ts	area.model.ts	Area.Model.ts
Selectores CSS	kebab-case	app-area-list	appAreaList

4. Estructura del Proyecto

El proyecto sigue una estructura modular y organizada para facilitar el mantenimiento y la escalabilidad:

Carpeta/Archivo	Descripción
src/app/	Carpeta principal que contiene los componentes, servicios y módulos
src/app/modules/	Contiene los módulos funcionales de la aplicación
src/app/shared/	Componentes, servicios y utilidades compartidas entre módulos
src/app/core/	Servicios singleton, interceptores HTTP, guardias, etc.
src/assets/	Recursos estáticos como imágenes, fuentes y archivos JSON
src/environments/	Archivos de configuración para diferentes entornos
src/styles/	Estilos globales y variables de temas

4.1. Estructura dentro de un Módulo Funcional

Cada módulo funcional debe seguir esta estructura:

```
modules/areas/
├── components/
│   ├── area-list/
│   │   ├── area-list.component.ts
│   │   ├── area-list.component.html
│   │   ├── area-list.component.scss
│   │   └── area-list.component.spec.ts
│   ├── area-new-edit/
│   └── area-show/
├── services/
│   └── area.service.ts
├── models/
│   └── area.model.ts
└── areas.module.ts
```

5. Modelos de Datos

5.1. Interfaces de Modelo

Las interfaces de modelo deben definir claramente la estructura de los datos:

```
export interface AreaModel {
  id: number;
  name: string;
  description: string;
  status: boolean;
  parentAreaId?: number;
}
```

5.2. Buenas Prácticas

- Definir tipos opcionales con **?** cuando corresponda
- Evitar el uso de **any** siempre que sea posible
- Crear interfaces específicas para cada tipo de datos
- Usar enums para valores predefinidos

6. Servicios

6.1. Estructura de servicios

Los servicios deben ser responsables de la comunicación con el backend y la lógica de negocio:

```
@Injectable({
  providedIn: 'root'
})
export class AreaService {
  private readonly apiUrl = `${environment.apiUrl}/Area`;

  constructor(
    private readonly http: HttpClient,
  ) { }

  createAsync(area: AreaModel): Observable<HttpResponse<void>> {
    return this.http.post<void>(
      `${this.apiUrl}/`,
      {
        name: area.name,
        description: area.description,
        status: area.status,
        parentAreaId: area.parentAreaId
      },
      { observe: 'response' }
```

```
    );
  }

  // Otros métodos...
}
```

6.2. Buenas Prácticas

- Usar la decoración **@Injectable({ providedIn: 'root' })** para servicios singleton
- Devolver Observables para operaciones asíncronas
- Mantener los nombres de métodos consistentes con el backend
- Agregar el sufijo "Async" para métodos que devuelven Observables
- Usar genéricos para tipado fuerte en llamadas HTTP
- Centralizar la construcción de URLs en el servicio

7. Componentes

7.1. Estructura de Componentes

Los componentes siguen un patrón consistente:

```
@Component({
  selector: 'app-area-list',
  standalone: true,
  imports: [
    CommonModule,
    FormsModule,
    // Otros módulos...
  ],
  providers: [ConfirmationService, MessageService],
  templateUrl: './area-list.component.html',
  styleUrl: './area-list.component.scss'
})
export class AreaListComponent {
  // Propiedades y métodos...
}
```

7.2. Buenas Prácticas

- Usar componentes independientes (standalone) cuando sea apropiado
- Separar la lógica de presentación de la lógica de negocio
- Implementar la estrategia de detección de cambios OnPush para componentes grandes
- Limitar la responsabilidad de cada componente a una funcionalidad específica
- Usar ViewChild para acceder a componentes hijos
- Implementar ngOnDestroy para cancelar suscripciones

8. Manejo de Estado y RxJS

8.1. Patrones para RxJS

```
unsubscribe$ = new Subject<void>();

loadAreas() {
this.spinnerPrimeNgService
.use(this.areaService.getAllAsync())
.pipe(takeUntil(this.unsubscribe$))
.subscribe({
next: (res) => {
this.areas = res;
  }
});
}

ngOnDestroy() {
this.unsubscribe$.next();
this.unsubscribe$.complete();
}
```

8.2. Buenas Prácticas

- Usar el patrón Subject/takeUntil para gestionar suscripciones
- Implementar correctamente ngOnDestroy para evitar fugas de memoria
- Usar operadores RxJS adecuados (map, filter, catchError, etc.)
- Manejar errores explícitamente en las suscripciones
- Considerar el uso de async pipe en las plantillas cuando sea apropiado

9. Estilos y CSS

9.1. Estructura de SCSS

Los estilos SCSS deben estar organizados y seguir un patrón consistente:

```
.content-block {
    font-family: 'Roboto', sans-serif;
    margin-top: 1.5rem;
    margin-left: 1.5rem;
    margin-right: 1.5rem;
}

.row-block {
    display: flex;
    flex-direction: row;
    justify-content: space-between;
    margin-bottom: 2rem;

    >* {
        padding: 0 0.5rem;
    }

    > :first-child {
        padding-left: 0;
    }

    > :last-child {
        padding-right: 0;
    }
}

.row-item-block-100 {
    width: 100%;
}

.row-item-block-60 {
    width: 60%;
}
```

9.2. Buenas Prácticas

- Utilizar encapsulación de estilos por componente
- Seguir un enfoque de nomenclatura BEM (Block Element Modifier) o similar
- Evitar estilos ¡important salvo cuando sea absolutamente necesario
- Utilizar variables SCSS para colores, espaciados y otros valores recurrentes
- Implementar diseño responsive

10. Manejo de Formularios

10.1. Formularios Reactivos vs Template-Driven

El proyecto utiliza formularios Template-Driven, pero se pueden implementar formularios Reactivos para casos más complejos.

```
        // Ejemplo de formulario Template-Driven
        <p-floatLabel>
          <input id="name" type="text" pInputText [(ngModel)]="area.name"
        required>
          <label for="name">Nombre</label>
</p-floatLabel>
```

10.2. Validaciones

- Implementar validaciones tanto en el frontend como en el backend
- Mostrar mensajes de error claros y específicos
- Deshabilitar botones de envío cuando el formulario no sea válido
- Usar decoradores como @Required para validaciones básicas

11. Comunicación entre Componentes

11.1. @Input() y @ Output()

```
// Componente hijo
@Component({
  selector: 'app-area-new-edit',
  // ...
})
export class AreaNewEditComponent {
  @Output() dialogClosed = new EventEmitter<boolean>();

  hideDialog() {
    this.visible = false;
    this.dialogClosed.emit(this.result);
  }
}

    // Componente padre (en HTML)
<app-area-new-edit    (dialogClosed)="onDialogClosed($event)"></app-area-
new-edit>
```

11.2. Servicios Compartidos

Para comunicación entre componentes no relacionados directamente, usar servicios con BehaviorSubject:

```
@Injectable({
  providedIn: 'root'
})
export class SharedDataService {
  private dataSubject = new BehaviorSubject<any>(null);
  public data$ = this.dataSubject.asObservable();

  updateData(data: any) {
    this.dataSubject.next(data);
  }
}
```

12. Manejo de Errores y Notificaciones

12.1. Patrón de Manejo de Errores

```
this.areaService.createAsync(this.area)
  .pipe(
    catchError(error => {
      this.messageService.add({
        severity: 'error',
        summary: 'Error',
        detail: 'No se pudo crear el área'
      });
      return throwError(() => error);
    })
  )
  .subscribe({
    next: () => {
      this.messageService.add({
        severity: 'success',
        summary: 'Éxito',
        detail: 'Área creada correctamente'
      });
      this.hideDialog();
    }
  });
```

12.2. Sistema de Notificaciones

El proyecto utiliza PrimeNG Toast para las notificaciones:

```
<p-toast />

<!-- En el componente TS -->
this.messageService.add({
  severity: 'success',
  summary: 'Éxito',
  detail: 'Registro guardado correctamente',
  life: 3000
});
```

13. Optimización de Rendimiento

13.1. Estrategias de Optimización

- Usar lazy loading para cargar módulos bajo demanda
- Implementar ChangeDetectionStrategy.OnPush para componentes grandes
- Utilizar trackBy en ngFor para mejorar el rendimiento de las listas
- Minimizar el número de suscripciones y cancelarlas adecuadamente
- Utilizar memoización para cálculos costosos

Ejemplo de trackBy

```
<tr *ngFor="let area of areas; trackBy: trackById">
  <!-- Content -->
</tr>

<!-- En el componente TS -->
trackById(index: number, item: AreaModel): number {
  return item.id;
}
```

14. Pruebas Unitarias

14.1. Estructura de Pruebas

```
describe('AreaListComponent', () => {
  let component: AreaListComponent;
  let fixture: ComponentFixture<AreaListComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [AreaListComponent]
    })
    .compileComponents();

    fixture = TestBed.createComponent(AreaListComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
    // Más casos de prueba...
});
```

14.2. Buenas Prácticas

- Escribir pruebas para todos los componentes y servicios
- Usar mocks para servicios y dependencias externas
- Probar los caminos felices y los casos de error
- Mantener las pruebas independientes entre sí
- Implementar pruebas de integración para flujos completos

15. Conclusión y Recomendación

15.1. Conclusión

La implementación de este estándar de programación en el frontend de Angular es fundamental para garantizar la calidad, seguridad, escalabilidad y mantenibilidad del código. Siguiendo las convenciones establecidas en este documento, logramos que el desarrollo sea más organizado, facilitando la colaboración entre los miembros del equipo y reduciendo la probabilidad de errores y vulnerabilidades en la aplicación.
Este estándar no debe verse como una restricción, sino como una herramienta para mejorar el desarrollo, optimizando tiempos de trabajo y evitando errores comunes. Con este estándar, aseguramos que el código del proyecto sea más robusto, mantenible y eficiente a lo largo del tiempo, permitiendo una mejor experiencia tanto para los desarrolladores como para los usuarios finales.

15.2. Recomendación

Para fortalecer aún más la implementación del estándar de programación en Angular, se recomienda integrar herramientas de análisis estático de código como ESLint y Prettier configuradas específicamente para Angular, además de establecer revisiones de código periódicas dentro del equipo. También sería valioso fomentar la adopción progresiva de patrones avanzados como Smart/Dumb Components y la implementación de librerías de manejo de estado como NgRx en proyectos de mayor complejidad. Finalmente, es importante actualizar este estándar cada cierto tiempo, incorporando buenas prácticas emergentes y adaptándolo a las nuevas versiones de Angular, garantizando así que el proyecto se mantenga alineado con las mejores prácticas de la industria.