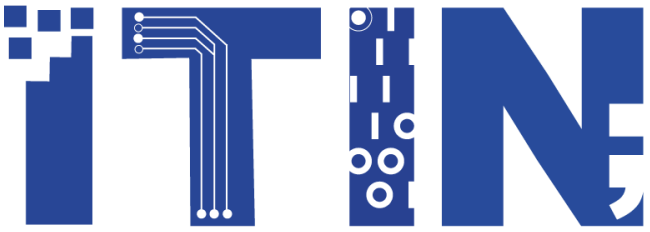
UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN



"Proyecto Primer Parcial"

Aplicaciones Distribuidas 3877

Carol Dayanara Jácome Sanmartin

Steven Alejandro Oviedo Buitrón

Sangolquí, 14 de diciembre de 2024

ÍNDICE

I. Objetivo General 3
II. Objetivos Específicos 3
III. Marco Teórico 3
IV. Especificación de Requisitos Funcionales y No Funcionales 4
A. Requisitos Funcionales 4
B. Requisitos No Funcionales 5
V. Diagrama de Arquitectura 6
VI. Diagrama de Flujos 7
VII. Guía de Implementación 7
VIII. Guía de Implementación 10
IX. Matriz de Roles y Permisos 13
X. Manejo de Errores 13
XI. Documentación para Pruebas 16
Referencias 18

I. Objetivo General

Diseñar un sistema web MVC integrado con una API RESTful para gestionar categorías y productos, utilizando ASP.NET Framework y Entity Framework, con el fin de facilitar el manejo eficiente de la información en una base de datos centralizada.

II. Objetivos Específicos

- Implementar el módulo de autenticación y autorización, utilizando Identity Framework, para garantizar la seguridad en el acceso al sistema.
- Desarrollar controladores y vistas en ASP.NET MVC, aprovechando la comunicación con la API REST, para realizar operaciones CRUD sobre categorías y productos.
- Configurar la API RESTful con Entity Framework y patrones de diseño, asegurando una arquitectura modular y reutilizable, para optimizar el manejo de datos.
- Incorporar un sistema de roles y permisos, administrado desde la interfaz gráfica, para definir y restringir las funcionalidades según los niveles de acceso.
- Documentar los requisitos funcionales y no funcionales del sistema, con base en estándares actuales de desarrollo web, para facilitar su implementación y mantenimiento.

III. Marco Teórico

En el desarrollo de aplicaciones web modernas, la arquitectura Modelo-Vista-Controlador (MVC) se ha consolidado como un estándar para la separación de preocupaciones, facilitando el mantenimiento y escalabilidad de los sistemas. ASP.NET MVC, un framework proporcionado por Microsoft, implementa este patrón permitiendo a los desarrolladores construir aplicaciones web de manera estructurada y eficiente. Este enfoque segmenta la aplicación en tres componentes principales: el Modelo, que representa la lógica de negocio y los datos; la Vista, encargada de la presentación y la interfaz de usuario; y el Controlador, que actúa como intermediario gestionando la comunicación entre el modelo y la vista.

Complementando esta arquitectura, Entity Framework (EF) se erige como un Object-Relational Mapper (ORM) que simplifica la interacción con bases de datos relacionales. EF permite a los desarrolladores trabajar con datos en forma de objetos de dominio específicos, eliminando la necesidad de escribir gran cantidad de código SQL. Esto se traduce en un desarrollo más ágil y menos propenso a errores, ya que el mapeo entre las entidades del dominio y las tablas de la base de datos se gestiona de manera automática.

La integración de ASP.NET MVC con Entity Framework facilita la implementación de operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en aplicaciones web. Mediante el uso de scaffolding, es posible generar automáticamente controladores y vistas que interactúan con el modelo de datos, acelerando el proceso de desarrollo y asegurando una coherencia en la manipulación de la información.

En el contexto de la comunicación entre aplicaciones, las APIs RESTful han emergido como una solución efectiva para la interoperabilidad. REST (Representational State Transfer) es un estilo arquitectónico que utiliza los protocolos HTTP para facilitar la

interacción entre sistemas. Las APIs RESTful permiten la exposición de recursos a través de URLs, utilizando métodos HTTP estándar como GET, POST, PUT y DELETE para realizar operaciones sobre dichos recursos. Esta metodología promueve una comunicación stateless y escalable, siendo ampliamente adoptada en el desarrollo de servicios web.

La combinación de ASP.NET MVC, Entity Framework y APIs RESTful permite la creación de sistemas robustos y escalables. Por ejemplo, en el desarrollo de un sistema de gestión de inventarios, ASP.NET MVC puede manejar la interfaz de usuario y la lógica de presentación, mientras que Entity Framework gestiona el acceso y manipulación de datos. Simultáneamente, una API RESTful puede facilitar la integración con otros sistemas o aplicaciones móviles, permitiendo operaciones como la consulta de stock o la actualización de productos de manera eficiente.

La implementación de estas tecnologías requiere una planificación cuidadosa y una comprensión profunda de cada componente. Es esencial considerar aspectos como la seguridad, el manejo de errores y la validación de datos para garantizar la integridad y confiabilidad del sistema. Además, la documentación adecuada y la realización de pruebas exhaustivas son fundamentales para asegurar que el sistema cumpla con los requisitos funcionales y no funcionales establecidos.

IV. Especificación de Requisitos Funcionales y No Funcionales

A. Requisitos Funcionales

- ***Gestión de Categorías***
  - El sistema debe permitir al usuario crear nuevas categorías especificando un nombre y una descripción.
  - El sistema debe permitir al usuario editar categorías existentes, actualizando su nombre y descripción.
  - El sistema debe permitir al usuario eliminar categorías, siempre y cuando cumpla con los criterios establecidos (si aplica).
  - El sistema debe listar todas las categorías disponibles en la base de datos.
- ***Gestión de Productos***
  - El sistema debe permitir al usuario crear productos especificando su nombre, categoría, precio y cantidad en stock.
  - El sistema debe permitir al usuario editar productos existentes, actualizando sus datos (nombre, categoría, precio y stock).
  - El sistema debe permitir al usuario eliminar productos de la base de datos.
  - El sistema debe listar todos los productos disponibles, mostrando el nombre, la categoría, el precio y el stock.
- ***Interacción con APIs RESTful***
  - El sistema debe comunicarse con las APIs RESTful para realizar operaciones CRUD sobre categorías y productos.
  - El sistema debe consumir las APIs para obtener la lista de categorías y productos desde el servidor.
- ***Gestión de Usuarios y Roles***

- El sistema debe permitir el registro de nuevos usuarios.
- El sistema debe permitir la autenticación y autorización de usuarios según roles asignados (por ejemplo, administrador).
- El sistema debe permitir asignar roles a usuarios existentes.

- ***Validación de Datos***
  - El sistema debe validar los datos ingresados por el usuario en los formularios antes de enviarlos al servidor.
  - El sistema debe notificar al usuario en caso de errores en el ingreso de datos (por ejemplo, campos requeridos o formatos inválidos).

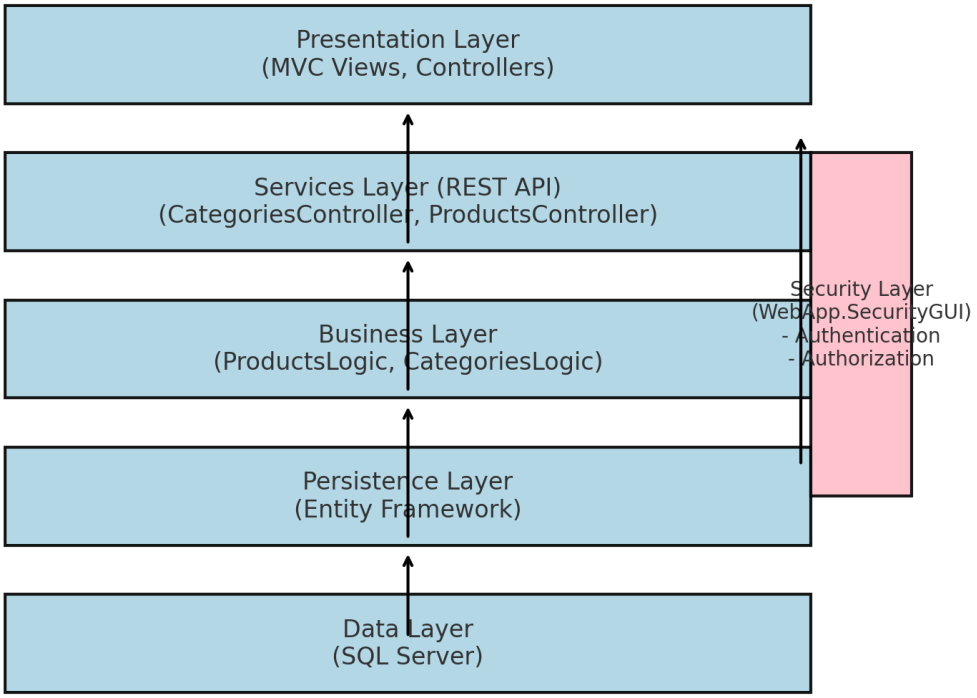
B. Requisitos No Funcionales

- ***Compatibilidad***
  - El sistema debe ser compatible con navegadores modernos, como Google Chrome, Mozilla Firefox y Microsoft Edge.
  - El sistema debe funcionar sobre la plataforma .NET Framework 4.7.2.
- ***Escalabilidad***
  - El sistema debe poder manejar un crecimiento en el número de categorías, productos y usuarios sin afectar significativamente el rendimiento.
- ***Rendimiento***
  - Las operaciones CRUD deben completarse en un tiempo razonable (menos de 2 segundos en condiciones normales).
  - El sistema debe cargar las listas de categorías y productos en menos de 3 segundos.
- ***Interfaz de Usuario***
  - La interfaz debe ser intuitiva y fácil de usar, siguiendo un diseño basado en Bootstrap para garantizar la compatibilidad en diferentes tamaños de pantalla.
- ***Seguridad***
  - El sistema debe usar tokens de verificación antifalsificación (AntiForgeryToken) para prevenir ataques CSRF.
  - Las contraseñas de los usuarios deben ser almacenadas de forma segura, utilizando hashing y salting.
  - Las APIs RESTful deben estar protegidas para evitar accesos no autorizados.
- ***Mantenibilidad***
  - El código del sistema debe estar estructurado y seguir buenas prácticas de desarrollo, como el uso de controladores separados y una arquitectura MVC.
  - La documentación del sistema debe estar disponible para facilitar futuras modificaciones y actualizaciones.
- ***Pruebas y Validación***

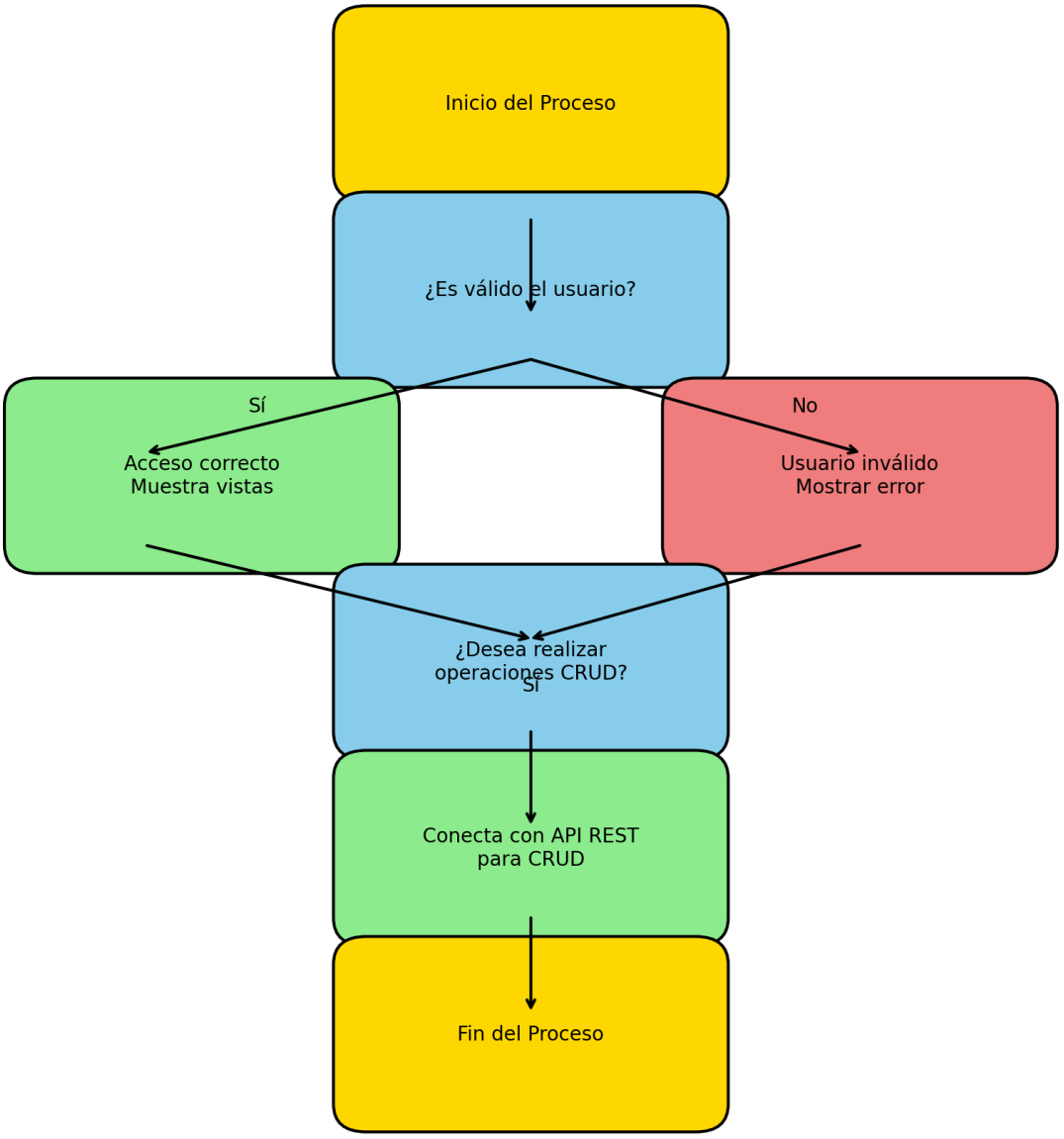
- El sistema debe ser probado exhaustivamente antes de su despliegue, cubriendo casos de prueba funcionales y no funcionales.

- ***Conexión a la Base de Datos***
  - El sistema debe conectarse a una base de datos SQL Server mediante Entity Framework para gestionar la persistencia de datos.
  - La base de datos debe ser capaz de manejar múltiples consultas simultáneamente sin pérdida de rendimiento.

V. Diagrama de Arquitectura



VI. Diagrama de Flujos.



VII. Guía de Implementación.

Paso 1: Configuración del Entorno de Desarrollo

Instalar Visual Studio 2022 (o versión equivalente):

- Seleccionar la carga de trabajo para desarrollo web y .NET.
- Asegurarse de incluir ASP.NET y Entity Framework en las opciones.

Configurar SQL Server:

- Instalar SQL Server Management Studio (SSMS).
- Crear una base de datos llamada Sales_DB.

Paso 2: Creación del Proyecto ASP.NET MVC

Crear una nueva solución en Visual Studio:

Seleccionar "Aplicación Web ASP.NET (.NET Framework)".

Elegir la plantilla de "MVC".

Configurar el proyecto para HTTPS.

Añadir las Capas del Proyecto:

- WebApp.SecurityGUI: Manejo de seguridad y autenticación.
- Service: Implementación de las APIs REST.
- BLL (Business Logic Layer): Lógica de negocio para CRUD de Productos y Categorías.
- DAL (Data Access Layer): Comunicación con la base de datos.
- Entities: Modelos de datos representados en clases.

Paso 3: Configuración de Seguridad

Integrar ASP.NET Identity:

- Configurar ApplicationUser y ApplicationDbContext en el proyecto de seguridad.
- Definir las reglas para autenticación y autorización en RolesController.
- Crear las vistas y controladores para autenticación:
  - Vistas: Registro, Inicio de Sesión y Asignación de Roles.
  - Métodos en RolesController para asignar roles y gestionar usuarios.

Configuración del web.config:

- Asegurarse de que las rutas están protegidas según los roles definidos.

Paso 4: Implementación de las APIs REST

Configurar la clase WebApiConfig en la capa Service:

```
config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

    name: "DefaultApi",

    routeTemplate: "api/{controller}/{id}",

    defaults: new { id = RouteParameter.Optional }

);
```

Crear Controladores de Categorías y Productos:

- CategoriesController: Métodos para CRUD (GetAll, GetById, Create, Update, Delete).
- ProductsController: Similar a CategoriesController, adaptado para productos.

Probar los endpoints con Postman:

- GET: http://localhost:56104/api/Categories
- POST: Crear una categoría con JSON.
- Validar respuestas HTTP para todos los métodos.
- Paso 5: Implementación de la Interfaz de Usuario
- Diseñar las vistas para Categorías y Productos:
- Crear las vistas Index, Create, Edit y Delete en las carpetas Views/Categories y Views/Products.

Configuración de los Controladores:

- CategoriesController y ProductsController: Conexión con las APIs REST utilizando HttpClient.
- Incluir validación de formularios y manejo de errores.

Añadir las rutas en el layout principal:

- Actualizar _Layout.cshtml para incluir los enlaces a Categorías y Productos:

```
<li>@Html.ActionLink("Categorías", "Index", "Categories")</li>

<li>@Html.ActionLink("Productos", "Index", "Products")</li>
```

Paso 6: Configuración de la Base de Datos

Crear las Tablas:

- Ejecutar el script SQL generado por Entity Framework para las tablas Categories y Products.

Verificar Relaciones:

- La tabla Products debe tener una clave foránea a Categories.

Configurar la Cadena de Conexión:

- Actualizar el archivo Web.config:

```
<connectionStrings>

          <add   name="Sales_DBEntities"   connectionString="Data
Source=.\SQLEXPRESS;Initial          Catalog=Sales_DB;Integrated
Security=True;" providerName="System.Data.SqlClient" />

</connectionStrings>
```

Paso 7: Configuración de Roles y Permisos

Crear Roles en la Base de Datos:

- Utilizar RoleManager para agregar roles como "Admin" y "User".
- Asignar roles a los usuarios mediante la vista Roles.

Restricción de Acceso:

- En cada controlador, usar decoradores [Authorize(Roles = "Admin")] para restringir funciones según roles.

Paso 8: Pruebas de Funcionalidad

Pruebas Unitarias:

- Crear un proyecto de prueba con NUnit o MSTest.
- Probar métodos en las capas BLL y Service.

Pruebas de Integración:

- Probar la interacción completa desde la interfaz hasta la base de datos.

Validación de Seguridad:

- Probar rutas protegidas y validación de usuarios no autenticados.

VIII. Guía de Implementación.

Requisitos del Sistema

Hardware:

- Procesador: Intel Core i5 o superior.
- RAM: 8 GB mínimo (16 GB recomendado).
- Espacio en disco: 10 GB libres para el proyecto y herramientas.

Software:

- Sistema operativo: Windows 10/11.
- Visual Studio 2022 con las cargas de trabajo:
- Desarrollo web y .NET.
- SQL Server 2019 o superior.
- Postman para pruebas de las APIs REST.
- IIS (Internet Information Services).
- Configuración del Proyecto en Visual Studio
- Abrir Visual Studio y cargar la solución.

Asegurarse de que cada capa está en su propio proyecto:

- WebApp.SecurityGUI: Seguridad y autenticación.
- Service: APIs REST.
- BLL: Lógica de negocio.
- DAL: Acceso a datos.
- Entities: Modelos.
- Configurar el proyecto principal (WebApp.SecurityGUI) como el Startup Project:
- Hacer clic derecho sobre el proyecto en el Explorador de Soluciones.
- Seleccionar "Establecer como proyecto de inicio".

Configurar el puerto del servidor de desarrollo local (IIS Express):

- Hacer clic derecho en el proyecto > Propiedades > Depurar > URL de la aplicación.
- Ejemplo: https://localhost:44303.

Configuración de la Base de Datos

- Creación de la Base de Datos
- Abrir SQL Server Management Studio (SSMS).

Crear una nueva base de datos llamada Sales_DB:

```
CREATE DATABASE Sales_DB;
```

Crear las tablas necesarias ejecutando el script generado por Entity Framework:

```
CREATE TABLE Categories (
    CategoryID INT PRIMARY KEY IDENTITY,
    CategoryName NVARCHAR(100) NOT NULL,
    Description NVARCHAR(250)
);
CREATE TABLE Products (
    ProductID INT PRIMARY KEY IDENTITY,
    ProductName NVARCHAR(100) NOT NULL,
                CategoryID    INT    FOREIGN    KEY    REFERENCES
Categories(CategoryID),
    UnitPrice DECIMAL(10,2),
    UnitsInStock INT
);
```

Cadena de Conexión

- Configurar la conexión en el archivo Web.config de WebApp.SecurityGUI:

```
<connectionStrings>
    <add name="Sales_DBEntities"
              connectionString="Data  Source=.\SQLEXPRESS;Initial
Catalog=Sales_DB;Integrated Security=True;"
         providerName="System.Data.SqlClient" />
</connectionStrings>
```

Probar la conexión en SSMS:

- Conectar al servidor.
- Ejecutar una consulta de prueba: SELECT * FROM Categories.

Configuración de Seguridad

Roles y Permisos

Configurar roles en ASP.NET Identity:

- Roles predeterminados: Admin, User.
- Scripts iniciales para la base de datos:

```
var roleManager = new RoleManager<IdentityRole>(new RoleStore<IdentityRole>(context));
if (!roleManager.RoleExists("Admin"))
{
    roleManager.Create(new IdentityRole("Admin"));
}
```

Restringir acceso por roles en controladores:

```
[Authorize(Roles = "Admin")]
public ActionResult AdminOnly()
{
    return View();
}
```

Configuración de Cookies

Configurar cookies en Startup.Auth.cs:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
    LoginPath = new PathString("/Account/Login"),
    ExpireTimeSpan = TimeSpan.FromMinutes(30)
});
```

Proteger rutas críticas con [ValidateAntiForgeryToken].

Configuración de Pruebas

Postman

Configurar las solicitudes:

- GET http://localhost:56104/api/Categories
- POST http://localhost:56104/api/Categories
- PUT http://localhost:56104/api/Categories/{id}
- DELETE http://localhost:56104/api/Categories/{id}.

Automatización de Pruebas

Implementar pruebas unitarias para BLL con MSTest:

```
[TestMethod]
```

```
public void Test_CreateCategory()

{

    var logic = new CategoriesLogic();

      var category = new Categories { CategoryName = "Test",
Description = "Testing" };

    var result = logic.Create(category);

    Assert.IsNotNull(result);

}
```

Probar rutas protegidas con usuarios autenticados:

- Crear un usuario de prueba y verificar acceso a recursos protegidos.

IX. Matriz de Roles y Permisos.

Módulo/Funcionalidad	Create	View	Edit	Delete	Admin
Gestión de Categorías					
Crear categorías	✔ (Permitido)	✘ (Denegado)	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)
Visualizar categorías	✘ (Denegado)	✔ (Permitido)	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)
Editar categorías	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)	✘ (Denegado)	✔ (Permitido)
Eliminar categorías	✘ (Denegado)	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)	✔ (Permitido)
Gestión de Productos					
Crear productos	✔ (Permitido)	✘ (Denegado)	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)
Visualizar productos	✘ (Denegado)	✔ (Permitido)	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)
Editar productos	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)	✘ (Denegado)	✔ (Permitido)
Eliminar productos	✘ (Denegado)	✘ (Denegado)	✘ (Denegado)	✔ (Permitido)	✔ (Permitido)

X. Manejo de Errores.

En la Capa de Presentación

Validación del Lado del Cliente:

- Implementa validaciones con JavaScript o frameworks como jQuery para evitar errores comunes, como campos vacíos o valores incorrectos, antes de enviar datos al servidor.
- Utiliza mensajes de error descriptivos y estilizados para guiar al usuario.

```
<div class="form-group">

    @Html.TextBoxFor(model => model.CategoryName, new { @class =
"form-control" })

     @Html.ValidationMessageFor(model => model.CategoryName, "",
new { @class = "text-danger" })

</div>
```

Manejo de Errores Genéricos:

- Agrega una página personalizada para manejar errores HTTP (como 404 y 500) utilizando el archivo Web.config.

```
<customErrors mode="On" defaultRedirect="Error">
```

```
    <error statusCode="404" redirect="NotFound" />
    <error statusCode="500" redirect="ServerError" />
</customErrors>
```

En la Capa de Seguridad

Control de Sesiones Expiradas:

- Verifica que el token de autenticación o la sesión del usuario esté activa antes de procesar cualquier solicitud.
- Redirige a los usuarios a la página de inicio de sesión si se detecta una sesión inválida.

Bloqueo de Acceso No Autorizado:

Configura mensajes claros cuando un usuario intenta acceder a recursos para los que no tiene permisos.

```
[Authorize(Roles = "Admin")]
public ActionResult AdminPage()
{
    return View();
}
```

En la Capa de Servicios REST (API)

Validación de Entradas:

- Verifica los datos que llegan a la API antes de procesarlos.

```
if (categories == null || string.IsNullOrWhiteSpace(categories.CategoryName))
       return BadRequest("Los datos de la categoría no son válidos.");
```

Respuestas Estandarizadas:

- Asegúrate de devolver respuestas HTTP claras y consistentes (códigos 200, 400, 404, 500).

```
public IHttpActionResult CreateProduct(Product product)
{
    if (product == null)
        return BadRequest("El producto no puede ser nulo.");

    try
```

```
    {
        var createdProduct = productLogic.Create(product);
            return Created($"api/Products/{createdProduct.Id}",
createdProduct);
    }
    catch (Exception ex)
    {
        return InternalServerError(ex);
    }
}
```

En la Capa de Negocios

Manejo de Excepciones:

- Utiliza bloques try-catch para capturar y manejar errores inesperados.

```
try
{
    var updated = productLogic.Update(product);
    if (!updated)
            throw new Exception("No se pudo actualizar el
producto.");
}
catch (Exception ex)
{
        throw new BusinessException("Error en la lógica de
negocios", ex);
}
```

Validaciones de Negocios:

- Asegúrate de validar todas las reglas de negocio antes de interactuar con la base de datos o la API.

5. En la Capa de Persistencia

Validación de Conexiones a la Base de Datos:

- Verifica la disponibilidad de la base de datos antes de realizar operaciones.

```
if (!dbContext.Database.Exists())
```

throw new Exception("La base de datos no está disponible.");

Manejo de Excepciones del ORM:

- Captura y registra errores específicos de Entity Framework.

```
try
{
    dbContext.Products.Add(product);
    dbContext.SaveChanges();
}
catch (DbUpdateException ex)
{
    throw new PersistenceException("Error al guardar en la base de datos", ex);
}
```

Estrategias Generales

Registro de Errores (Logging):

- Utiliza herramientas como log4net o Serilog para registrar errores.

```
log.Error("Ocurrió un error en la aplicación", ex);
```

Manejo Global de Excepciones:

- Configura un filtro global para manejar errores no controlados.

```
protected void Application_Error(object sender, EventArgs e)
{
    Exception exception = Server.GetLastError();
    // Loguea el error y redirige a una página de error.
    Response.Redirect("/Error");
}
```

XI. Documentación para Pruebas.

Herramientas Utilizadas

Postman

- Se usó para probar las API de CategoriesController y ProductsController.
- Validó las operaciones CRUD (Create, Read, Update, Delete).
- Probó escenarios como respuestas con éxito, errores de validación y accesos no autorizados.

Interfaz de Usuario (UI)

- Se realizaron pruebas directas en la aplicación web para validar:
- Formularios y flujos de trabajo.
- Acciones como creación, edición y eliminación de productos y categorías.
- Validaciones en cliente y servidor.

Escenarios de Prueba

Pruebas con Postman:

- GET /api/Categories: Obtener la lista de categorías.
- Resultado Esperado: Retorna un JSON con todas las categorías disponibles.
- POST /api/Products: Crear un nuevo producto.
- Entrada: JSON con los datos del producto.
- Resultado Esperado: Código 201 (Creado) y el producto agregado.
- PUT /api/Categories/{id}: Editar una categoría existente.
- Resultado Esperado: Código 200 (OK) y categoría actualizada.
- DELETE /api/Products/{id}: Eliminar un producto.
- Resultado Esperado: Código 200 (OK) y producto eliminado.

Pruebas con la Interfaz (UI)

- Crear categoría desde el formulario.
- Acción: Completar los campos obligatorios y guardar.
- Resultado Esperado: La nueva categoría aparece en la lista.
- Editar un producto existente.
- Acción: Modificar los datos y guardar.
- Resultado Esperado: Los cambios son visibles en la tabla.
- Intentar eliminar una categoría sin permisos.
- Resultado Esperado: Mensaje de "Acceso denegado".

Resultados de las Pruebas

Postman:

- Todos los endpoints respondieron correctamente en escenarios válidos.
- Los errores como IDs inexistentes o entradas inválidas devolvieron mensajes claros.

Interfaz de Usuario

- Las acciones de los botones (crear, editar, eliminar) funcionaron como se esperaba.
- La validación en el cliente impidió enviar formularios incompletos.
- Los mensajes de éxito y error mejoraron la experiencia del usuario.

Referencias

Marcia, E. (2021). Api Rest: Crear una Api Rest con Asp.Net MVC y C#. Recuperado de

https://erickmarcia.github.io/blog/Crear-una-Api-Rest-con-AspNet-MVC-y-CSharp/

Microsoft. (2024). Tutorial: Implementar la funcionalidad CRUD con Entity Framework en ASP.NET MVC. Recuperado de

https://learn.microsoft.com/es-es/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application

Rootstack. (2023). Desarrollando APIs RESTful con .NET Framework: Mejores prácticas y casos de uso. Recuperado de

https://cms.rootstack.com/es/es/blog/desarrollando-apis-restful-con-net-framework-mejores-practicas-y-casos-de-uso

Estrada Web Group. (2017). Sistema de control de inventarios con ASP.NET MVC Parte 1 (Diseño de la base de datos). Recuperado de

https://estradawebgroup.com/Post/Sistema-de-contol-de-inventarios-con-ASP-NET-MVC-Parte-1--Diseno-de-la-base-de-datos-/4258