**Puntos Favorables en SecureClose**

**Código Comentado**

El código comentado es esencial para facilitar la comprensión y colaboración entre los miembros del equipo de desarrollo.

**Código Asíncrono**

El uso de código asíncrono mejora la capacidad de respuesta y la eficiencia del software al permitir que múltiples operaciones se ejecuten de manera concurrente.

**Configuraciones en el Web.config**

Al mantener las configuraciones en el archivo Web.config, se logra una mayor flexibilidad y facilidad en la gestión de cambios y configuraciones del sistema. Microsoft aboga por el uso de archivos de configuración externos, como el Web.config, para separar las configuraciones del código fuente y permitir una gestión más sencilla de configuraciones en diferentes entornos.

**Uso de ORM para Conectar a Base de Datos**

El uso de un Mapeador Objeto-Relacional (ORM) simplifica la interacción con bases de datos al proporcionar una capa de abstracción entre el código de la aplicación y las consultas de la base de datos. Entity Framework, el ORM recomendado por Microsoft, simplifica el acceso y la manipulación de datos, lo que resulta en un desarrollo más rápido y menos propenso a errores.

**Utilización de Features de Azure**

Aprovechar las características y servicios de Azure, como Azure App Service, Azure Storage y Azure Functions, proporciona escalabilidad, alta disponibilidad y seguridad a las aplicaciones.

**Uso de Async con Task**

El uso de async y Task para manejar operaciones asíncronas es una práctica recomendada por Microsoft para mejorar la capacidad de respuesta de las aplicaciones. Task es un tipo que representa una operación asincrónica, permitiendo un manejo más eficiente de la concurrencia y la programación asíncrona.

**Puntos Desfavorables en SecureClose**

**No se utilizan los principios S.O.L.I.D**

No utilizar los principios SOLID puede llevar a una serie de problemas y dificultades en el desarrollo de software. Aquí hay algunas razones por las cuales no seguir estos principios podría ser perjudicial:

**-Dificultades de Mantenimiento:** La falta de adherencia a los principios SOLID puede resultar en código difícil de entender. Esto hace que el mantenimiento y la extensión del software sean complicados y propensos a errores.

**-Rigidez del Código:** Si el código no sigue los principios SOLID, es probable que esté altamente acoplado, lo que significa que un cambio en una parte del código puede tener efectos no deseados en otras partes. Esto hace que el código sea rígido y difícil de adaptar a nuevos requisitos.

**-Dificultades en la Reutilización:** Los principios SOLID fomentan la reutilización de componentes y módulos de código. Si no se siguen estos principios, es más probable que los componentes sean difíciles de reutilizar en diferentes partes del sistema.

**-Escalabilidad Limitada:** El código que no sigue los principios SOLID tiende a ser más difícil de escalar para manejar cargas de trabajo crecientes.

**-Dificultades en la Colaboración:** Diferentes desarrolladores pueden interpretar el código de manera diferente, lo que puede llevar a inconsistencias y errores.

**-Mayor Riesgo de Errores:** La falta de encapsulación y cohesión puede hacer que sea más difícil identificar y corregir problemas.

**-Dificultades en las Pruebas:** El código que no sigue los principios SOLID es más difícil de probar, ya que las dependencias no se pueden inyectar fácilmente y las unidades individuales no son independientes para las pruebas unitarias.

**-Mayor Costo a Largo Plazo:** Aunque puede parecer más rápido desarrollar inicialmente sin seguir los principios SOLID, esto a menudo resulta en costos más altos a largo plazo debido a la dificultad de mantenimiento, reparación de errores y adaptación a cambios.

**No es Multiplataforma, Depende de Windows**

La dependencia de una plataforma específica, como Windows, limita la portabilidad y el alcance de la aplicación. Esto podría dificultar la adopción en entornos no Windows y restringir las opciones de implementación. Microsoft promueve la creación de aplicaciones multiplataforma utilizando tecnologías como .NET Core y .NET 5+.

**No se utilizan pruebas unitarias**

No utilizar pruebas unitarias en el desarrollo de software puede tener varios efectos negativos:

**-Mayor Propensión a Errores:** Sin pruebas unitarias, es más probable que los errores y defectos pasen desapercibidos durante el desarrollo.

**-Dificultad en la Depuración:** Cuando se produce un error en el código, puede ser difícil identificar la causa raíz y corregirlo sin pruebas unitarias. Las pruebas proporcionan un mecanismo para aislar y reproducir errores, lo que facilita la depuración.

**-Mayor Costo a Largo Plazo:** Si los errores no se descubren y corrigen temprano en el proceso de desarrollo, pueden requerir un esfuerzo significativo para su corrección posterior. Esto puede aumentar los costos generales del proyecto y retrasar la entrega.

**-Falta de Documentación Ejecutable:** Las pruebas unitarias actúan como una forma de documentación ejecutable que describe cómo se supone que debe comportarse el código.

**-Mayor Riesgo de Regresiones:** Cuando se realizan cambios en el código sin pruebas unitarias, existe un riesgo significativo de introducir regresiones, es decir, problemas que antes estaban resueltos pueden volver a aparecer.

**Réplica de Modelos**

Esto puede conducir a dificultades en el mantenimiento, actualización y detección de errores.

**Falta de Prevención de Concurrencia**

La falta de prevención de concurrencia puede llevar a problemas de integridad de datos y rendimiento en aplicaciones que manejan múltiples operaciones concurrentes. Microsoft recomienda el uso de técnicas de sincronización, como semáforos o bloqueos, para controlar el acceso concurrente a recursos compartidos y prevenir problemas de concurrencia.

**No se Validan Requerimientos Mínimos de Uso del Sistema**

La falta de validación de requerimientos mínimos, como la velocidad mínima de internet necesaria para ciertas operaciones, puede afectar la experiencia del usuario y la estabilidad del sistema.

**El API no tiene seguridad (jwt ,api key o basic):** Cualquiera con la URL puede ver las rutas de consumo de los datos y obtenerlos.

**Problemas con la Cultura**

Los problemas con la cultura pueden resultar en errores y comportamientos inesperados en aplicaciones que manejan diferentes idiomas y formatos. Microsoft proporciona guías para manejar la internacionalización y la localización de aplicaciones.

**Uso de Try-Catch en conversiones de datos**

No usar bloques try-catch en el manejo de excepciones puede traer varios problemas como por ejemplo :errores no controlados, dificultades de Depuración, comportamientos inesperados.

**Recomendaciones o cosas que se podrían implementar en algún momento:**

**Blazor Server**

Blazor Server es una opción poderosa y beneficiosa para desarrollar aplicaciones web en .NET. Aquí hay varias razones por las cuales podría ser una buena elección:

**-Interactividad en Tiempo Real:** Blazor Server permite crear aplicaciones web interactivas en tiempo real utilizando la tecnología SignalR.

**-Seguridad y Mantenimiento Simplificado:** Al ejecutar la lógica del lado del servidor, Blazor Server ofrece un mayor control sobre la seguridad y protege la lógica interna de tu aplicación, ya que el código del cliente no es accesible para los usuarios.

**-Menor Consumo de Ancho de Banda:** Blazor Server transfiere solo actualizaciones de interfaz de usuario y eventos, en lugar de enviar toda la página en cada interacción, lo que reduce el consumo de ancho de banda.

**-Eficiencia en Recursos:** Las aplicaciones Blazor Server tienen un menor consumo de recursos en comparación con las aplicaciones Blazor WebAssembly, lo que las hace adecuadas para escenarios donde se debe optimizar el uso de recursos del servidor.

**Arquitectura de Microservicios**

Una arquitectura de microservicios ofrece una serie de ventajas y beneficios que la hacen atractiva para muchas organizaciones. Aquí están algunas razones por las cuales una arquitectura de microservicios puede ser beneficiosa:

**-Escalabilidad Eficiente:** Los microservicios permiten escalar cada servicio de manera independiente según su demanda.

**-Desarrollo Ágil:** Los equipos pequeños y autónomos pueden trabajar en cada microservicio de manera independiente. Esto acelera el desarrollo, permite la adopción de metodologías ágiles y facilita la entrega continua.

**-Facilita la Innovación:** Al dividir una aplicación en componentes más pequeños y especializados, es más fácil introducir nuevas características, mejoras y cambios sin afectar a todo el sistema.

**-Fácil Mantenimiento:** Los microservicios facilitan la gestión y el mantenimiento del software. Cada servicio puede actualizarse, corregirse y escalarse sin afectar al resto del sistema.

**-Tecnologías y Lenguajes Variados:** Cada microservicio puede estar implementado con tecnologías y lenguajes diferentes si es necesario.

**-Mayor Flexibilidad:** Los equipos pueden optar por utilizar diferentes bases de datos, frameworks o herramientas según las necesidades de cada servicio.

**-Resistencia a Fallos:** Si un microservicio falla, no afectará a todo el sistema. Además, el resto de los servicios pueden seguir funcionando normalmente.

**-Despliegue Independiente:** Los microservicios pueden desplegarse y actualizarse de manera independiente.

**-Escalabilidad Horizontal:** Los microservicios se pueden distribuir en diferentes servidores, lo que facilita la escalabilidad horizontal para manejar cargas de trabajo más grandes.

**-Facilita la Adopción de la Nube:** Las arquitecturas de microservicios son muy compatibles con la infraestructura en la nube, lo que permite aprovechar características como la elasticidad y la administración de recursos bajo demanda.

**-División de Responsabilidades:** Cada microservicio puede ser propiedad de un equipo específico, lo que ayuda a definir y delimitar responsabilidades.

**KafKa o RabbitMQ en caso de necesitar comunicar Microservicios**

Ambos son mecanismos para comunicar microservicios o servicios que se ejecutan en servidores diferentes.

**Pipelines de Azure Devops**

Los pipelines de Azure DevOps tienen una importancia significativa en el ciclo de vida de desarrollo de software. Estas son algunas de las razones por las cuales los pipelines de Azure DevOps son tan importantes:

**-Automatización del flujo de trabajo:** Los pipelines permiten automatizar la construcción, prueba, despliegue y entrega de aplicaciones de software. Esto reduce la intervención manual y los posibles errores humanos, lo que a su vez mejora la calidad y consistencia del proceso.

**-Entrega Continua:** Los pipelines permiten implementar la entrega continua, lo que significa que los cambios en el código pueden ser probados y desplegados automáticamente en el entorno de producción de manera rápida y segura. Esto agiliza el proceso de lanzamiento de nuevas características y correcciones de errores.

**-Feedback Rápido:** Los pipelines facilitan la ejecución automática de pruebas unitarias, de integración y de aceptación en cada cambio de código.

**Gestión de Versiones:** Los pipelines pueden integrarse con sistemas de control de versiones, lo que permite una gestión más eficiente y rastreable de las versiones del código fuente y las implementaciones.

**Flexibilidad en la Configuración:** Los pipelines de Azure DevOps ofrecen flexibilidad en la configuración, lo que permite adaptar los flujos de trabajo a las necesidades específicas de cada proyecto.

**Despliegue en Múltiples Entornos:** Los pipelines facilitan el despliegue en diferentes entornos (desarrollo, prueba, producción, etc.), lo que permite una gestión más sencilla de las implementaciones en diferentes etapas del ciclo de vida.

**Monitoreo y Trazabilidad:** Los pipelines pueden integrarse con herramientas de monitoreo y trazabilidad, lo que permite seguir el progreso de las implementaciones y detectar problemas en tiempo real.

**Escalabilidad:** Los pipelines de Azure DevOps son escalables y pueden manejar proyectos pequeños y grandes.

**Seguridad:** Los pipelines permiten la implementación de prácticas de seguridad, como la revisión de código, pruebas automatizadas y análisis estático, lo que ayuda a mitigar riesgos de seguridad en el código.

**Eficiencia y Ahorro de Tiempo:** La automatización de tareas repetitivas y la implementación continua ahorran tiempo y recursos en el proceso de desarrollo y despliegue de software.

**OData:**

Usar OData en un API puede ofrecer varias ventajas significativas para la exposición y manipulación de datos a través de servicios web. OData es un protocolo y un estándar de la industria que permite crear y consumir APIs RESTful de manera más eficiente y flexible. Aquí hay algunas razones importantes para considerar el uso de OData en un API :

**-Flexibilidad en Consultas y Filtros:** OData permite a los clientes realizar consultas complejas y personalizadas en los datos utilizando una sintaxis uniforme de consulta. Los clientes pueden filtrar, ordenar, paginar y realizar operaciones de agregación en los datos de manera eficiente y estandarizada.

**-Reducción del Tráfico de Red:** Gracias a su capacidad de filtrado y paginación, OData puede ayudar a reducir la cantidad de datos transferidos entre el cliente y el servidor. Esto mejora el rendimiento y reduce la carga en la red, especialmente en escenarios donde solo se necesitan subconjuntos de datos.

**-Reutilización de Lógica Empresarial:** OData permite encapsular la lógica empresarial en la capa de servidor, lo que significa que los clientes pueden acceder a funcionalidades específicas sin necesidad de saber cómo se implementan internamente.

**-Abstracción de la Fuente de Datos:** OData permite exponer datos de diferentes fuentes, como bases de datos, servicios web, archivos, etc., a través de una única API uniforme. Esto permite a los clientes acceder a múltiples fuentes de datos utilizando la misma sintaxis de consulta.

**-Interoperabilidad:** OData es un estándar ampliamente aceptado y compatible con diferentes plataformas y lenguajes de programación. Esto facilita la interoperabilidad entre sistemas desarrollados en diferentes tecnologías.

**-Versionado Simplificado:** OData proporciona mecanismos incorporados para el versionado de la API, lo que permite a los desarrolladores agregar nuevas características o realizar cambios en la estructura de datos sin afectar a los clientes existentes.

**-Documentación Automática:** Muchos frameworks y herramientas de desarrollo ofrecen generación automática de documentación de API basada en OData. Esto facilita la comprensión y adopción de la API por parte de los desarrolladores y equipos.

**-Seguridad y Control de Acceso:** OData admite controles de seguridad y autenticación integrados, lo que permite controlar el acceso a los datos y proteger la API de accesos no autorizados.

**Anexos**

<https://www.youtube.com/watch?v=2X50sKeBAcQ>

<https://www.youtube.com/watch?v=QxqkmOC-eQY>

<https://www.youtube.com/watch?v=xbg3hp57xx8&t=16s>

<https://www.youtube.com/watch?v=7pzBwuMjpP0>

<https://www.youtube.com/watch?v=WCC5iF5Gg-A>