ARQUITECTURA DE SOFTWARE

Factoría de Software

Plataforma ÁUREA

Versión: 1.1

Hoja de Control

Unidad Responsable:	VIEM		
Proyecto:	Plataforma ÁUREA		
Entregable:	Arquitectura de Software		
Autor:	Documentación Factoría de Software		
Versión/Edición:	1.1	**Fecha de Versión:**	18/09/2024
Aprobado por:		**Fecha de Aprobación:**	19/09/2024
		No total de páginas:	42

Modificaciones/Actualizaciones

Versión:	**Fecha del cambio:**	**Descripción resumida de Modificación/Actualización:**
1.0	03/09/2024	Primera versión emitida
1.1	18/09/2024	Se amplían algunos conceptos.

Tabla de contenido

Introducción

La Arquitectura AUREA es un marco conceptual y práctico diseñado para guiar el desarrollo eficiente y sostenible de sistemas de información en entornos web. Esta arquitectura está respaldada por la experiencia de más de 10 años de desarrollos in-house adelantados por la Universidad Nacional Abierta y a Distancia (UNAD), lo que llevó a la conformación de lo que hoy se conoce como el grupo de la Factoría de Software. A lo largo de este tiempo, se han implementado prácticas que garantizan un rendimiento óptimo y una gestión eficaz de los recursos.

El presente documento detalla las consideraciones clave y las mejores prácticas recomendadas para implementar la Arquitectura AUREA. A lo largo de los siguientes capítulos, se explorarán conceptos generales, herramientas recomendadas, estrategias de almacenamiento de información, diseño modular y gestión de datos desde la perspectiva del usuario.

1. Conceptos Generales

La base de la Arquitectura AUREA se fundamenta en la estructuración del sistema como un monolito modular. Este enfoque combina lo mejor de dos mundos: la simplicidad y eficiencia de un monolito tradicional con la flexibilidad y escalabilidad típicas de arquitecturas más distribuidas (Microservicios).

Un monolito en términos de desarrollo de software es una arquitectura en la cual todas las funcionalidades del sistema se encuentran integradas en una única aplicación. Esto permite que todas las partes del sistema interactúen directamente entre sí, sin depender de comunicaciones complejas entre servicios separados. El principal beneficio de esta aproximación es la simplicidad en el despliegue y la gestión: todo el sistema se ejecuta en una sola instancia y bajo un único proceso, lo que facilita su control y mantenimiento.

Sin embargo, el concepto de monolito modular expande esta idea al introducir una estructura interna que divide el sistema en componentes o módulos independientes. En lugar de tener una única base de código en la que todas las funcionalidades están fuertemente acopladas, en un monolito modular cada módulo está diseñado para funcionar de manera independiente, con responsabilidades claramente definidas y una interfaz bien establecida para interactuar con otros módulos.

- Autonomía de los Módulos: Cada módulo dentro del monolito puede ser desarrollado, testeado y mantenido por separado, aunque todos formen parte de la misma aplicación. Esta separación modular permite escalar el desarrollo en equipos grandes y evitar el "caos monolítico" que puede surgir en sistemas no organizados adecuadamente. Los módulos son autónomos en términos de lógica de negocio y pueden operar sin interferir con otros módulos, excepto por el componente transversal y las librerías compartidas.

- Componente Transversal y Librerías Compartidas: En un monolito modular, existe un componente transversal que actúa como el núcleo que provee servicios comunes a todos los módulos (autenticación, manejo de errores, etc.). Además, se utilizan librerías compartidas que ofrecen funcionalidades reutilizables y estandarizadas, las cuales pueden ser utilizadas por

varios módulos. Esto permite reducir la duplicación de código y facilita la consistencia en todo el sistema.

- Facilita el Mantenimiento y la Escalabilidad: Aunque el sistema sigue siendo una aplicación monolítica en términos de despliegue, la separación modular facilita el mantenimiento y la escalabilidad del sistema, ya que cada módulo puede ser actualizado o extendido sin afectar al resto de la aplicación. Además, permite la reutilización de módulos en diferentes partes del sistema o incluso en otros proyectos, brindando flexibilidad a largo plazo.

Un monolito modular es particularmente útil cuando se espera que un sistema crezca en complejidad pero se busca evitar los costos y desafíos de gestionar múltiples servicios distribuidos, como es el caso de las arquitecturas basadas en microservicios. Proporciona una solución intermedia donde el sistema sigue siendo fácil de manejar y desplegar como una unidad completa, pero está organizado de tal manera que cada parte tiene una responsabilidad clara y puede evolucionar de manera independiente.

1.1. Estructura Modular del Sistema

Con este tipo de estructura buscamos asegurar que cada componente del sistema sea autónomo y que cualquier cambio o extensión en un módulo no afecte la estabilidad o el funcionamiento de otros módulos.

1.1.1 Componentes Autónomos.

Cada componente del sistema debe ser independiente y capaz de funcionar por sí mismo, permitiendo desarrollar, probar y desplegar funcionalidades de manera aislada sin afectar al resto del sistema, teniendo en cuenta los siguientes parámetros:

a) Identificación de Módulos: Cada módulo se identifica con un número único de módulo, lo que permite su clasificación y gestión dentro del sistema. La forma de gestionar y asignar estos números de módulo será explicada más adelante en detalle.

b) Estructura Modular basada en MVC: Cada módulo sigue el patrón de diseño Modelo-Vista-Controlador (MVC), aunque con una simplificación que mejora la eficiencia en la gestión de archivos. En lugar de tener archivos separados para la Vista, el Modelo y el Controlador, el controlador está integrado dentro de la página de vista, lo que reduce la cantidad de archivos que deben ser gestionados y facilita el mantenimiento. Esta integración permite que cada módulo sea más compacto sin sacrificar la separación de responsabilidades.

c) Páginas de Vista y Modelo: Cada módulo puede estar compuesto por una página de Vista y una página de Modelo. La página de Vista maneja la interfaz de usuario y las interacciones, mientras que la página de Modelo gestiona los datos y la lógica de negocio, siguiendo las mejores prácticas del patrón MVC.

d) Gestión del Idioma: Se recomienda que cada módulo tenga archivos separados dedicados a la gestión del idioma. Esto permite que si en el futuro es necesario cambiar el idioma de la aplicación, se pueda realizar actualizando solo estos archivos. Estos archivos, al estilo de los archivos de recursos, contienen todas las etiquetas y cadenas de texto que son mostradas en la interfaz, facilitando la internacionalización y localización del sistema sin modificar el código central.

1.1.2 Componentes Transversales:

En un monolito modular, el componente transversal desempeña un papel esencial al proporcionar servicios y funcionalidades comunes que son utilizados por todos los módulos del sistema. Este componente no solo facilita la interoperabilidad entre módulos, sino que también establece un entorno de trabajo básico sin el cual los demás módulos no pueden ser gestionados de manera eficiente, y debe incluir:

a) Modelo General de Auditoría de Datos: El componente transversal debe incluir un modelo general de auditoría de datos, que permita rastrear y registrar las acciones realizadas por los usuarios en el sistema. Este modelo garantiza que se mantenga un historial de cambios, accesos y operaciones, cumpliendo con requisitos de seguridad, cumplimiento normativo y trazabilidad de la información.

b) Modelo de Autenticación: Es esencial que este componente incluya un modelo robusto de autenticación para gestionar la verificación de usuarios. Esto asegura que solo los usuarios autorizados puedan acceder a las funcionalidades y datos del sistema, proporcionando una capa fundamental de seguridad.

c) Modelo de Gestión de Variables de Entorno: Para facilitar la configuración y adaptación del sistema a diferentes usuarios, el componente transversal debe incorporar un modelo de gestión de variables de entorno. Estas variables permiten a los usuarios controlar configuraciones predeterminadas del sistema conforme lo deseen.

d) Modelo de Gestión de Menús y Permisos: El componente transversal debe incluir también un modelo de gestión de menús que permita definir y estructurar la navegación del sistema de forma dinámica. Asimismo, un modelo de gestión de permisos es crucial para controlar qué acciones (consultar, guardar, modificar, eliminar, imprimir, exportar) están disponibles para cada usuario o rol dentro del sistema.

e) Modelo de Control de Usuarios: Dependiendo del nivel de seguridad esperado, es recomendable incluir un modelo de control de usuarios que permita la gestión avanzada de usuarios, roles, y políticas de acceso. Esto asegura que se puedan definir jerarquías y restricciones en función de las necesidades específicas del sistema.

f) Gestión del Componente Visual: Desde una perspectiva visual, el componente transversal debe gestionar de manera preferente una librería separada que a su vez gestione las referencias al conjunto de archivos visuales (CSS, JavaScript, íconos, plantillas de interfaz) de forma que

sean fácilmente referenciables desde los diferentes módulos sin interferir con la capa de negocio. Esto asegura una separación clara entre la lógica de negocio y la presentación visual.

Este componente transversal, al proporcionar estos modelos y servicios, garantiza que el sistema funcione de manera consistente y segura, facilitando la escalabilidad y el mantenimiento de la arquitectura en el largo plazo.

1.1.3 Librerías Compartidas.

Además de los componentes específicos, se utilizan librerías compartidas de uso común entre múltiples componentes, permitiendo mantener la consistencia en todo el sistema. Estas librerías se clasifican en tres tipos principales, según su función y ámbito de aplicación:

- a) Librerías de Módulos Compartidos: Estas librerías corresponden a las mismas librerías de negocio utilizadas por los módulos específicos, pero que pueden ser referenciadas por más de una aplicación. Por ejemplo, un módulo de gestión de paz y salvos podría ser referenciado por otros componentes, por lo que su librería se marca como compartida. Recordando que cada librería se identifica con su número de módulo, facilitando su gestión.

- b) Librerías de Componentes: Estas librerías contienen las funciones de negocio específicas de un componente particular. Son utilizadas por varios módulos del mismo componente y generalmente incluyen funciones relacionadas con el cálculo o procesamiento de datos. Se identifican mediante el número o nombre del componente, lo que asegura que puedan ser fácilmente referenciadas por los módulos que lo necesiten.

- c) Librerías de Propósito Común: Estas librerías contienen funciones generales que pueden ser utilizadas en todo el sistema. Se recomienda contar, al menos, con las siguientes librerías básicas:

1. Librería de Gestión de Base de Datos: Proporciona funciones estándar para interactuar de manera eficiente con la base de datos independiente al motor de base de datos que estemos usando.

2. Librería de Funciones Generales: Incluye funciones para validación de cadenas, gestión de fechas, y formatos, esto con el fin de evitar dependencias a las configuración regional de los usuarios.

3. Librería de Gestión de Datos Comunes: Centraliza el manejo de datos compartidos por varios módulos, como catálogos o configuraciones.

4. Librería Auxiliar de Componentes HTML: Facilita la generación y manejo de componentes de interfaz de usuario en HTML.

5. Librería para Control e Inicio de Sesión: Gestiona el inicio de sesión y la autenticación de usuarios.

6. Librería Auxiliar para Reportes: Ofrece soporte para la creación de reportes en formatos como Microsoft Excel o PDF, asegurando que el sistema pueda exportar datos de manera flexible.

Estas librerías permiten mantener un estándar en el desarrollo de nuevas funcionalidades. Aunque se recomienda este conjunto mínimo de librerías, no se limita la posibilidad de crear nuevas librerías según los requisitos del proyecto.

1.2. Organización Interna de los Componentes

Cada componente se divide internamente en múltiples módulos, cada uno responsable de una funcionalidad o conjunto de funcionalidades específicas.

- Permisos y Seguridad: Cada módulo se asocia con un conjunto de permisos que controlan el acceso y las operaciones permitidas. Los permisos comunes incluyen:

  - Consultar: Permite la visualización de datos.

  - Guardar: Autoriza la creación de nuevos registros.

  - Modificar: Habilita la edición de registros existentes.

  - Eliminar: Permite la eliminación de registros.

  - Imprimir: Autoriza la generación de versiones impresas de los datos.

  - Exportar: Permite la extracción de datos en formatos externos.

Esta estructura jerárquica y controlada garantiza una gestión segura y eficiente de las funcionalidades del sistema, facilitando también la asignación y gestión de roles de usuario.

2. Herramientas a Considerar

Si bien es cierto que las tecnologías avanzan rápidamente ofreciendo soluciones cada vez más sofisticadas, en países en vías de desarrollo como Colombia, no siempre es posible financiar estas herramientas. Los costos asociados al uso de ciertas tecnologías pueden ser, si no insostenibles, al menos excesivos para nuestras organizaciones. Esto lleva a muchas instituciones a buscar soluciones más accesibles y sostenibles, como el uso de plataformas de código abierto.

En el caso de Colombia, la Ley 1712 de 2014 sobre Transparencia y el Derecho de Acceso a la Información Pública fomenta la adopción de datos abiertos, y políticas como la de Gobierno Digital recomiendan el uso de tecnologías abiertas por sus costos de mantenimiento y operación más bajos. Además, el Plan Nacional de Infraestructura de Datos establece lineamientos que promueven la interoperabilidad y la apertura de datos, apoyando el uso de plataformas accesibles. Esto responde a la necesidad de reducir el impacto financiero de los sistemas propietarios en entidades públicas.

2.1 Retos a Considerar en la Implementación de Sistemas de Información

Al implementar un sistema de información, es fundamental tener en cuenta varios desafíos que pueden impactar la durabilidad y sostenibilidad del proyecto a lo largo del tiempo:

- Durabilidad de los Proyectos: Las tecnologías avanzan rápidamente, y con ellas, los sistemas de información pueden volverse obsoletos en un periodo de tiempo relativamente corto. Esto representa un reto importante para las instituciones que desarrollan y mantienen sus sistemas in-house, ya que la obsolescencia tecnológica requiere actualizaciones constantes para mantener la competitividad y funcionalidad del sistema.

- Rotación del Personal: Cada vez que hay un cambio de personal en el equipo de desarrollo o mantenimiento, la institución debe pasar por un nuevo ciclo de entrenamiento y evaluación. Durante este proceso, es crucial garantizar que los nuevos colaboradores adquieran las capacidades necesarias para gestionar los sistemas existentes. Esto puede llevar tiempo y recursos, retrasando las mejoras o el mantenimiento del sistema hasta que el equipo esté completamente adaptado.

- Costos de Operación: Los costos asociados con la operación de un sistema de información pueden escalar significativamente, sobre todo si se utilizan tecnologías propietarias. Además del costo de las licencias, se deben considerar los gastos en infraestructura, mantenimiento, y capacitación continua del personal. Para muchas organizaciones, estos costos pueden volverse excesivos, lo que refuerza la necesidad de buscar soluciones más sostenibles, como las plataformas de código abierto.

- Obsolescencia de Versiones de las Herramientas de Desarrollo: Un reto adicional que debe considerarse al implementar sistemas de información es la obsolescencia de versiones en las herramientas de desarrollo. A medida que las plataformas y entornos de desarrollo evolucionan, las versiones antiguas de estas herramientas pueden quedar obsoletas, lo que puede impactar significativamente la sostenibilidad y el mantenimiento del sistema.

Ejemplos de Obsolescencia:

a) Un claro ejemplo de esta problemática es la desaparición de Adobe Flash, una tecnología ampliamente utilizada para desarrollar aplicaciones web y multimedia que, tras años de uso, fue oficialmente descontinuada en 2020. Los sistemas desarrollados con Flash quedaron inutilizables a menos que fueran migrados a otras plataformas, lo que implicó un esfuerzo considerable para las organizaciones que todavía dependían de esta tecnología.

b) Cambios Frecuentes en Visual Studio: Otro ejemplo común es el caso de Visual Studio, un entorno de desarrollo que constantemente introduce nuevas versiones y actualizaciones. Estos cambios frecuentes, aunque mejoran las funcionalidades y seguridad, pueden generar incompatibilidades con proyectos desarrollados en versiones anteriores. Esto obliga a las organizaciones a mantenerse al día con las actualizaciones, lo que puede incrementar los costos y el esfuerzo de mantenimiento.

Es fundamental que las organizaciones que implementan sistemas de información consideren el uso de tecnologías estables con un ciclo de vida prolongado y una comunidad activa que garantice el soporte a largo plazo Tanto para el entorno de desarrollo como para las bases de datos, buscando siempre hacer uso de herramientas que han demostrado ser robustas, escalables y con comunidades activas que ofrecen soporte continuo.

Sin embargo, la actualización de estas tecnologías es un aspecto obligatorio para asegurar la longevidad y seguridad del sistema. Las razones para mantener los sistemas actualizados incluyen:

1. Retos de Seguridad: Las vulnerabilidades en versiones antiguas de las herramientas de desarrollo y software pueden ser explotadas por atacantes, poniendo en riesgo la integridad del sistema y los datos.

2. Crecimiento de la Infraestructura: A medida que la infraestructura de la organización crece, puede ser necesario actualizar las herramientas para soportar el volumen de datos, usuarios y transacciones.

3. Cambios en las Tipologías Implementadas: Los avances en las mejores prácticas de desarrollo y la aparición de nuevas arquitecturas o patrones de diseño pueden requerir actualizaciones en las herramientas para mantenerse alineado con los estándares actuales.

4. Migración de Equipos de Cómputo: Los cambios en el hardware y la incorporación de nuevas tecnologías pueden requerir actualizaciones para garantizar que las herramientas de desarrollo sigan siendo compatibles con la infraestructura actualizada.

5. Actualización de Sistemas Operativos: Las nuevas versiones de sistemas operativos pueden introducir cambios que hacen que versiones antiguas de herramientas de desarrollo se vuelvan incompatibles.

6. Componentes de Terceros: El uso de componentes de terceros añade un nivel de complejidad adicional, ya que estos componentes pueden ser descontinuados o pasar de ser gratuitos a

tener un costo, lo que implica la necesidad de reemplazarlos o actualizarlos para evitar interrupciones en el funcionamiento del sistema.

Es importante considerar que aunque la actualización de una plataforma de desarrollo puede implicar un esfuerzo significativo, es un proceso indispensable para garantizar la seguridad, estabilidad y escalabilidad de cualquier sistema de información, particularmente en un entorno tecnológico en constante evolución.

2.2. Uso de Código Puro

La decisión de utilizar código puro en el desarrollo de sistemas de información debe alinearse con las prioridades a largo plazo de la organización. Este enfoque permite un mayor control sobre el desarrollo y optimización del software, pero también conlleva desafíos que deben gestionarse cuidadosamente.

- Priorización Organizacional

Es fundamental que cada organización defina claramente sus prioridades antes de optar por este enfoque. Por ejemplo:

- Organizaciones con Bajo Volumen de Procesamiento: Para una organización con pocos usuarios o un volumen de procesamiento bajo, la prioridad puede ser reducir los costos al máximo. Esto puede implicar sacrificar un poco de escalabilidad o estar dispuestos a pagar por licenciamiento y actualizaciones para usar herramientas que no exijan una personalización profunda.

- Caso de la UNAD: En el caso de la Universidad Nacional Abierta y a Distancia (UNAD), la institución ha priorizado la velocidad de respuesta de las aplicaciones y el blindaje en seguridad. Dado que la UNAD es una institución pública, está constantemente expuesta a intentos de intrusión de usuarios malintencionados. Además, debido a las perspectivas de crecimiento y actualización constante, el uso de código puro permite personalizar y optimizar cada componente del sistema para estas necesidades.

- Ventajas del Uso de Código Puro

1. Rendimiento Optimizado: Al desarrollar en código puro, se elimina la sobrecarga introducida por frameworks o bibliotecas externas. Esto permite un rendimiento más ágil y una mejor gestión de los recursos computacionales, lo cual es clave en entornos donde la velocidad de respuesta es una prioridad, como en el caso de la UNAD.

2. Control Total: El uso de código puro proporciona un control absoluto sobre todos los aspectos del sistema, desde la lógica de negocio hasta la gestión de datos y la seguridad. Esto permite que cada módulo sea optimizado y ajustado de acuerdo a las necesidades específicas de la organización.

3. Alta Personalización: Al no depender de estructuras rígidas impuestas por frameworks, el código puro permite una personalización extrema de las funcionalidades, asegurando que cada módulo se ajuste a las necesidades exactas del sistema y pueda evolucionar junto con la organización.

- Retos del Uso de Código Puro

1. Mantener una Estandarización: Dado que no se utilizan frameworks estandarizados, uno de los principales desafíos es asegurar que todo el equipo de desarrollo siga una normativa interna coherente para mantener el código limpio y organizado. Esto requiere una constante revisión y actualización de las mejores prácticas, para lo cual al interior de la UNAD se ha establecido un manual de estilo para lenguaje PHP, el cual es de obligatorio cumplimiento, siendo este un documento derivado del presente documento.

2. Rotación de Programadores: La rotación del personal es un desafío significativo. Los nuevos programadores requieren una curva de aprendizaje considerable para adaptarse a la arquitectura personalizada. Este proceso puede ralentizar la incorporación de nuevos desarrolladores, lo que podría afectar la velocidad de implementación de nuevas funcionalidades, para mitigar este riesgo se ha implementado un curso de entrenamiento el cual es de obligatorio cumplimiento para nuestros colaboradores.

3. Mantenimiento de la Documentación: Al no tener una estructura impuesta por herramientas de terceros, la responsabilidad de mantener una documentación detallada y actualizada recae completamente en el equipo de desarrollo. La falta de una buena documentación puede dificultar la gestión a largo plazo del sistema, en tal sentido el equipo de desarrollo cuenta con un grupo de apoyo de documentación.

4. Incorporación de Nuevas Funcionalidades: Aunque el código puro permite personalizar profundamente el sistema, la incorporación de nuevas funcionalidades puede requerir desarrollos desde cero, lo cual puede ser más lento en comparación con la reutilización de componentes preexistentes en frameworks, sin embargo su impacto de largo plazo tiende a ser bajo pues una vez definida, construida y documentada la funcionalidad esta es reutilizada con facilidad.

5. Adecuada Gestión de los Costos de Almacenamiento y Procesamiento de Datos: La eficiencia en el uso del código puro también implica tener una estrategia sólida para la gestión del almacenamiento y los costos de procesamiento. La estructura del sistema debe garantizar que se optimicen los recursos sin sacrificar el rendimiento o la escalabilidad. En tal sentido el cumplimiento del manual de estilo garantiza que el sistema pueda crecer de manera organizada.

2.3. Desarrollo de Librerías Propias

El desarrollo de librerías propias es una práctica recomendada en la Arquitectura AUREA para estandarizar funcionalidades comunes y promover la reutilización de código dentro del proyecto. Este enfoque permite optimizar el trabajo en equipo, reducir la duplicación de código y garantizar que los módulos del sistema sigan principios coherentes.

- Beneficios de Crear Librerías Propias

1. Estandarización: Las librerías propias ayudan a establecer convenciones y prácticas uniformes que mejoran la coherencia y calidad del código en todo el sistema. Esto asegura que todas las funcionalidades sigan un patrón consistente y bien documentado, facilitando el mantenimiento.

2. Facilita el Aprendizaje: Aunque el desarrollo de librerías internas puede implicar una curva de aprendizaje pronunciada para los nuevos programadores, una vez familiarizados con estas librerías y los estándares internos, los desarrolladores pueden contribuir de manera eficaz a cualquier componente o módulo del sistema. Además, estas librerías proporcionan un marco que otorga autonomía a los programadores, permitiéndoles desarrollar sin comprometer los estándares generales.

3. Mantenibilidad: Al centralizar funcionalidades comunes en librerías compartidas, las actualizaciones y mejoras pueden realizarse de manera más eficiente. Cualquier cambio o corrección en una librería afecta a todos los módulos que dependen de ella, evitando inconsistencias y errores.

- Consideraciones para el Desarrollo de Librerías

En el contexto de la Arquitectura AUREA, es importante definir el alcance de cada librería según su función y nivel de uso dentro del sistema. Sin entrar en detalles de implementación de lenguaje, describimos a continuación algunas consideraciones clave:

- a) Alcance de la Librería:

1. Librerías Generales: Son aquellas que se utilizan frecuentemente en todos los componentes del sistema. Ejemplos comunes incluyen:

- Librerías de inicialización de sesión: Para gestionar la autenticación y control de usuarios.

- Librerías de gestión de bases de datos: Encargadas de las interacciones con la base de datos, como la ejecución de consultas.

- Librerías de funciones comunes: Para operaciones de uso general, como la validación de datos, gestión de fechas o manipulación de cadenas de texto.

- Librerías de notificaciones: Para enviar alertas o mensajes a los usuarios, principalmente por correo electrónico.

Estos tipos de librerías no se fuerza a una estructura de nombres, únicamente se solicita que el nombre de la librería seria descriptivo, ejemplo: libnotifica para la librería de notificaciones.

2. Librerías Compartidas entre Componentes: Estas librerías tienen un alcance intermedio y son utilizadas por múltiples componentes dentro del sistema. Un ejemplo típico sería una librería para la gestión de estudiantes, que podría incluir funciones comunes como el procesamiento de matrículas o la actualización de información académica, y ser referenciada por varios módulos relacionados con el sistema académico.

Generalmente estas librerías heredan el nombre del número de módulo que las contiene, ejemplo si el modulo de estudiantes es el 2202 la librería de estudiantes deberá llamarse lib2202

3. Librerías de Terceros: Aunque la Arquitectura AUREA no promueve el uso excesivo de librerías de terceros, en algunos casos es recomendable utilizarlas para resolver problemas específicos. El uso de librerías de terceros debe ser limitado y su integración debe garantizar que no comprometan el rendimiento o la escalabilidad del sistema. Son altamente recomendadas en tareas especializadas que implicarían un alto costo de desarrollo interno, como la generación de informes en PDF o la integración con servicios externos.

- Estrategia para Librerías de Terceros

Es importante minimizar la dependencia de librerías de terceros, ya que estas pueden ser descontinuadas o pasar de ser de uso libre a requerir un pago por licencia. Para mitigar este riesgo, se recomienda utilizar librerías que cuenten con una comunidad activa de soporte y evitar aquellas cuya actualización o mantenimiento dependa exclusivamente de un proveedor. Además, se debe considerar la posibilidad de desarrollar internamente algunas de estas funcionalidades en el futuro si las condiciones cambian.

- Uso de Estándares de Nombres de Clases y Funciones

En el desarrollo de librerías propias, es esencial seguir una guía clara y consistente para el uso de nombres de funciones y parámetros, así como políticas claras de retorno de datos. Esto no solo facilita el mantenimiento del código, sino que también optimiza las labores de soporte técnico y mejora la claridad en la comunicación dentro del equipo de desarrollo.

- Estándares de Nombres

1. Guía de Nombres de Funciones y Parámetros: En la Arquitectura AUREA, se recomienda que todas las funciones sigan una convención de nombres descriptivos que indiquen claramente la acción que realizan,. Además, es útil establecer una política donde los parámetros de entrada también tengan nombres intuitivos que describan su propósito. Un aspecto clave es que la mayoría de las funciones deben:

- Recibir un parámetro de depuración: Este parámetro permite registrar la actividad de la función durante su ejecución, lo que es útil para detectar y corregir errores.

- Devolver un mensaje de error y el registro del proceso de depuración: Esto facilita ampliamente las labores de soporte y permite una mayor transparencia sobre el comportamiento interno de las funciones.

2. Incorporación del Número de Módulo en el Nombre de las Funciones: Las funciones que estén vinculadas a un módulo específico del sistema deben heredar el número de módulo en su nombre usando de prefijo la letra f y luego del número de módulo una barra al piso. Esto hace que sea mucho más fácil identificar a qué parte del sistema pertenece cada función y simplifica la gestión del código a largo plazo. Por ejemplo, si una función pertenece al módulo de estudiantes con el número de módulo 2202, su nombre podría ser algo como `f2202_PuedeEditarPlanEstudios(idRegistro, bDebug)`. Tengase en cuenta que los nombres de las funciones usan la primera letra en mayúscula para poder hacer una lectura más sencilla de la misma, práctica que se recomienda, pero que se desarrolla en el manual de estilo correspondiente a cada lenguaje de programación.

- Políticas de Retorno de Datos

3. Control de Fallas y Gestión de Procesos: A nivel arquitectónico, es crucial considerar que los procesos pueden sufrir fallas de comunicación o saturación. Por ello, es recomendable implementar un control de fallas en las funciones que requieran un alto nivel de procesamiento, en lugar de delegar esta responsabilidad únicamente a la base de datos. Esto permite que el sistema responda de manera más robusta frente a imprevistos.

4. Procesamiento de Datos: Para mejorar el rendimiento y la estabilidad, es altamente recomendable que los datos no se procesen directamente dentro de la función a partir de parámetros pasados, sino que estos datos estén previamente guardados y almacenados antes de la ejecución de procesos. Este enfoque asegura que las funciones trabajen sobre datos ya validados y listos para su manipulación, evitando sobrecargas en la ejecución.

2.4. Consideraciones sobre la Curva de Aprendizaje

El uso de código puro y la creación de librerías propias dentro de la Arquitectura AUREA presentan una curva de aprendizaje pronunciada, especialmente para los nuevos desarrolladores. Sin embargo, existen estrategias y herramientas que pueden facilitar esta adaptación, asegurando que el equipo mantenga la coherencia y calidad en el desarrollo.

- Documentación Exhaustiva

Una documentación detallada y actualizada es fundamental para garantizar que los desarrolladores comprendan y sigan los estándares establecidos en la arquitectura. Para facilitar la incorporación de nuevos programadores y asegurar el correcto mantenimiento del sistema, se deben mantener actualizados los siguientes documentos:

1. Documento de Arquitectura: El presente documento, que describe la estructura modular, las librerías compartidas, los estándares de desarrollo y otras consideraciones clave de la Arquitectura AUREA.

2. Manual de Estilo para el Lenguaje de Programación: Un manual que especifica las convenciones y buenas prácticas para el uso del lenguaje de programación empleado, incluyendo nombres de variables, funciones, y estructuras recomendadas para asegurar la coherencia en el código, así como los aspectos relativos a la seguridad.

3. Manual General del Administrador del Sistema: Este manual se centra en las tareas relacionadas con la administración del sistema, como la gestión de usuarios, la configuración de servidores, la monitorización de la seguridad y tareas frecuentes de mantenimiento.

4. Documento Explicativo de los Procesos Atendidos por Cada Componente: Un documento que detalla cuáles son los procesos que atiende cada componente del sistema, permitiendo a los desarrolladores comprender el propósito y el alcance de cada parte de la aplicación.

Es importante aclarar que, en este capítulo, nos referimos a los documentos generales del sistema, sin entrar en el detalle de la documentación técnica específica de cada proceso de desarrollo.

- Programa de Capacitación

Para reducir la curva de aprendizaje, se ha implementado un curso de entrenamiento específico para la metodología AUREA. Este programa incluye:

- Mentoría por Programadores del Grupo: Los desarrolladores más experimentados actúan como mentores de los nuevos miembros del equipo. Esto acelera el proceso de aprendizaje, brindando soporte continuo para que los nuevos colaboradores adopten y apliquen correctamente las prácticas establecidas.

El programa de capacitación se enfoca en proporcionar una formación completa sobre la arquitectura, las librerías propias, y los estándares de desarrollo utilizados, asegurando que los desarrolladores puedan trabajar con cierto nivel de autonomía en el menor tiempo posible.

- Comunidad y Colaboración

El enfoque de desarrollo en la Arquitectura AUREA se basa en un modelo de comunidad abierta y colaboración continua. Este formato fomenta la participación activa de todos los desarrolladores, permitiendo compartir conocimientos y resolver dificultades en equipo. Las prácticas incluyen:

- Reuniones Semanales de Seguimiento: Estas reuniones tienen el propósito de revisar el progreso, discutir problemas y encontrar soluciones colaborativas. Se establece un entorno abierto donde los desarrolladores pueden compartir experiencias y buenas prácticas, lo que contribuye a una mejora continua en el uso de las herramientas y la aplicación de los estándares establecidos.

Este enfoque asegura que todo el equipo esté alineado con los objetivos del proyecto y facilite la rápida adaptación de nuevos desarrolladores.

3. Guardado de Información

El diseño eficiente y organizado de la base de datos es fundamental para garantizar la integridad, seguridad y rendimiento del sistema. En la Arquitectura AUREA, se han implementado dos herramientas clave para gestionar este control:

1. Diccionario de Datos: Esta herramienta es gestionada por el coordinador del grupo de desarrollo y permite llevar un control preciso de los números de componentes, números de módulos y un registro completo de todas las tablas y los campos que estas contienen. El diccionario de datos es esencial para mantener la organización y la estandarización en el diseño de la base de datos.

2. Script/Módulo de Base de Datos: Este script o módulo, dependiendo del lenguaje de programación, contiene el registro histórico y secuencial de todas las sentencias de la base de datos. De este modo, en caso de requerirse una nueva instalación, el script puede generar automáticamente la estructura completa de la base de datos, garantizando una implementación coherente y sin errores.

Adicionalmente, la Arquitectura AUREA aplica técnicas de bases de datos no relacionales (Big Data), mientras mantiene en la capa de negocio las características de las bases de datos relacionales. Este enfoque híbrido permite aprovechar los beneficios de ambos modelos. Se contemplan las siguientes consideraciones:

1. Bases de Datos como Contenedores de Datos: En AUREA, las bases de datos se utilizan exclusivamente como contenedores de datos, lo que significa que no se permite el uso de procedimientos almacenados ni vistas como procedimientos habituales. Excepciones a esta regla se consideran solo en circunstancias especiales, pero deben evitarse en la medida de lo posible.

2. Estructura de las Tablas: La estructura de las tablas en AUREA está dividida en tres partes:

- Llave única de los datos.
- Llave primaria de la tabla (numérica, autoincremental y gestionada desde el código).
- Atributos adicionales de las tablas.

3. No Uso de Relaciones en Cascada: Para minimizar los costos transaccionales de la base de datos, no se permite el uso de relaciones en cascada. Este enfoque garantiza un mejor control sobre las operaciones de base de datos, mejorando el rendimiento.

4. Usuarios de Base de Datos y Permisos: Los usuarios de la base de datos no están relacionados con los usuarios del sistema, lo que permite el uso de un único usuario de base de datos por servidor. Estos usuarios deben contar con un conjunto amplio de permisos (crear tablas, crear índices, realizar inserciones y eliminaciones) para permitir que el sistema gestione la base de datos de manera autónoma, sin necesidad de intervención manual.

5. Almacenamiento de Archivos: Los archivos que el sistema necesita almacenar se guardan en contenedores externos. En la base de datos principal, solo se mantienen dos apuntadores: uno que señala al número del contenedor y otro que apunta al número del archivo. Cada contenedor almacena tanto el archivo como su información asociada.

6. Categorización de Tablas Temporales: Las tablas que contienen datos temporales, como las de auditoría, inicios de sesión y notificaciones, se gestionan en contenedores por año o por mes, según el volumen de datos. Los contenedores antiguos pueden ser trasladados a una base de datos de históricos o eliminados, lo que permite mantener bajo control el crecimiento de la base de datos y optimizar tanto el rendimiento como las copias de seguridad.

3.1. Nomenclatura de Tablas

- Uso de Prefijos Estandarizados: Adoptar un esquema de nomenclatura consistente donde cada tabla comienza con un prefijo de cuatro letras que identifica el componente al que pertenece, seguido de dos números que indican el número de tabla dentro de ese componente. Ejemplo: `COMP01`, donde `COMP` representa el componente y `01` la primera tabla.

- Beneficios:

  - Facilita la identificación y organización de las tablas.

  - Evita conflictos de nombres y mejora la claridad en consultas y mantenimiento.

  - Ayuda en la gestión y documentación de la estructura de la base de datos.

3.2. Gestión de Claves

- Claves Primarias Separadas de Claves Únicas:

- Claves Primarias: Deben ser numéricas y generadas de manera secuencial o aleatoria sin relación directa con los datos del registro. Esto asegura:

- Integridad Referencial: Facilita la gestión de relaciones entre tablas.

- Seguridad: Evita exponer información sensible a través de identificadores predecibles.

- Eficiencia: Mejora el rendimiento en operaciones de búsqueda y unión de tablas.

- Claves Únicas: Se utilizan para garantizar la unicidad de los registros basándose en campos significativos (ejemplo: número de identificación, correo electrónico). Esto permite:

- Validación de Datos: Previene la duplicación de información crítica.

- Acceso Rápido: Facilita la localización de registros específicos mediante información conocida.

3.3. Almacenamiento de Archivos

- Contenedores Externos: Los archivos asociados (documentos, imágenes, etc.) deben almacenarse en contenedores separados, que pueden ser:

- Discos Duros Externos: Para un acceso rápido y almacenamiento local.

- Bases de Datos Especializadas: Para una gestión más estructurada y segura, especialmente en entornos distribuidos o con requisitos de escalabilidad.

- Consideraciones:

- Rendimiento: La separación permite optimizar el acceso y reduce la carga sobre la base de datos principal.

- Seguridad: Facilita la implementación de políticas de seguridad y respaldo específicas para archivos.

- Escalabilidad: Permite manejar grandes volúmenes de datos sin afectar el rendimiento del sistema central.

3.4. Tipos de Datos Limitados

La Arquitectura AUREA busca ser independiente del motor de base de datos, lo que implica que los tipos de datos a usar en las tablas se deben limitar para asegurar compatibilidad y flexibilidad a largo plazo. Los tipos de datos permitidos se han reducido cuidadosamente para cumplir con los siguientes objetivos:

- Estandarización de Tipos de Datos

1. Consistencia: Limitar y controlar los tipos de datos garantiza que los mismos tipos de información se manejen de manera uniforme a lo largo de todo el sistema, facilitando la validación y el procesamiento de datos.

2. Compatibilidad: Al utilizar un conjunto estándar de tipos de datos, se facilita la interoperabilidad entre distintos componentes del sistema, así como con posibles migraciones a otros motores de bases de datos.

3. Optimización: Evitar conversiones y operaciones innecesarias mejora el rendimiento del sistema, ya que cada dato se almacena y procesa de manera eficiente.

- Estrategias de Uso

- Se ha definido un conjunto limitado de tipos de datos aprobados que deben usarse en todas las tablas del sistema.

- Se proporcionan guías claras sobre cuándo y cómo utilizar cada tipo de dato.

- Se implementan validaciones y restricciones a nivel de base de datos para garantizar el cumplimiento de estas reglas.

- Tipos de Datos Definidos

- Entero: Este tipo se utiliza exclusivamente para las llaves primarias de las tablas y para cualquier otro dato de sistema que sea necesario.

- Texto: Se limita a un máximo de 250 caracteres y es utilizado para almacenar cadenas de texto que no superen este límite.

- Doble (Moneda): Se utiliza para datos numéricos que requieren el uso de decimales, principalmente para valores relacionados con cantidades monetarias.

- Memo: Se emplea para datos que necesitan almacenar texto sin limitación de longitud, como descripciones o notas extensas.

- Consideraciones Especiales

1. Fechas: En lugar de utilizar un tipo de dato de fecha específico, las fechas se guardan como Entero en formato AAAAMMDD (donde AAAA es el año, MM es el mes y DD es el día). Las horas se almacenan como Entero en un rango de 0 a 23, y los minutos en un rango de 0 a 59.

2. Booleanos: Los valores booleanos se almacenan como Entero, con los valores 0 (falso) y 1 (verdadero). Si es necesario representar un valor NULL, se utiliza el valor -1.

3. Datos Binarios: No se permite almacenar datos binarios dentro de las tablas de datos. Los archivos y datos binarios se almacenan en contenedores de archivos externos.

4. Tamaño de Enteros y Decimales: El tamaño de los campos Entero o Decimal se puede definir según las necesidades de cada caso, sin que exista una restricción específica para ello.

3.5. Gestión del Volumen de Datos

Gestión del Volumen de Datos

El manejo adecuado del volumen de datos es esencial para garantizar que el sistema mantenga su rendimiento y escalabilidad a lo largo del tiempo. En la Arquitectura AUREA, se implementan varias estrategias para controlar el tamaño de las tablas y optimizar las consultas y operaciones, especialmente cuando el sistema crece en complejidad y volumen de información.

- Control del Tamaño de las Tablas

- Segmentación de Datos: Aunque no siempre es posible predecir el crecimiento de una tabla, la estrategia empleada en la arquitectura AUREA consiste en segmentar las tablas grandes. Esto puede implicar dividir la tabla en particiones o utilizar técnicas de archivado para mantener un tamaño manejable.

- Para los datos históricos, se utiliza la segmentación por año o año-mes, agregando esta información al final del nombre de la tabla. Es importante asegurarse de que el nombre de la tabla no supere los 30 caracteres para evitar problemas de gestión en el futuro.

- En otros casos, cuando no es posible la segmentación por fecha, se pueden definir parámetros específicos basados en las reglas de negocio. Por ejemplo, para gestionar el historial de actividades de estudiantes, se asigna un contenedor a cada grupo de 30.000 estudiantes. De este modo, si el sistema cuenta con 300.000 estudiantes, la consulta del historial se realiza en dos pasos: uno para obtener los datos generales del estudiante y otro para acceder al contenedor específico con sus datos históricos. Esta estrategia divide la carga entre 10 tablas, mejorando significativamente el tiempo de respuesta frente a una única tabla que podría ser consultada simultáneamente por muchos usuarios.

- Indexación Eficiente: Los índices son una herramienta clave para acelerar las consultas y operaciones frecuentes en las tablas. Si una búsqueda frecuente en una tabla no está indexada y comienza a mostrar retrasos, se debe evaluar la posibilidad de agregar un índice en el campo consultado.

- No se recomienda indexar campos de tipo texto o memo. Para estos casos, cuando las consultas dependen de tablas enlazadas, es preferible realizar una búsqueda previa por los ID asociados a estos datos y relacionarlos con la consulta principal. Esto evita que el motor de base de datos tenga que realizar la búsqueda como un único proceso, mejorando el rendimiento.

- Monitoreo y Mantenimiento: Se recomienda realizar una revisión anual del tamaño de las tablas para asegurarse de que el crecimiento está bajo control. Además, es importante eliminar o trasladar datos históricos a bases de datos de históricos. Cuando se detecten cuellos de botella en el rendimiento, se deben evaluar las consultas involucradas y optimizar su diseño para mejorar la eficiencia.

- Objetivos

1. Garantizar tiempos de respuesta rápidos, incluso cuando el volumen de datos aumenta significativamente.

2. Facilitar operaciones de respaldo y recuperación, minimizando la complejidad de gestionar grandes volúmenes de información.

3. Prevenir problemas de rendimiento y escalabilidad a largo plazo, mediante una gestión proactiva del crecimiento de las tablas y las consultas.

3.6. Prácticas de Lecciones Aprendidas

- Implementación de Mejores Prácticas:

Algunas de las prácticas descritas en esta sección pueden parecer contra intuitivas, pero en la práctica han demostrado ser eficientes y se encuentran implementadas en entornos de producción, ofreciendo un alto rendimiento y optimización de recursos en el manejo de datos.

1. Mantener las Consultas lo más Simples Posibles: En entornos de alto rendimiento, las consultas complejas que involucren múltiples tablas pueden ralentizar el sistema. Un ejemplo común es la necesidad de listar estudiantes matriculados, donde participan varias tablas como la de matrícula y la de estudiantes. En lugar de hacer una única consulta que enlace ambas tablas y muestre los campos deseados, se recomienda dividir la operación en dos partes:

- Primero, realizar una consulta que obtenga los datos de matrícula.
- Luego, realizar una consulta separada que traiga los datos del estudiante.

Esta separación puede parecer más costosa en términos de procesamiento, pero en un entorno de alto rendimiento es más eficiente, ya que reduce la carga de las transacciones complejas. Además, si las tablas secundarias tienen un número bajo de registros, se puede cargar primero estos datos en un arreglo en memoria y luego consultar la información principal. Otra opción es consultar los datos derivados a medida que se necesitan, guardando en memoria los resultados (por ejemplo, datos del profesor asignado).

2. Actualización de Datos en Momentos de Estrés: Las actualizaciones en momentos de alta carga de la base de datos son uno de los principales factores que afectan el tiempo de respuesta. Un ejemplo típico es el cierre de periodo académico en la universidad, cuando miles de profesores califican a sus estudiantes simultáneamente. Durante este proceso, el sistema puede llegar a manejar más de 4 millones de registros que deben ser totalizados y procesados en paralelo.

Para reducir la carga en estos momentos, se emplean las siguientes estrategias:

- Separar procesos de totalización que pueden ejecutarse en otro momento, aunque esto implique mostrar un mensaje de espera a los usuarios.
- Realizar actualizaciones masivas en lugar de procesar registró a registro.

- Cuando no es posible, hacer una consulta previa que verifique qué datos necesitan actualización y luego filtrar los resultados por los ID únicos de las tablas. Este enfoque evita que la base de datos realice una búsqueda y filtrado complejos, que pueden colapsar el motor en situaciones de alta demanda.

3. Ajuste de Consultas y Modelo de Datos: Aunque algunas de estas cuestiones se pueden mitigar aumentando la capacidad de procesamiento, esta solución a largo plazo genera costos crecientes que pueden hacer inviable la plataforma. Es por ello que, además de optimizar las consultas, se deben diseñar procesos que reduzcan el impacto de estas cargas, equilibrando el costo de desarrollo inicial con el costo continuo de procesamiento.

Para más detalles sobre estos enfoques, existe un documento complementario al curso AUREA exclusivamente dedicado a la gestión de datos que contiene ejemplos prácticos y destaca la importancia de diseñar consultas óptimas para que el modelo sea lo más eficiente posible.

4. Diseño de los Módulos

El objetivo principal de la arquitectura AUREA al definir los módulos es mantener una uniformidad en el desarrollo y garantizar un nivel de calidad alto en el software producido, promoviendo la adopción de mejores prácticas. Las reglas aplicables al diseño de los módulos permiten flexibilidad y eficiencia, mientras aseguran que el sistema sea mantenible a largo plazo.

- 4.1. Estructura Visual en la Arquitectura AUREA

La presentación visual de un módulo en la arquitectura AUREA está dividida en seis partes:

1. Encabezado y menú del módulo.

2. Formulario de edición del módulo principal, con un conjunto de botones generalmente ubicados en la parte superior derecha.

3. Formularios de edición de módulos secundarios, cada uno con su propio conjunto de botones.

4. Área de datos existentes, que funciona como buscador para filtrar datos.

5. Sectores auxiliares para tareas derivadas, como búsqueda o carga de archivos adjuntos.

6. Pie de página del módulo.

No se recomienda el uso de controles tipo acordeón por la dificultad de manejo para personas con limitaciones visuales. En general, los usuarios prefieren tener toda la información desplegada en la pantalla (como en una historia clínica), lo que facilita la navegación.

- 4.2. Consideraciones Backend

En cuanto al backend, se recomienda mantener la funcionalidad en archivos separados, procesando la información completamente antes de enviarla a los usuarios. Aunque es común separar el diseño del controlador, en sistemas de información esto no es tan relevante como en sitios web. Optar por una presentación más estática puede mejorar los tiempos de respuesta, optimizando el rendimiento del sistema. Además, los archivos de frontend (CSS y JavaScript) deben descargarse una sola vez para optimizar el canal de datos.

- Estructura General del Backend

Se cuenta con las siguientes secciones:

1. Sección de Dependencias Generales: Aquí se cargan las dependencias que son comunes para todo el sistema o para un conjunto de módulos. Esto incluye librerías, controladores y cualquier recurso compartido que el módulo pueda necesitar.

2. Sección de Inicialización de Variables Globales y Control de Acceso: Se configuran las variables globales que son requeridas para el funcionamiento del módulo, además del sistema de control de acceso que valida los permisos del usuario para asegurar que tenga los privilegios necesarios.

3. Sección de Dependencias Específicas del Módulo: Esta sección carga las dependencias específicas del módulo que está en ejecución. Puede incluir librerías particulares adicionales requeridas solo por ese módulo.

4. Sección de Validación de Datos de Entrada: Toda variable que puede ser pasada como parámetro debe ser validada en esta sección. La validación de los datos es crucial para evitar inyecciones de código o entradas no válidas que puedan afectar la integridad del sistema.

5. Sección de Llamadas a Funcionalidades y Procesamiento de Información: Aquí se realiza el procesamiento de la información. En los módulos estándar, las operaciones comunes incluyen:

- Traer un registro existente.
- Guardar o actualizar un registro.
- Ejecutar tareas sobre un registro.
- Eliminar un registro.

Estas operaciones se encuentran estandarizadas, y la forma en que se implementan dependerá del manual de estilo específico para el lenguaje de programación utilizado.

6. Sección de Alistamiento de la Forma: Esta parte prepara los datos que el usuario verá en el frontend.

7. Sección de Alistamiento de Datos Existentes: Prepara los datos existentes que deben mostrarse al usuario en el bloque de datos existentes, cargándolos para su presentación.

- 4.3. Independencia de los Módulos

Cada módulo dentro del sistema debe ser independiente y estar controlado por un número de módulo. Este número de módulo se aplica para gestionar los permisos, control de acceso, y la identificación de las librerías. Si un módulo gestiona varias tablas, como por ejemplo matrícula y cursos matriculados (que son dos tablas diferentes), se asignarán números de módulo independientes a cada tabla, aunque el acceso al módulo general se controle solo por el número del módulo principal (matrícula).

- 4.4. Tipos de Módulos

A pesar de que el código está altamente estandarizado, la arquitectura es lo suficientemente flexible para soportar varias tipologías de módulos que se desarrollan como sigue.

- 4.4.1. Módulos Básicos

- Definición: Estos módulos interactúan con una sola tabla y gestionan operaciones simples (CRUD).

- Características:

  - Simplicidad: Son fáciles de desarrollar y mantener.

  - Uso Común: Se utilizan en funcionalidades básicas del sistema.

  - Rendimiento: Al interactuar con una sola tabla, suelen ser muy eficientes.

- 4.4.2. Módulos Padre-Hijo

- Definición: Estos módulos gestionan relaciones más complejas, con una tabla principal (padre) y una o más tablas relacionadas (hijo), como el caso de matrícula y calificaciones.

- Características:

  - Complejidad Controlada: Manejan datos complejos de forma organizada.

  - Integridad de Datos: Garantizan que los datos hijos dependan correctamente del padre. Al eliminar registros, se debe garantizar que los datos hijos se eliminen antes que los datos del padre.

  - Flexibilidad: Facilitan la extensión de funcionalidades añadiendo tablas hijas según sea necesario.

- 4.5. Mejores Prácticas en el Diseño de Módulos

- Consistencia en la Interfaz

Mantener un estándar de entorno facilita a los usuarios aprender una sola vez la estructura y saber dónde encontrar la información en cada módulo. Los botones de acciones comunes, como guardar o eliminar, deben ocultarse si el usuario no tiene permisos, evitando confusión.

- Validación y Manejo de Errores

Cada módulo debe incluir una variable controladora que indique la acción a realizar, inicializándola con una operación predeterminada (como limpiar el módulo). Además, debe haber una variable de error que, al contener un valor, detenga todas las operaciones. También se debe incluir una variable de depuración, inicializada en falso, que permita a los usuarios técnicos generar registros para el soporte.

- 4.6 Actividades de Documentación

La documentación en la arquitectura AUREA se organiza en tres etapas distintas, que permiten abordar los diferentes aspectos del desarrollo de software atendiendo las necesidades de los procesos involucrados.

1. Diseño del Componente

Esta etapa inicial se enfoca en la conceptualización del componente, asegurando que todas las partes interesadas comprendan su propósito y funcionamiento.

- Diagrama de Servilleta: Este diagrama proporciona una explicación global y simplificada del proceso que cubrirá el componente, ideal para una comprensión inicial.

- Diagramas Derivados: A partir del diagrama de servilleta, se elaboran diagramas más detallados que explican el proceso en términos más técnicos, pero aún entendibles para usuarios finales.

- Historia de Usuario: Un documento que describe las funcionalidades esperadas del componente desde la perspectiva del usuario, especificando lo que se debe lograr.

2. Diseño del Módulo

En esta etapa, la documentación se enfoca en el diseño y comportamiento específico de cada módulo, con énfasis en las interacciones de datos y las responsabilidades de cada elemento.

- Diccionario de Datos: Define y describe las tablas, campos y relaciones entre ellos, proporcionando una visión técnica clara de cómo se gestionarán los datos en el módulo.

- Documento de Responsabilidades: Detalla la tarea específica que ejecutará cada tabla, así como cualquier consideración adicional o restricción que deba aplicarse.

3. Producción

Una vez que los módulos han sido probados y están listos para su implementación en producción, se debe generar documentación que permita a los usuarios y administradores utilizar y mantener el sistema correctamente.

- Manual General del Sistema: Un documento que describe el funcionamiento general del sistema o del componente, incluyendo sus características y cómo operarlo.

- Instructivos Paso a Paso: Guías que detallan de manera clara y concisa cómo realizar operaciones específicas dentro del sistema, desde tareas simples hasta procesos más complejos.

- Infografías: Opcionales según las necesidades del módulo, las infografías pueden ofrecer una forma visual y resumida de comprender los procesos o flujos clave del sistema, facilitando la comprensión para usuarios finales.

Conclusión

La Arquitectura AUREA se ha desarrollado con el objetivo de proporcionar una estructura sólida, eficiente y adaptable para la creación de sistemas de información en entornos web. Este enfoque modular y basado en componentes transversales ofrece una combinación de rendimiento optimizado, control total sobre el desarrollo, y una alta personalización. Estas características son fundamentales para organizaciones que buscan mantener un sistema robusto y escalable a lo largo del tiempo, adaptándose a los crecientes desafíos del entorno tecnológico.

A lo largo de los años, AUREA ha demostrado su durabilidad en la Universidad Nacional Abierta y a Distancia (UNAD), donde se ha optimizado para ofrecer una respuesta rápida y seguridad robusta, aspectos clave para instituciones públicas expuestas a constantes intentos de intrusión. Las prácticas descritas en este documento han permitido gestionar de manera eficiente tanto los costos operativos como la escalabilidad del sistema, manteniendo un balance entre el uso de recursos y la entrega de valor.

Asimismo, la creación de librerías propias, la implementación de un código puro, y el énfasis en la documentación amplia unido al un programa de capacitación han permitido para garantizar la adopción de la arquitectura por parte de nuevos desarrolladores, dando continuidad a la evolución de los proyectos a lo largo del tiempo. Estos elementos, junto con la flexibilidad de las librerías compartidas y el enfoque en la gestión eficiente de datos, aseguran que AUREA se mantenga como una plataforma capaz de sostener el crecimiento y las necesidades cambiantes de la organización.

En resumen, la Arquitectura AUREA no solo permite la creación de sistemas robustos, escalables y eficientes, sino que también garantiza una mayor mantenibilidad y longevidad en entornos de desarrollo web, contribuyendo significativamente al éxito operativo de las organizaciones que la implementan.