SERVICIO NACIONAL DE APRENDIZAJE SENA

Plan de trabajo para un aplicativo de gestión de inventario

Título: Ordenamiento y alineación de columnas en Bootstrap 5

Juliana Tique
David Solano
Jaider Moreno

09/06/2025

TABLA DE CONTENIDOS

¿Por qué muchos proyectos pequeños terminan siendo difíciles de mantener o escalar?

Cuando comenzamos con proyectos pequeños, muchas veces la simplicidad y los plazos cortos nos llevan a no pensar en la escalabilidad o el mantenimiento a largo plazo. Esto puede llevar a varios problemas:

Falta de planificación a largo plazo: Muchos proyectos pequeños se desarrollan con un enfoque de "trabajar sobre la marcha". Esto puede resultar en una arquitectura y una estructura de código que funcionan bien en el corto plazo, pero que no están preparadas para escalar o adaptarse a nuevos requisitos.

Código espagueti: Con el tiempo, a medida que se añaden más funcionalidades, el código puede volverse desorganizado y difícil de entender. Esto se debe a la falta de una arquitectura clara y principios de diseño que guíen el desarrollo.

Poca modularidad: Un proyecto pequeño puede carecer de una separación adecuada entre las distintas partes de la aplicación. Por ejemplo, la lógica de negocio, la interfaz de usuario y la base de datos pueden estar demasiado entrelazadas, lo que hace que cualquier cambio en una de estas partes requiera modificaciones en muchas otras.

Falta de pruebas y documentación: Los proyectos pequeños a menudo se entregan rápidamente, pero esto puede resultar en la falta de pruebas automatizadas y documentación adecuada. Sin pruebas, las actualizaciones o correcciones pueden romper el sistema sin que se detecten errores a tiempo.

Escalabilidad de infraestructura: Los proyectos pequeños no siempre consideran cómo escalar la infraestructura. Si el proyecto crece en usuarios o en complejidad, la infraestructura subyacente puede no estar diseñada para soportar ese crecimiento.

Buenas prácticas y patrones

Uso de patrones de diseño: Aplicar patrones de diseño como el Singleton, Factory, Observer, Strategy, etc., puede ayudar a crear un código más modular y flexible.

Principios SOLID: Estos principios ayudan a tener un código más mantenible y escalable:

S (Single Responsibility Principle) – Un módulo debe tener una única responsabilidad.

O (Open/Closed Principle) – Los módulos deben estar abiertos para su extensión pero cerrados para su modificación.

L (Liskov Substitution Principle) – Los objetos deben ser reemplazables por instancias de sus subtipos sin alterar la correcta ejecución del programa.

I (Interface Segregation Principle) – Los clientes no deberían depender de interfaces que no utilizan.

D (Dependency Inversion Principle) – Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.

Diseño modular y desacoplamiento: Se debe trabajar en la creación de módulos independientes y reutilizables que se puedan cambiar sin afectar al resto del sistema.

Pruebas unitarias: Desarrollar con pruebas automatizadas permite refactorizar el código y agregar nuevas funcionalidades sin temor a romper algo que ya está funcionando.

Documentación clara: Aunque no siempre es el aspecto más emocionante, una buena documentación hace que el mantenimiento y escalado sean mucho más fáciles.

Patrones de arquitectura en el desarrollo de software

El patrón MVC (Modelo-Vista-Controlador) es uno de los más conocidos, pero existen muchos otros patrones de arquitectura que se pueden utilizar dependiendo de los requisitos y las características del proyecto. Algunos de los más populares son:

MVVM (Modelo-Vista-ViewModel):

Similar al MVC, pero en lugar de un controlador, utiliza un ViewModel, que es responsable de preparar los datos para la vista.

Uso típico: Aplicaciones en tecnologías como Xamarin o WPF.

MVP (Modelo-Vista-Presentador):

El Presentador es el intermediario entre la vista y el modelo. A diferencia de MVC, en el que el controlador puede manipular directamente la vista, el presentador tiene un control más explícito de las interacciones de la vista.

Uso típico: Aplicaciones de escritorio o móviles.

Capas (Layered Architecture):

El sistema está dividido en capas, cada una con una responsabilidad diferente, como presentación, lógica de negocio, acceso a datos, etc. Las capas más altas (como la presentación) solo pueden interactuar con las capas más bajas (como la base de datos) a través de interfaces.

Uso típico: Aplicaciones empresariales o sistemas grandes y complejos.

Microservicios:

Cada componente del sistema es un servicio independiente que interactúa con otros servicios a través de APIs. Esto facilita la escalabilidad y el mantenimiento al permitir que diferentes equipos trabajen de forma independiente en distintos servicios.

Uso típico: Grandes aplicaciones distribuidas.

Event-Driven Architecture (Arquitectura orientada a eventos):

Los sistemas se comunican mediante eventos, lo que permite la desacoplar los componentes. Es ideal cuando se necesita alta escalabilidad o sistemas asíncronos.

Uso típico: Sistemas en tiempo real, como plataformas de mensajería o aplicaciones financieras.

CQRS (Command Query Responsibility Segregation):

Se separa la lectura de la escritura de datos en dos modelos distintos. Esto permite optimizar cada parte por separado (por ejemplo, un modelo para consultas y otro para comandos).

Uso típico: Sistemas con altos volúmenes de transacciones y consultas.

Conclusiones

Los patrones de diseño son estándares para el buen manejo y desarrollo del software, permiten la comprensión , estructuración, cooperación y mantenimiento del software para que sea escalable.

Bibliografía

ChatGPT. (s/f). Chatgpt.com. Recuperado el 12 de mayo de 2025, de https://chatgpt.com/

Otto, M., & Thornton, J. (s/f-a). *Columns*. Getbootstrap.com. Recuperado el 12 de mayo de 2025, de https://getbootstrap.com/docs/5.3/layout/columns/

Otto, M., & Thornton, J. (s/f-b). *Grid system*. Getbootstrap.com. Recuperado el 12 de mayo de 2025, de https://getbootstrap.com/docs/4.0/layout/grid/