Parcial 2 de tecnología - Seminario de aplicación profesional - Alumno Gabriel Enrique Wolny

Punto 1

¿Por qué es necesario diseñar procesos de datos en forma distribuida? (20)

Distribuir los procesos de datos nos permite tener software y sistemas más escalables y adaptables. Cuanto más dividido se encuentren los procesos, funcionalidades, tendremos un control más preciso y certero de cómo se desarrollan las tareas.

Por ejemplo, si hay varios microservicios dedicados a la captura de los datos, con que uno se caiga, el resto podría seguir funcionando normalmente, siempre que se haya separado la base de datos o repositorio en que cada microservicio esté funcionando.

A su vez, quizá haya datos que no necesiten tanto cambio, pero si sean de mucha lectura. En tal caso, dividir los procesos nos permite que por ejemplo los datos de alta lectura vayan a una arquitectura/solución caché (Redis, DynamoDB, etc) mientras que otra más tradicional se aloje en un servidor SQL más convencional.

Punto 2

Estudie los modos de renderizado en Blazor Web-App en ASP.NET Core NET 8.0 y ejemplifique un caso práctico para cada uno (20)

En Blazor existen cuatro formas de renderizar[1]:

A. Static server
B. Interactive server
C. Interactive WebAssembly
D. Interactive Auto

Static server

Es el renderizado estático desde el lado del servidor, no es interactivo. Por default, los componentes usan este modo. Los request a los componentes de las páginas Razor son procesados por un middleware del ASP.NET Core del lado del servidor para el enrutamiento y autorización. Por eso funciones dedicadas de Blazor para enrutamiento y autorización no funcionan porque los componentes no son renderizados durante el procesamiento de los requests del lado del servidor.

Este modo nos puede ser útil en casos donde el SEO sea de mucha importancia, porque al ser el contenido estático, será indexado por los algoritmos de búsqueda de forma más rápida y duradera.

Podríamos utilizarlo en sitios web del tipo *landing*, ya que quizá una página de alguna empresa o negocio pequeño con información de contacto simple no necesitará muchas actualizaciones y el contenido se mantendrá estable a lo largo del tiempo. Inclusive se podría cachear el contenido con herramientas CDN como Cloudflare optimizando el ancho de banda necesaria en el hosting.

Interactive server

Acá las interacciones con el usuario se manejan en una conexión en tiempo real con el navegador. El circuito de conexión se establece cuando el componente del servidor es renderizado. Particularmente se utiliza conexiones SignalR para permitir actualizaciones e interacciones con el usuario.

Podremos utilizar este modo cuando necesitemos actualizaciones en tiempo real y una interfaz enriquecida para nuestros usuarios. Dashboards que tengan accesos a métricas de carga constante, apps colaborativas/creativas como generadores de imágenes o chats son algunos ejemplos donde podríamos usar este renderizado.

Interactive WebAssembly

En este modo, el renderizado se hace del lado del cliente utilizando Blazor WebAssembly. No se necesita una conexión constante con el servidor, una vez que ya se cargó con anterioridad a la aplicación. Como el código corre directamente en el navegador del usuario cliente, la velocidad de carga y actualización de algunas operaciones puede crecer muchísimo.

Si tenemos que armar una PWA (progressive web app), podremos hacer uso de este modo, por lo general en estos casos los datos se guardan también de manera local. Podríamos utilizar esto para plataformas de juegos o apps educativas que no precisan de conexión constante con el servidor. También, como el procesamiento y el almacén de datos se puede mantener local, podríamos utilizarlo con desarrollos que requieran más seguridad y no nos exponga a enviar datos sensibles

[1] https://learn.microsoft.com/en-us/aspnet/core/blazor/components/render-modes?view=aspnetcore-8.0

a un servidor, por ejemplo, para apps que tengan datos de nuestra ubicación en base al GPS del celular o de salud/pasos recorridos.

Interactive auto

Es el modo más flexible porque permite alternar los modos estáticos e interactivos de Blazor. Inicialmente usa Interactive server rendering con Blazor Server, pero en sucesivas visitas hace uso del renderizado del lado del cliente una vez que el paquete Blazor es descargado.

Podríamos utilizarlo en apps que combinen características que se detallaron anteriormente, quizá un mismo desarrollo precisa de un apartado en el mismo sitio web que tenga métricas en tiempo real, pero en el que el mismo sitio web tenga otro apartado con datos que no cambiaran esencialmente en el tiempo como de contacto o de ubicación.

En estos casos, al momento del runtime se determina cómo se renderizan los componentes. Debe notarse que el renderizado auto nunca cambia dinámicamente cómo renderiza un componente que ya se encuentra en una página. Este renderizado hace una decisión inicial de cómo es la interacción que tiene un componente, manteniendo dicha interactividad mientras esté en la página. El modo auto prefiere elegir un modo que coincida con componentes interactivos que ya existen para no introducir nuevos runtimes.

Punto 3

¿Qué necesidades tecnológicas son cubiertas por gRPC? Ejemplifique. (20)

gRPC es un framework de comunicación desarrollado por Google, lo especialmente útil que tiene es que permite una comunicación muy rápida y eficiente entre distintos servicios/componentes.

Por ejemplo, podríamos emplear gRPC en una solución de streaming, donde la codificación de video y la entrega de contenidos requieren mucha velocidad o en tiempo real con mejor optimización del ancho de banda.

Otro aspecto clave que gRPC aborda es la serialización y deserialización eficientes de datos. Al utilizar Protocol Buffers como formato de serialización predeterminado, gRPC puede serializar y deserializar datos de manera más compacta y rápida en comparación con formatos basados en texto como JSON. Ya conocemos que con grandes volúmenes de datos System.Text.Json de NET, la memoria y el garbage collector no se llevan bien. Esto se traduce en un menor consumo de ancho de banda y una mayor eficiencia en la transmisión de datos entre servicios.

Además, gRPC simplifica enormemente el proceso de desarrollo al ofrecer la generación automática de código a partir de los archivos de definición de servicio (.proto). Esto significa que los desarrolladores pueden definir la estructura de los mensajes y los servicios en archivos .proto, y gRPC se encargará de generar automáticamente el código del cliente y del servidor en una amplia gama de lenguajes de programación. A fines del usuario, quizá no represente mucho, pero se reduce la cantidad de código repetitivo y propenso a errores que los desarrolladores escribimos.

Por último, gRPC satisface la necesidad de comunicación bidireccional y en tiempo real a través de su soporte para el streaming bidireccional. Esto permite que tanto el cliente como el servidor envíen y reciban datos de forma continua, lo cual es esencial en escenarios como la transmisión de datos en tiempo real, la carga y descarga de archivos grandes, y las aplicaciones de colaboración en tiempo real.

Punto 4

¿Qué necesidades tecnológicas son cubiertas por SignalR? Ejemplifique. (20)

En cierto sentido, podría decirse que SignalR que está desarrollado por Microsoft para no depender tanto de gRPC. Una de las características más destacadas de SignalR es su capacidad para establecer una comunicación bidireccional fluida entre el servidor y los clientes. A diferencia de los enfoques tradicionales en los que los clientes deben sondear constantemente al servidor en busca de actualizaciones, SignalR permite que el servidor envíe datos y notificaciones a los clientes de manera instantánea. Esto se logra mediante el uso de WebSockets, un protocolo de comunicación avanzado que garantiza una latencia ultra baja y una eficiencia óptima.

SignalR también es muy útil para enviar notificaciones en tiempo real a los clientes. Por ejemplo, podríamos usarlo entonces para una app de redes sociales donde los usuarios deben ser notificados instantáneamente cuando reciben un nuevo mensaje, una solicitud de amistad o cuando son arrobados. Con SignalR, estas notificaciones se pueden enviar de manera eficiente y sin retrasos.

Punto 5

Compare ventajas y desventajas de desarrollar aplicaciones con arquitecturas que implementen clientes gruesos y clientes delgados. ¿Podría dar algún ejemplo de cada una? (20)

Los clientes gruesos son aplicaciones autónomas que asumen la mayor parte de la carga de procesamiento y lógica de negocio. Estas aplicaciones se ejecutan directamente en el dispositivo del usuario, aprovechando al máximo sus recursos de

hardware. Una de las principales ventajas de los clientes gruesos es su capacidad para ofrecer un rendimiento mejorado y una experiencia de usuario fluida, ya que no dependen de la latencia de la red para cada interacción.

En contrapartida, el proceso de desarrollo puede ser más complejo, ya que los desarrolladores debemos tener en cuenta las diferentes configuraciones de hardware y software de los usuarios. Además, la distribución y el mantenimiento de las actualizaciones pueden ser más difíciles, ya que los usuarios deben descargar e instalar manualmente las nuevas versiones de la aplicación.

En cambio, las apps de cliente delgados se basan en un navegador web y dependen del servidor para la mayor parte del procesamiento y la lógica de negocio. Los clientes delgados ofrecen una mayor accesibilidad, ya que los usuarios pueden acceder a la aplicación desde cualquier dispositivo con un navegador web compatible, sin necesidad de instalaciones adicionales.

Sin embargo, los clientes delgados dependen de una conexión de red estable para funcionar correctamente, y la velocidad y la calidad de la conexión pueden afectar la experiencia del usuario. Además, las funcionalidades offline son limitadas, ya que la mayoría de las tareas requieren comunicación constante con el servidor.

En resumen, la elección entre clientes gruesos y delgados depende de los requisitos específicos de la aplicación y de las prioridades del proyecto. Los clientes gruesos son ideales para aplicaciones que requieren un alto rendimiento, capacidades avanzadas y funcionalidades offline robustas. Por otro lado, los clientes delgados son adecuados para aplicaciones que priorizan la accesibilidad, la facilidad de mantenimiento y la colaboración en tiempo real.