ChatGPT

Arquitetura de API Centralizada com Leituras Globais Otimizadas

Visão Geral

Imagine um sistema de monitoramento de torres (por exemplo, torres de telecomunicações) que precisa exibir dados estáticos das torres *e* seus status operacionais em tempo real para usuários espalhados globalmente. Para impressionar em uma entrevista de arquitetura, podemos propor uma solução utilizando os serviços adequados de forma **otimizada para alta disponibilidade global e baixa latência**. A ideia central é combinar um banco de dados relacional para dados estáticos com um banco NoSQL distribuído globalmente para dados dinâmicos em tempo real, expondo tudo via uma API centralizada. Além disso, utilizar **comunicação em tempo real** para atualizar instantaneamente o front-end quando houver mudanças de status nas torres. Em resumo, a arquitetura inclui:

- **Azure SQL Database** (SQL Server gerenciado na nuvem) para armazenar dados **estáticos e "core"** das torres (propriedades fixas como localização, altura, proprietário, etc.), que mudam raramente.
- **Azure Cosmos DB** (NoSQL) para armazenar os **status em tempo real** de cada torre (online/offline, alarmes, métricas de throughput), com replicação global para leitura local em múltiplas regiões.
- **API Web .NET Core** centralizada que expõe endpoints REST. Essa API lê os dados estáticos do Azure SQL e os dados dinâmicos do Cosmos DB, agregando as informações para o cliente. Também incorpora o **SignalR** para enviar atualizações em tempo real ao front-end quando um status muda.
- **Aplicação Front-end (Angular + PrimeNG)** exibindo um *dashboard* web com mapa e lista de torres, consumindo a API. O front-end mantém conexão via SignalR para receber notificações instantâneas de mudanças (por exemplo, uma torre fica offline).

Essa combinação mostra um entendimento claro de quando usar um banco relacional tradicional versus um banco NoSQL distribuído, além de demonstrar domínio de conceitos de arquitetura para alta disponibilidade global e atualização em tempo real.

Componentes da Solução

- **Azure SQL Database:** Banco de dados relacional gerenciado (SQL Server) utilizado para persistir os dados principais de cadastro das torres. Por ser relacional, suporta esquemas estruturados e consultas SQL complexas. É adequado para informações que requerem consistência transacional e apresentam relações (por exemplo, torre pertencente a uma determinada região ou empresa) [1]. Os dados nesse repositório mudam pouco (são relativamente estáticos), então uma instância primária (com possíveis réplicas de leitura geo-distribuídas, se necessário) é suficiente.
- **Azure Cosmos DB:** Banco de dados NoSQL globalmente distribuído, ideal para dados em tempo real que exigem **baixa latência e alta escalabilidade**. O Cosmos DB replica automaticamente os dados em múltiplas regiões, oferecendo leituras e escrituras multi-região com **alta disponibilidade e baixa latência** ao redor do mundo [2]. Neste contexto, ele armazena o status atual de cada torre (por exemplo, online/offline, alertas ativos, taxa de utilização), dados que são atualizados com alta

frequência. O Cosmos DB foi projetado para **aplicações de acesso em tempo real** e fornece desempenho rápido e previsível mesmo em escala global [3].

- **API .NET Core:** Uma aplicação back-end construída em .NET (por exemplo, ASP.NET Core Web API) que atua como camada de serviços central. Esta API expõe endpoints RESTful para o front-end obter informações. Ao receber requisições, a API faz a leitura dos **dados estáticos** das torres a partir do Azure SQL Database (por exemplo, detalhes da torre como nome, localização, altura, etc.) e obtém os **dados dinâmicos** de status a partir do Azure Cosmos DB (por exemplo, se a torre está online, qual o throughput atual). A lógica da API integra essas informações, garantindo que o cliente receba um modelo unificado (por exemplo, uma lista de torres com seus atributos estáticos e status atual). Além disso, a API hospeda um **hub SignalR** (ou integra-se ao Azure SignalR Service) para comunicação em tempo real com os clientes conectados. Quando um status de torre muda (por exemplo, uma torre cai offline), a API (ou um serviço associado) emite um evento via SignalR para **notificar instantaneamente** o front-end. Essa capacidade de *push* em tempo real evita a necessidade de *polling* constante do cliente, economizando recursos e permitindo atualizações imediatas [4].
- **Front-end Angular + PrimeNG:** Aplicação web cliente, construída em Angular, usando a biblioteca de componentes PrimeNG para uma interface rica (por exemplo, tabelas, painéis, mapas). Esse dashboard mostra um mapa com as torres plotadas e uma lista detalhada. No carregamento inicial, o front-end chama a API .NET via HTTP (REST) para obter a lista de torres e seus status atuais. A partir de então, o front-end mantém uma conexão aberta através do **SignalR** (websocket) com o servidor para receber atualizações em tempo real. Por exemplo, se uma torre que estava “online” passar para “offline”, o backend enviará imediatamente uma mensagem via SignalR, e o código Angular atualizará a interface daquela torre (mudando a cor do marcador no mapa, exibindo um alerta, etc.) sem que o usuário precise atualizar a página. Isso proporciona uma experiência em tempo real fluida ao usuário final.

Fluxo de Dados e Interação

1. **Carga Inicial:** O usuário acessa o dashboard web (Angular), que então envia uma requisição HTTP para a API .NET Core buscando os dados das torres. A API, por sua vez, consulta o Azure SQL Database para recuperar as informações estáticas de cadastro (por exemplo, ID da torre, localização geográfica, características físicas, proprietário) e consulta o Azure Cosmos DB para obter o status atual de cada torre (por exemplo, estado operacional, indicadores de alarme, métricas recentes). A API combina esses dados em uma resposta consolidada (p. ex., uma lista de objetos Torre com campos estáticos e dinâmicos) e retorna ao front-end.
2. **Exibição no Dashboard:** O front-end recebe os dados e exibe o mapa com marcadores para cada torre em sua localização, bem como uma lista/tabulação com detalhes. Cada item inclui o status corrente (por exemplo, um ícone verde para online, vermelho para offline, etc.), obtido do Cosmos DB via API. Como os dados estáticos raramente mudam, eles podem ser armazenados em cache no front-end ou mesmo em cache no nível da API para otimizar desempenho em requisições subsequentes. Já os dados de status refletem a **situação em tempo real**, portanto é crucial que estejam atualizados.
3. **Atualização em Tempo Real:** O diferencial dessa arquitetura é reagir rapidamente a mudanças no status das torres. Suponha que uma torre que estava operacional sofra uma queda (falha) e mude seu status para *offline*. Assim que essa mudança é registrada – por exemplo, um sistema de monitoramento detecta e escreve a atualização no Azure Cosmos DB – a **API .NET** ou um componente serverless (como uma Azure Function conectada ao *change feed* do Cosmos DB) é

notificado da alteração. Imediatamente, a API emite um evento via SignalR a todos os clientes conectados que precisam saber dessa mudança. No dashboard Angular, a aplicação já está conectada através do SignalR e recebe esse evento em tempo real, podendo então atualizar a interface do usuário instantaneamente (por exemplo, exibindo a torre em vermelho e mostrando um aviso de "torre offline"). Tudo isso ocorre **sem qualquer intervenção do usuário ou refresh manual**, graças à capacidade de *push* do SignalR.

4. **Interação do Usuário:** Se o usuário fizer alguma ação (por exemplo, solicitar detalhes de uma torre ou reconhecer um alarme), o front-end chamará novos endpoints REST da API .NET. A API pode então gravar alguma informação (por exemplo, inserir um registro de acknowledgment em uma base) ou retornar dados solicitados, possivelmente envolvendo tanto o Azure SQL quanto o Cosmos DB conforme a natureza da operação. Independentemente das interações, o canal em tempo real permanece ativo para enviar novas atualizações de status conforme elas acontecem.

Esse fluxo garante que o usuário sempre veja informações atualizadas: dados mestres confiáveis (do SQL) combinados com dados de telemetria em tempo real (do Cosmos DB). A separação de responsabilidades por tecnologia (relacional vs NoSQL) otimiza a performance e escalabilidade de cada tipo de dado. Vale notar que, como o Cosmos DB é multi-região, **usuários em diferentes continentes terão tempos de resposta similares** para os dados de status, pois estão lendo de réplicas locais do Cosmos, ao passo que a API central e o banco SQL (se hospedados em uma região específica) podem adicionar alguns milissegundos de latência para usuários distantes nos dados estáticos – porém esses são menos volumosos e menos voláteis, minimizando impacto.

Alta Disponibilidade e Desempenho Global

Uma preocupação fundamental em arquiteturas modernas é proporcionar **alta disponibilidade** do sistema e **baixa latência** de acesso para usuários globalmente distribuídos. A solução proposta aborda isso em vários níveis:

- **Banco de dados globalmente distribuído (Cosmos DB):** O Azure Cosmos DB foi escolhido para os dados dinâmicos justamente por sua capacidade nativa de distribuição global. Com alguns cliques ou configurações, é possível replicar os dados do Cosmos DB para múltiplas regiões do Azure, próximas de onde os usuários estão. Isso significa que as leituras dos status das torres podem ser atendidas por uma réplica na região geográfica mais próxima do usuário, garantindo latência mínima. De fato, o Cosmos DB oferece **escrita e leitura multi-região**, proporcionando *failover* transparente e altíssima disponibilidade (99,999% para leituras, dependendo da configuração de múltiplas regiões) [2]. Essa característica garante que, mesmo que haja uma falha completa em um datacenter/região, os dados de status ainda estarão acessíveis em outras regiões, mantendo o sistema funcional.
- **Banco de dados relacional gerenciado (SQL) com replicação opcional:** O Azure SQL Database, embora não tão trivialmente replicável quanto o Cosmos, também suporta estratégias de alta disponibilidade. Podemos configurar réplicas de leitura georeduntantes (Azure SQL permite até 4 réplicas read-only) em regiões secundárias [5]. Na prática, dado que os dados de cadastro de torres mudam pouco, poderíamos manter um único nó primário (por exemplo, nos EUA) e réplicas em outros continentes para melhorar a latência de consultas de leitura, se necessário [6]. Entretanto, muitas vezes esse nível de otimização pode não ser crítico para os dados estáticos, podendo o sistema tolerar uma pequena latência adicional nas leituras de cadastro. O importante é que o Azure

SQL fornece garantias ACID e consistência forte para os dados relacionais, enquanto o Cosmos DB lida com a escala e distribuição dos dados de telemetria.

- **API e Serviços escaláveis:** A camada de API .NET pode ser implantada em um serviço escalável (como Azure App Service ou Kubernetes), com possibilidade de **deploy multi-regional** também. Em um cenário mais avançado, poderíamos distribuir instâncias da API em, por exemplo, Américas, Europa e Ásia, e usar um gerenciador de tráfego global (Azure Traffic Manager ou Azure Front Door) para encaminhar cada cliente à API mais próxima geograficamente. Assim, as chamadas REST iniciais também teriam baixa latência. Contudo, mesmo com uma única implantação central da API, a maior parte da carga (consulta aos status) recai sobre o Cosmos DB que já é otimizado globalmente. O uso do SignalR ajuda a escalar para muitos clientes sem eles terem que ficarem perguntando constantemente o estado – em vez disso, um único push de atualização alcança todos, reduzindo a carga e o tempo de propagação das informações. Vale citar que o Azure SignalR Service permite escalar para **milhões de conexões simultâneas**, oferecendo inclusive suporte a várias regiões e failover transparente caso se deseje uma topologia global para a comunicação em tempo real [7] [8].
- **Desacoplamento e Tolerância a Falhas:** Separar os dados em dois repositórios (SQL e Cosmos) também adiciona certa resiliência – mesmo que a parte em tempo real fique temporariamente indisponível, os dados fundamentais de cadastro ainda podem ser exibidos (com um indicador de "dados de status indisponíveis no momento", por exemplo). E vice-versa: se o SQL estiver indisponível em algum momento, usuários ainda poderiam ver dados *cacheados* de cadastro e receber atualizações de status do Cosmos (pelo tempo que o cache for válido), assegurando funcionalidade degradada porém útil. Em ambos os serviços, há garantias robustas de backup, replicação interna e recuperação de desastres fornecidas pela plataforma Azure.

Em suma, essa arquitetura tira proveito das **características fortes de cada tecnologia** para atingir alta disponibilidade e desempenho global: o Azure Cosmos DB lida com distribuição geográfica e escala de throughput, enquanto o Azure SQL garante integridade e simplicidade para os dados relacionais. A combinação dos dois, orquestrada por uma API bem projetada, resulta em um sistema globalmente otimizado.

Diagrama da Arquitetura

Figura: Diagrama da arquitetura proposta integrando Azure SQL Database (dados estáticos das torres) e Azure Cosmos DB (status em tempo real) através de uma API .NET Core centralizada. O front-end Angular/PrimeNG consome a API via HTTP e recebe atualizações em tempo real via SignalR. O Cosmos DB é replicado globalmente para oferecer baixa latência de leitura dos status.

Acima, o diagrama ilustra os principais componentes e fluxos de comunicação da solução. O **cliente web (dashboard Angular)** se comunica com a **API .NET** por HTTPS para operações comuns (①). A API, hospedando a lógica de negócio, consulta o **Azure SQL** para dados fixos (②) e o **Cosmos DB** para dados voláteis de status (③). A qualquer momento em que haja uma mudança significativa (como alteração de status), o servidor utiliza o canal **SignalR** para *push* ao cliente (④), garantindo que o dashboard esteja sempre atualizado em tempo real sem consultas adicionais. Notar que o Azure Cosmos DB, representado no diagrama, está distribuído globalmente (podendo haver réplicas em múltiplas regiões do Azure), embora isso seja transparente para a aplicação – do ponto de vista da API, ela consulta o Cosmos através de um endpoint único e este serviço se encarrega de direcionar a consulta para a réplica mais próxima disponível [9] [10].

Conclusão

A arquitetura proposta demonstra um conjunto de boas práticas e escolhas tecnológicas que certamente causarão uma boa impressão em uma entrevista técnica. Ao explicar essa solução, enfatize os seguintes pontos-chave:

- **Escolha da tecnologia adequada para cada tipo de dado:** dados relacionais e de cadastro em Azure SQL Database, garantindo consistência e schema bem definido; dados de telemetria e estado em Azure Cosmos DB, aproveitando escalabilidade e replicação global [3] [1]. Essa separação mostra entendimento de *trade-offs* entre bancos relacionais e NoSQL distribuído.
- **Alta Disponibilidade e Alcance Global:** O uso do Cosmos DB multi-região garante que usuários ao redor do mundo acessem informações de status com mínima latência, e que o sistema continua operando mesmo diante de falhas regionais [2]. Mencionar as diferenças de replicação global entre Cosmos e SQL (Cosmos com múltiplas réplicas *versus* SQL com replica read-only limitada) evidencia conhecimento aprofundado [5].
- **Atualizações em Tempo Real (Reactive Architecture):** A integração do SignalR para push de eventos em tempo real mostra preocupação com experiência do usuário e eficiência. Explique que, em vez de cada cliente ficar consultando o servidor periodicamente (*polling*), o servidor notifica somente quando necessário, economizando recursos e provendo dados frescos instantaneamente [4]. Esse padrão é muito valorizado em sistemas modernos (dashboards, IoT, etc.), e citar o Azure SignalR Service como facilitador de escalabilidade adiciona ainda mais valor arquitetural.
- **Escalabilidade e Flexibilidade:** Destacar que a solução pode escalar horizontalmente – mais instâncias da API, sharding no Cosmos DB por partição de dados, elasticidade do Azure SQL aumentando DTUs ou vCores conforme demanda – demonstra preocupação com crescimento futuro. Além disso, é uma arquitetura relativamente desacoplada: cada componente (front-end, API, SQL, Cosmos) poderia ser substituído ou evoluído independentemente (por exemplo, poderíamos introduzir um cache distribuído, ou substituir o front-end por um mobile app consumindo os mesmos serviços, etc.).

Em um cenário de entrevista, concluir afirmando que **essa arquitetura equilibra consistência de dados, desempenho em tempo real e alcance global** mostra uma visão madura de arquitetura. Você evidenciou saber **quando usar Cosmos DB versus SQL Server**, e como orquestrá-los com uma API bem construída e um front-end reativo, atingindo um resultado de alta disponibilidade global. Com um diagrama claro (como o apresentado) e estes argumentos, você certamente demonstrará domínio de arquitetura de soluções em nuvem de forma concisa e impactante.

Referências: Para embasar a proposta, você pode mencionar a documentação oficial e blogs técnicos: o Azure Cosmos DB é descrito como *"banco de dados multi-modelo globalmente distribuído, projetado para aplicações com acesso a dados em tempo real, baixa latência e alta escalabilidade"* [3], oferecendo distribuição automática de dados entre regiões para garantir baixa latência global [2]. Já o Azure SQL Database é um serviço relacional gerenciado adequado para dados estruturados e transacionais [1], com suporte a réplicas de leitura geográficas para distribuição limitada [5]. A combinação desses dois, cada qual no seu ponto forte, é uma prática recomendada quando se deseja **equilibrar consistência relacional e performance global**. Por fim, tecnologias como SignalR permitem adicionar funcionalidades em tempo real a aplicações web facilmente, **empurrando atualizações do servidor aos clientes conectados** sem necessidade de polling constante [4] – ideal para dashboards e sistemas de monitoramento ao vivo. Essas fontes corroboram as decisões tomadas na arquitetura, reforçando sua validade.

1 3 Azure Cosmos DB vs. Azure SQL Database: Which One is Right for Your Data Needs? – The Tech Guy
https://thetechguyin.wordpress.com/2023/04/14/azure-cosmos-db-vs-azure-sql-database/

2 SQL vs Cosmos database - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/2084372/sql-vs-cosmos-database

4 7 8 What is Azure SignalR Service? | Microsoft Learn
https://learn.microsoft.com/en-us/azure/azure-signalr/signalr-overview

5 6 9 10 Global Databases: Cosmos DB vs Azure SQL Database
https://pragmaticworks.com/blog/global-databases-cosmos-vs-azure-sql