Contesto:

Planteamiento — se describe una arquitectura moderna en la nube, con un enfoque con DDD + Hexagonal + API + tokens basada en una tecnológia sólida, escalable y compatible con Azure y React Native.

1. Objetivo General

Tener una aplicación Backend modular, desacoplado y escalable que va a ser consumido por una aplicación React Native

APIs seguras (autenticación y autorización)

Base de datos NoSQL

Preparación para despliegue en Azure (App Service, Container Apps o Kubernetes)

Como objetivo principal y para una implementación sin limitaciones, la visión es realizar un desarrollo escalable, pero que al inicio va a estar desplegado en un solo servidor en la nube. Abaratando costos lo que mas se pueda, pero que a un futuro y dependiendo de la demanda del mismo estos apis desarrollados modularmente con arquitectura DDD vayan siendo migrados a dokers y kubernets para su contenerización y escalamiento. De igual manera la BDD que puede ser No relacional o relacional dependiendo de como evolucione la arquitectura implementada no sea una limitante sino una ayuda para facilitar el cambio o transición.

Por ahora la BDD puede ser local, no se si eso se pueda, pero analiza la posibilidad porque la información que se va almacenar no va a ser muy grande, pero si variable y dependiente, pero recuerda que lo que busco es abaratar costos lo que mas se pueda y a un futuro migrar esta información y estructura a la BDD como Cosmo DB o Sql Server. Por lo que la arquitectura hezagonal es excelente para realizar la transición y de igual manera contemplar que las consultas deben estar orientadas a la implementación de Entity Framework o su simil dependiendo de la tecnología que en este caso es NestJs

Capa Lógica / Backend – Tecnología Recomendada Node.js con NestJS

(Para la velocidad, compatibilidad, escalabilidad, soporte cloud, DDD/Hexagonal):

Vamos a implementar una Arquitectura modular perfecta para DDD y Hexagonal.

NestJS ya soporta OpenAPI (Swagger), JWT, Guards, y estructuras limpias para casos de uso.

Integración a una BDD en Cosmos DB (Azure) a futuro. No importa si por ahora es local

📱 React Native App (Front)

|

▼

🌐 API Gateway / BFF (NestJS)

|

▼

🧩 Aplicación (Dominios / Casos de Uso)

|

├── Infraestructura (DB, Integraciones, Cloud Services)

├── Repositorios (implementaciones)

└── Core (Entidades + Interfaces)

Esta estructura nos permitirá tener:

Capas bien separadas

Interfaces de puertos y adaptadores

Testabilidad alta

Cambio de proveedor de BDD sin afectar el dominio

Base de Datos No Relacional

Azure Cosmos DB (modo Mongo API) si consideras una mejor opción temporal que se pueda implementar en un solo servidor (Front, Back y BDD) No hay problema

Autenticación y Comunicación Front–Back

🔐 Tokens

Implementa JWT (JSON Web Tokens) con expiración y refresh tokens.

Usa @nestjs/jwt en el backend y almacena el token de forma segura en el cliente (Secure Storage o Keychain/Keystore).

🧾 Estructura de Petición

Header propuesta: Si consideras una mejor estructura no hay problema, pero en el header debe viajar el código de la empresa porque estamos implementando arquitectura multiempresas, usuario, legacy o aplicación desde donde se consume el servicio, dirección ip desde donde se consume, Authorization: Bearer <token> y Content-Type: application/json

Body (ejemplo):

{

"data": dentro de data toda la estructura genérica de respuesta del servicio

}

Message: El message debe venir siempre con la respuesta. El objetivo es que esta sección me centralice que sucedió con mi petición ya que puede ser satiscatoria o no por lo que la capa superior lo primero que hará es leer esta sección para saber que hacer o presentar por ejemplo mensajes de error, alertas, etc que deban ser comunicadas o presentadas en pantalla con la siguiente estructura (Es importante que me entiendas que el nombre es una sugerencia que si consideras debe ir con un mejor nombre o mas adecuado a su objetivo, modificalo):

{

code: Es el codigo de error en caso de una respuesta insatisfactoria o con error y 000 en caso de ser afirmativa o sin errores (corrige el nombre si consideras adecuado),

description: Aquí vendrá el mensaje a presentarse en pantalla como una notificación si es necesario mostrarlo como por ejemplo “No se ha logrado asignar permisos al usuario” (corrige el nombre si consideras adecuado),

details: Aquí va a viajar el detalle de error técnico porque pueda que no se haya asignar permisos al usuario como el ejemplo anterior, pero y porque. Entonces aquí viajaría el detalle del error presentado para que el técnico pueda saber que sucedió y dar solución al problema (corrige el nombre si consideras adecuado)

}

🚀 Comunicación segura

HTTPS obligatorio a un futurio (Azure lo maneja automáticamente). Por ahora como el objetivo es abaratar costos y va a estar en un solo servidor como “monolito” pos así decirlo aunque estructuralmente si va a estar estructurado correctamente para su escalabilidad se podría manejar http

CORS configurado para los orígenes del front. Estro no estoy seguro, pero dejo a tu criterio ya que mi conocimiento de CORS es muy básico

Arquitectura DDD

Por ahora vamos a enfocarnos en la comunicación, generación y validación de tokens para que tanto el front y back puedan hacerlo correctamente, pero sin perder la arquitectura DDD, eso quiere decir que vamos a segregar los módulos como apis independientes o controllers si así puedes entenderme de mejor manera y para que los fuentes no estén botados en repositorios independientes vamos a generar un monorepo ósea un contenedor que almacene todas las apis modulares que se vayan generando por ejemplo: Seguridades, Servicios, etc, cada uno con su puerto independiente, pero que sean consumidos desde un Gateway centralizado el cual se encarga de direccionar las peticiones a las diferentes apis, pero para el cliente es transparente, el apuntaría a un solo puerto