Design and Decisions

Service Architecture

I chose Vertical Slice architecture for the service. Vertical Slice has been my go-to choice of architecture style for the last two years and it has served well so far.

What I like about is the most is that It creates and forces coherency. Everything a feature needs to work is together. When you need to look up or modify a functionality you know where to look for. This makes adding and modifying new features much easier.

It also enables of taking out a feature into a new microservice if it grows too much without refactoring several layers.

Another positive effect of Vertical Slice is that it makes onboarding much easier. When everything is together it becomes much easier to understand how it works.

Why not Clean/Hexagon/Onion/etc.?

To my experience these N-layered architectures become too complex when they grow. They become Dependency Injection, abstraction and indirection hell. To add or modify a feature you need to write code to several different layers. This becomes such a hassle down the line even for project veterans.

I chose Vertical Slice as a demonstration purpose. I wouldn't use it if the service is small/micro enough (e.g. has only one feature). In that case I would create a much smaller layered architecture.

CQRS

I created different handlers for each query and command.

I created a seperate bus' for queries and commands. Their responsibility is to forward requests to the handlers. I usually never include any business logic in API endpoints.

Why not use MediatR?

Last I've checked MediatR had only one pipeline, IRequest/IRequestHandler. There was no way to differentiate between commands and queries. This violates Responsibility Seperation and Single Responsibility principles of CQRS. To me it's basically a CQS Command Dispatcher. The difference between CQS and CQRS is that with CQRS you also need to make separations at infrastructure level.

I also tend to not include any new dependencies in a project as much as I can.

I did not include any pipeline filters like MediatR do, they can be done easily with DI decorators or Chain of Responsibility implementations.

Authentication

I used IdentityServer for a basic authentication/STS server, since it's the library I've experience with the most.

I used JWT authentication and scope-based authorization in the API.

Since the test application is console-type I used client credentials grant type to get access token for simplicity. I usually use code + PKCE flow for web-based applications. And if it was a security-critical application I'd look into using reference tokens and OAuth2/OIDC extensions like Demonstration of Proof-of-Possession, Pushed Authorization Request, Backchannel Logouts, etc.

Caching

I used Redis with StackExchange.Redis and Microsoft Distributed Cache libraries.

I used cache in GetPost CQRS handler but in a larger project I'd probably go either as an API Filter or Bus Filter.

Rate Limiting

I used Microsoft Rate Limiter with fixed window rate. The limit is 5 requests in 5 seconds. It's partitioned per client.

Packages

I've used following Nuget packages:

IdGen: It's a Twitter Snowflake-like Id generator library. It enables time-sortable, numeric, conflict-free Id generation in a distributed environment.

Mapperly: It's a object-to-object mapper. I chose it over AutoMapper because AutoMapper uses reflection and Mapperly creates mapping classes using source generations. This makes Mapperly multitudes of times faster than AutoMapper.

FluentValidation: I used this library for validations in domain handlers.

Testing

I used Repository Pattern for abstracting the ORM. This is a pattern I usually try to stay away because since Entity Framework's DbContext is already an abstraction I feel like using another abstraction on top of it is unnecessary redirection with little benefits.

I used in-memory SQLite to replace SQL Server during testing. I didn't mock DbContext because the extension method I used for querying, FirstOrDefaultAsync, can't be mocked since it's a static extension method.

I chose both of these methods for the simplicity for this task. If it was a regular project I'd probably spin up ephemeral containers in CI/CD pipeline or use testcontainers.com.

Test App

I created a console app for testing. It performs the following requests:

- Get access token
- Create Post
- Add two Comments to the Post
- Get Post with Comments
- Send 20 requests in parallel to test the rate limiter.

I've enjoyed working on this task. I'd like to thank you for the opportunity and hopefully see you in future tasks.