From Tech Challenge to Real-Life Transformations

Due to the limited time and resources for this challenge, I took many shortcuts. In a real-life scenario, with more time, access to additional resources, and a deeper understanding of the specific requirements, the solution would differ in several ways. Here are some points:

- Integration of Cloud Services: In a real-life scenario, I suggest leveraging cloud services like Amazon SQS, SNS, EventBridge, MQ, GCP Pub/Sub, and GCP Cloud Tasks. Or if we are Azure fans, we can use Service Bus, Event hubs, or Notification Hubs.
- In a real-life scenario, I'd explore using Elasticsearch, Solr or Azure cognitive search (Azure search previously). For efficient real-time search and complex query capabilities.
- I'd go with Azure Application Insights for monitoring, especially if we're using .NET. It's super-friendly for .NET and plays well with it, unlike most other monitoring tools. Features like Application Map, .Net profiler, and memory dumps make it very convenient to troubleshoot problems.
- Consider implementing SQL Server database partitioning to distinguish between delivered parcels and those in transit. This partitioning approach will significantly enhance query performance and reduce locking issues.
- But the next phase in optimizing query performance is to consider Elasticsearch or Solr. While they may be slightly less up-to-date and less reliable, they can handle millions of queries with ease, making them suitable for less sensitive services.
- Consider a non-relation Database, like MongoDB, as the schema will be very flexible compared to rational DB.

Assumptions

- Event API is an external system. We can't change it.
- I don't fully understand the data structure and the data relationship, So I will assume:
    - One parcel has many events.
    - Every event has one device and one user,
- What is the usage of Status Code, Device, and User?

Preparing for production

- Testing and UAT
- Load testing: We need to understand the flow rate of the events vs the processing rate of each event to ensure the queue will not be filling up, especially in high-load situations.
    - Assuming we can scale up workers in the cloud environment, we can increase/decrease the number of workers based on the queue size.
- Configuration to handle different environments.
- There should be more null checks.
- Entity Framework is very dangerous. If it's not used wisely in production, it could cause many issues. The main one is the queries can't be optimized. I'm using it here as it's quick.
- SQLite is by no means a prod-ready database; again, I'm using it here as it's quick and doesn't require much setup.
- Setup DevOps pipelines
- Automation testing, postman, or similar
- Pen Testing, firewalls, and infra design considerations.

- API authentication depends on the infrastructure (E.g. if Azure APIM and certificate-based)
- Load balancing and scaling and budget caping considerations.
- App monitoring, e.g., App insight
- Apps dependencies, i.e., which apps will be deployed to prod first and if there are any downtimes and business impact analysis.
- Rollback plan, disaster recovery plans.
- Domain names and SSL certificate considerations
- As-built documents and other design and user manual documentation.
- After the prod release, we will monitor the performance for a while.

Future enhancements

- I used a pulling approach throughout. This could be better. We should go with the push approach.
- Unit testing & integration testing
- I should only use public if it's meant to be public. I overuse it here, for simplicity's sake.
- Producer design needs to be more scalable. Although we assume one producer could handle the load of moving the scan events to the queue, as it's a relatively small load.

Solution Projects description

- Events API
    - The Event API serves as the primary source of events, simulating the generation of events by an external system.
- Producer Worker
    - The Producer Worker project continually pull events from the Events API and registers them in the queue.
- Consumer Worker
    - The Consumer Worker project is responsible for continuously dequeuing events from the queue and processing them. This processing involves restructuring the event and storing it in the Parcel Database.
- Data Access
    - The Data Access project, a shared project, contains all the repositories and DbContexts, it houses the business logic for managing these events and transfer them from/to Databases.
- Data Models
    - A shared project encompasses POCO classes as well as API and EF models.