Ian Chen's submission to Java Developer Code Challenge V1.1 - Trade Reporting Engine

Setup

1. Run `./gradlew bootRun` to start the service. Run `.\gradlew.bat bootRun` if using Windows / PowerShell.
    a. The app is ready when `Init duration for springdoc-openapi is` printed.
2. http://localhost:8080/swagger-ui/index.html#/ to try the trade submission and query APIs via Swagger UI.
    a. POST /trades/batch: submit your own XML trade event files
    b. POST /trades/batch/bundled: submit the files bundled in resources
    c. GET /trades/prefiltered: get the trades by filters. CHALLENGE (default - take-home task requirements) and ANY are available. Try UNREGISTERED to see error handling.
3. http://localhost:8080/h2-console/login.jsp with JDBC url `jdbc:h2:mem:hello` to inspect the in-memory H2 DB.
4. `./gradlew jacocoTestReport` to view coverage report under `build/reports/jacoco/test/html/index.html`

Design and implementation (<1 page)

- [ ] I used the typical three layer controller-service-repository class design. As the trade reporting engine is essentially a microservice, I used a controller-service-repository-model subdirs folder structure instead of vertical slices.
- [ ] A single controller calls two services. TradeQueryService orchestrates filter lookup and trade fetching. TradeSubmissionService processes user-submitted and bundled XML files. Both services call TradeRepository, which uses Spring Data JPA with domain-driven repository methods instead of DAOs to highlight db write events.
- [ ] RESTful APIs: paths have /trades prefix with nouns (/batch, /prefiltered) for subpaths that match their purpose.
- [ ] XML reading uses the stdlib DocumentBuilder and XPath implementations. As they are stateful and not thread safe, we can either use thread-local instances & reset them at the end of each request or create a new instance per request. I chose the latter as it's easier to guarantee per-request isolation. The stdlib DocumentBuilderFactory and XPathFactory factories are singletons as newXPath / newDocumentBuilder methods are thread-safe.
- [ ] I used the standard library InputStream interface in method arguments throughout the submission classes to make it easy to integrate and test the submission and XML mapping responsibilities with disparate data sources like HTTP multipart forms, resources bundled in the JAR, and strings. Coding to interfaces makes testing easier!
- [ ] The input stream -> XML document parsing in XmlModelMapper was decoupled from the mapping of XPath expressions to a parameterized/generic output type (e.g. Trade using TradeXmlExtractor). This aids testing, separation of concerns, and enables XmlModelMapper to be reused for any future model/DTO.
- [ ] Dependencies - all are Eclipse Public License (JUnit, H2) or Apache 2.0 (others) - can be used in corporate apps.
- [ ] Each filter implements a TradeFilter interface. Due to the arbitrary requirements of such filters (e.g. two fields of a record are anagrams of each other), I chose to implement these "predefined" filters in code, each with their own "kind". To minimise the network traffic from sending DB records over the network, there is both a database-level filter for filters that can be performed with simple raw SQL or JPA specifications, and an app-level filter for conditions that are cumbersome to express and deploy with stored procedures, like anagrams or other RPC calls.
- [ ] The TradeFilterFactory associates all known TradeFilters on the classpath with their kind, ensuring that each TradePredefinedFilterKind has exactly one enabled implementation at app startup.
- [ ] Adding more criteria later without impacting existing filters is easy and scalable. (1) Add a new variant to TradePredefinedFilterKind. (2) Create a new class implementing TradeFilter and register it as a Spring bean e.g. with @Component. (3) Call POST /trades/prefiltered?kind=YOUR_NEW_KIND.
- [ ] Composing two filters together can be done using the decorator and strategy patterns - the wrapper TradeFilter implementation can delegate to the underlying JPA specification and predicates, then || / && them together.
- [ ] I've added unit and integration tests, Flyway DB migrations on startup, and a CI job with GitHub actions.

Assumptions and tradeoffs made

- [ ] The TradeFilter interface exposes both an application-level filter and a DB filter (JPA Specification<T>), introducing some coupling between the service and repository layers. I think this is acceptable, as at some point a DB filter needs to be associated with an application-level filter to form a single business-level filter, while allowing efficient reads (DB filter) and complex predicates (app filter).
- [ ] Arbitrarily complex filter logic means writing code for predefined filters - I have kept it reasonably scalable still. A separate endpoint could be added which filters by a known, limited set of criteria without complex predicates, e.g. filter by currency, buyer party, and seller party only.
- [ ] The app filter predicate only considers the trade itself - reasonable IMO though one limitation of the current factory is that it can't dynamically build filters per-request. This can be added, but I've intentionally kept it simple as is.
- [ ] I have assumed there aren't too many trades, so I have not implemented keyset/cursor pagination which is needed to prevent denial of service or excessive costs. This is quite an involved problem as there is an additional app-level filter which can result in fewer records being returned than requested, even if there are more pages. Could be solved using a loop until page size is reached or with a downstream data store like ElasticSearch.
- [ ] Validation that leaf nodes of XPath expressions exist - defaults to empty string if not found. To solve, can add write-time validation when constructing the Trade model asserting that the values are non-empty. I could not find a way to perform strict path evaluation in the XML mapping layer instead - this would improve the robustness of "offline" input XML validation in which we don't try to resolve the schema URLs.
- [ ] DB indexes were chosen assuming the ability of the underlying query engine to perform bitmap index scans like Postgres, SQL Server (not H2). Extremely high volume of trades would likely involve moving older trades to an OLAP database, resulting in faster range queries with filters but with higher baseline latency than an OLTP db.
- [ ] I used auto-incrementing IDs as trade event data has no other client-supplied unique identifier. Alternatively could use UUIDv7 which offers reasonable DB performance due to temporal locality and can be generated in the app.
- [ ] No idempotency in the API atm. Can do this by adding a request/idempotency id to each request + Redis cache with key expiry
- [ ] I only used entity and API models/DTOs - in this small microservice, another intermediary domain model would add indirection with minimal benefit.