I Created a SOQL Query Builder That Was Immediately Adopted Company Wide

Context

We use Salesforce to manage the service activity portion of our service offering and we provide visibility to internal teams and client teams visibility a number of portals and apis. A lot of engineers touch SOQL every sprint.

We had several senior and staff engineers who knew how to write SOQL queries and everybody had their own way of doing it. Some queries used shared helper functions. Some manually stitched together strings. Some mixed both. That meant two engineers solving the same problem could produce two completely different implementations.
Here is what that inconsistency actually looked like.

Old query style #1: helper-driven composition

```
fun createQuery(relevantIds: List<String>, contactTypes: List<String>?): SoqlQuery {
    val caseOrWorkOrderCondition =
        orOp(
            inFilter("CaseId", relevantIds),
            inFilter("WorkOrderId", relevantIds),
        )
    val contactCondition =
        subQueryIn(
            filter = contactTypes,
            outerField = "ContactId",
            subQueryTable = "AccountContactLink",
            selectField = "ContactId",
            filterField = "ContactType",
        )
            return   SoqlQuery(listOfNotNull(BASE\_SELECT,   whereClause(caseOrWorkOrderCondition,
contactCondition)))
}
```

Old query style #2: manual string assembly

```
fun createQuery(request: WorkOrderSearchRequest): SoqlQuery {
    val clause =
        sequenceOf(
            fromFilter(request.workOrderIds, "Id"),
            fromFilter(request.caseIds, "Case.Id"),
            fromFilterNot(if (request.onlyOpen == true) CLOSED\_STATUSES else null, "Status"),
            fromFilter(request.assetIds, "Asset.Name"),
        ).filterNotNull().joinToString(" AND ")
    val where =
        if (clause.isBlank()) {
            ""
        } else {
            " WHERE "
        }
    return SoqlQuery(BASE_QUERY + where + clause)
}
```

These two queries were doing the same basic job: build a `SELECT ... FROM ... WHERE ...` statement with optional filters.

But they did it through two different mental models.

- One relied on helper functions like `orOp(...)`, `subQueryIn(...)`, and `whereClause(...)`.
- The other built a list of clause fragments, joined them manually, then handled spacing and `WHERE` insertion itself.

This was a problem because new members of the team found it challenging to write bug-free queries efficiently and code reviewers could not quickly pattern-match. Every query change required re-reading custom assembly logic.

Action

The absence of a standardized, safe integration pattern across teams was hindering delivery speed and stability. My first action was to audit our existing query features: optional filters, grouped boolean logic, nested subqueries, aggregation, ordering, and some other custom edge use cases.Then I spent a month to build a type-safe Kotlin DSL that solved this problem beautifully. The two queries above were refactored as below:

```
fun createQuery(relevantIds: List<String>, contactTypes: List<String>?): SoqlQuery =
    soql {
        select(FIELDS)
        from("VoiceCall")
        where {
            "CaseId" isIn relevantIds
            or { "WorkOrderId" isIn relevantIds }
        }
        and {
            "ContactId" isIn {
                select("ContactId")
                from("AccountContactLink")
                where { "ContactType" isIn contactTypes }
            }
        }
    }
```

And the work-order query that used to hand-assemble strings became this:

```
fun createQuery(request: WorkOrderSearchRequest): SoqlQuery =
    soql {
        select(FIELDS)
        from("WorkOrder")
        where { "Id" isIn request.workOrderIds }
        and { "Case.Id" isIn request.caseIds }
        and { "Status" notIn if (request.onlyOpen == true) CLOSED\_STATUSES else null }
        and { "Asset.Name" isIn request.assetIds }
    }
```

That change did three things at once:

- It made the query readable. You could look at the code and immediately see the SOQL shape.
- It made the common path consistent. Every query creator now used the same grammar: `select`, `from`, `where`, `and`, `or`, `groupBy`, `orderBy`.
- It moved correctness into the abstraction. Empty filters were skipped consistently. Grouping logic was explicit. Query validation happened at the query object boundary instead of being left to whoever happened to be writing string concatenation that day.

Coming up with the builder turned out to be the easy part. Getting broad adoption quickly became the next challenge.

When I brought the new builder to senior and staff engineers across teams, one concern was surfaced: "How do we know this won't subtly change behavior in production?" My answer was straightforward: use tests to prove it. I expanded test coverage until we had 100% query use case coverage, which gave the team the confidence they needed in the correctness of the new builder.

The final hurdle was migrating existing queries over to it. Rather than distributing that work across teams and waiting six months for it to trickle through, I decided to do the migrations myself. I had the time, and I knew that keeping the project's momentum was critical to success. Over the following two months, I handled the migration work myself. I used code gen tools to accelerate the repetitive rewrites, then reviewed the generated changes against the pinned query outputs.

In the end, it felt great to not have to ask a dozen engineers to drop what they were doing and clean up their queries. Instead, I brought them a PR where the heavy lifting was done, the changes were well-validated, and the review surface was clear and focused. It made it really hard for anyone to push back.

One thing I'm particularly proud of is that the builder was designed to support exactly the use cases we had at the time, yet it was straightforward for other engineers to extend when new needs emerged. For example:

- One engineer added null-check operators like `isNull()` and `isNotNull()`
- Another introduced upper-bound date filtering with `lessThanOrEqual()`

Neither required reworking the core they just slotted in cleanly with a couple of lines of code. That was not a side effect. That was the point. I did not want a library only I could safely change.

Result

The migration landed with zero production bugs.

After that:

- SOQL work got faster. The average task dropped from roughly 3 story points to 2.
- Code reviews got faster because reviewers were checking a standard pattern, not reverse-engineering custom string logic.
- Onboarding got easier because there was one way to write queries, not ten.
- Query-related defects dropped because the builder removed whole categories of assembly mistakes.
- The builder kept growing because other engineers could extend it without redesigning it.

The biggest win was not prettier Kotlin.

The biggest win was that I turned a messy, team-by-team problem into a standardized platform capability with near-zero adoption cost for everybody else.

Learning

If the problem shows up across teams, standardization is the leverage. I was not optimizing one query. I was removing repeated decision-making from an entire part of the codebase.

Coming up with the abstraction is only half the job. Staff-level work is not just "design something good." It is also "get the organization to trust it."

Buy-in gets easier when you can make the risk concrete. In this case, that meant proving output equivalence with tests instead of arguing from taste.

Code does not automatically create adoption. Sometimes the fastest way to get adoption is to absorb the migration cost yourself so everyone else gets the improvement for free.

Good abstractions reduce thinking, not just lines of code. The builder worked because engineers no longer had to think about spacing, empty clauses, or composition rules every time they touched SOQL.

Leave extension points obvious. The strongest signal that the design worked was that other engineers extended it later without needing me in the loop.

Deep Dive into the Implementation

The core strategy was straightforward: represent the query as an **Abstract Syntax Tree (AST)** that a serializer then transforms into a valid SOQL string.

```
class QueryBuilder {
    private val selectFields = mutableListOf<String>()
    private var fromObject: String? = null
    private val conditions = mutableListOf<Condition>()
    private var limitCount: Int? = null

    fun build(): Query {
        // Enforce structural integrity at the boundary
        require(selectFields.isNotEmpty()) { "SELECT clause is required" }
        require(fromObject != null) { "FROM clause is required" }

        return Query(
            listOfNotNull(
                "SELECT ${selectFields.joinToString(", ")}",
                "FROM $fromObject",
                   conditions.takeIf { it.isNotEmpty() }?.joinToString(" ", prefix = "WHERE ") {
it.render() },
            ).joinToString(" "),
        )
    }
}
```

Crafting the DSL

To make this builder feel like a native part of the language rather than a "utility class," I leveraged three specific Kotlin features to implement a clean, declarative API.

1. Function Literals with Receiver

The `soql { ... }` and `where { ... }` blocks work because the lambdas execute with the builder as the **receiver** (`this`). This provides a compact API that feels like a configuration block, removing the need to pass builder objects around manually.

2. Extension Functions and Infix Notation

The ability to add **extension functions on** `String`, allowed me to use field names as the entry

point for predicates. Combined with **infix functions**, the DSL reads like a natural sentence:

```
soql {
    select(FIELDS)
    from("ObjectName")
    where { "Id" isIn request.caseIds }
    and { "Status" isIn request.statuses }
    and { "CreatedDate" greaterThanOrEqual request.createdSince }
}
```

Where the Safety Actually Lives

I want to be clear: I wouldn't claim that "the type system makes malformed SOQL impossible." We still used strings for field names to maintain compatibility with our existing codebase.

However, this approach successfully neutralized the "death by a thousand cuts" common in manual query building:

- **Structural Requirements:** You literally cannot call `.build()` without a `select` and a `from`.
- **Consistency:** Empty filters are skipped automatically and consistently.
- **Logic Isolation:** Boolean grouping is explicit, replacing the "wild west" of manual string concatenation.
- **Formatting Single-Source-of-Truth:** Quoting, escaping, and `NULL` formatting live in the `Condition` class, not scattered across the service layer.