Kafka Streams DSL Grammar

Table of Contents

Context

The Streams DSL (Domain Specific Language) is what's known as an "Embedded DSL". This means that when you program in the Streams DSL, you're not writing source code in a separate language with its own parser and runtime. Instead, you write the code by making library calls in a "host" language. In our case, the host language is Java (and Scala). This carries with it some tradeoffs, but mostly benefits in our case.

One downside of embedded DSLs, though, it that they make it a little too easy for the authors to gloss over defining the grammar of the language. Maybe this is a little unfair, since not all parsers are grammar-based, but it seems like the act of writing a language parser puts you more in the frame of mind to think about the structure of your language than writing interfaces and method headers does.

For small, one-off DSLs, a defined grammar probably isn't that important. Users can easily understand the whole extent of the language at once, even if they occasionally have to look at the docs. But for a large and complex language, with lots of objects and operations, having a defined grammar is incredibly beneficial for both users and maintainers. From the user's side, using a programming language with a compact grammar is *far* easier than one where each statement seemingly follows its own rules. It's harder to predict in advance how you can make certain statements, and users will constantly be facing compiler errors and referring to the docs to figure out where to put which arguments on which statements. As an analogy, imagine trying to have a conversation in which you have to adhere to completely different grammatical rules, depending on the subject of the sentence!

As maintainers, there are other benefits of sticking to a grammar. We can avoid debating whether we should add new overloads or not, counting how many new method signatures a change would create, debating over the names of config objects, etc. We also avoid a constant thrash of adding new overloads and deprecating old ones, which increases our maintenance surface area.

This is why we define a Streams DSL grammar and coerce our API to match it.

The grammar is informally specified on this page and should serve as a roadmap for future modifications to the DSL. Creating a formal specification is out of scope right now; we need experience to decide if it would be useful. Also, making a pass over the whole DSL to coerce it to comply with this grammar is left as an available task for whoever is interested (and would require KIPs).

This is a living document. As we live with the grammar over time, we will discover shortcomings of the current specification, and we would update the spec as needed.

Grammar Specification

DSL Structure

1. There are DSLObjects and DSLOperations.
    a. Top-level DSLObjects: KTable, KStream, GlobalKTable
    b. examples of other DSLObjects: TimeWindowedKStream, KGroupedTable
    c. examples of DSLOperations include filter, mapValues, aggregate, join, etc.
2. `StreamsBuilder` is the entry point, and includes methods to produce the top-level DSLObjects
3. DSLObjects offer DSLOperations via method chaining
4. DSLOperations methods are either terminal or produce new DSLObjects

Grammatical Rules

1. Operations take either zero or more arguments, consisting of **at most one Parameter argument** (defined below) and **zero or more DSLObject arguments** (depending on the arity of the operation).
    - syntactically:
        - **`operand: a DSLObject`**
        - **`operation: operand.operator(Parameter?) | operand.operator(operand, Parameter?) | operand.operator(operand collection, Parameter?)`**
    - rationale: DSLOperations either need no parameters at all, or they need some parameters. If they need some parameters, history teaches us that there will be nuances, and we open the door to additions and adjustments to the parameters in the future. By isolating all this stuff into a single Parameter object, we prevent a proliferation of overloads, deprecations, etc. that are inevitable when we start adding different parameters directly to the argument list. Also, if you think about writing a syntax rule for Streams operations, as I've done above: without this restriction, each operation would have its own special syntax, which is a huge language-design antipattern.
2. If present, the Parameter argument uses the following builder pattern:
    a. Must be named `{DSLOperation}Parameters` (e.g., `FilterParameters`, `FlatTransformValuesParameters`, `ToParameter`)
        - rationale: This circumvents naming collisions where we try to generalize, something like "StatefulParameters", only to find later on that some new operator doesn't fit cleanly into the categories we thought were adequate. It also circumvents sensibility conflicts in which we come up with a more "natural" pattern, like using the past participle of the operation (as today with Joined,

Grouped, etc.), only to find that some of our operation names don't *have* a past participle, like `through`. Plus, the past participle strategy isn't machine verifiable.

b. Must be namespaced by a package corresponding to the DSLObject the operation is on. (e.g., `ktable.FilterParameters` vs `kstream.FilterParameters`)
    - rationale: As the example demonstrates, this prevents parameter name collisions between the DSLObjects, which we could otherwise only resolve by namespacing the parameter class itself (like `TableFilterParameters`), which is more verbose, and also opens a pandora's box where we might try to namespace only the operations that have collisions (like, we can just use `ToStreamParameters` because only the KTable API as this operation), but then later on want to add a new operation to one of the DSLObjects and actually create a collision (like maybe we decide to add `toStream` to the GlobalKTable API), but now it's worse because one of them is like `GlobalKTableToStreamParameters` and the other is just `ToStreamParameters`, which we just have to know only applies to the KTable API... in short, it would be a mess.

c. Must be a "struct"-style class, aka a simple data container, with only private final fields, getters, and setters. No extra methods implementing special logic.
    - rationale: This allows us to build a clean separation between the "source code" of the program and the actual logic (the "compiled version") of it. Adhering to this separation is a key to long term maintainability. Note that in non-embedded DSLs, where the source code is just literally a text file, it's not possible to violate this rule. Embedded DSL authors have to have a little extra discipline.

d. Must be a final class, not extending superclasses, but potentially implementing interfaces.
    - rationale: Gives us a high degree of confidence that the Parameters objects are exactly what they seem, and we don't have to worry about what subclassing would do to their semantics. We've seen that the intentional subclassing in the `Windows` ecosystem results in unintentional rigidity, which we now cannot remove: the worst kind of tech debt, because it can't be cleaned up. It also prevents *us* from trying to be clever over time and combine similar objects into complicated hierarchies, which only come back to bite us when we realize that the abstractions are leaking.

e. Instantiation via static factory methods:
    i. if no required arguments, then the method is a camel case version of the parameters class name. E.g., `filterParameters()`, `flatTransformValuesParameters()`, etc.
        - rationale: Consistent, compact, and guaranteed to be meaningful for all parameters. Also, the methods can be statically imported without collisions.

    ii. if there are required arguments, the method must begin with `from` and contain a description of the arguments, like `fromSerdes(key serde, value serde)`
        - rationale: Consistent and compact. Provides informative context (i.e., you don't have to reason about the relative meanings of five methods all named "`as`" but different argument lists). Also, guaranteed not to conflict with the optional parameters (below)

    iii. if there are multiple alternative construction strategies, then multiple of the above formulations are ok
        - rationale: just stating the obvious...

f. Augmenting via instance methods:
    i. Optional parameters are provided using builder-pattern instance methods. These methods must begin with `with` and contain a description of the arguments, like `withKeySerde(key serde)` or `withName(name)`
        - rationale: Consistent and compact. Provides informative context (i.e., you don't have to reason about the relative meanings of five methods all named "`with`" but different argument lists). Also, guaranteed not to conflict with the "reqired arguments" version.

    ii. Optional parameters that just toggle a flag can be named `enable{FlagName}` or `disable{FlagName}`, but cannot take any arguments
        - rationale: Basically just a concession to avoid boolean flags as arguments, like `withLogging(false)`.

g. Getters: All arguments, whether from the factory or the builder methods, should be available by getters
    - rationale: This makes the Parameters objects transparent, so users can inspect their own parameters (this is a stumbling point that our current setters-only policy creates for some users). Plus it benefits maintainers, since we no longer have to have an "internal" doppelganger for every. single. config. Finally, it's necessary in conjunction with the "final class" rule.