

Third-party library exposure in four open-source Java projects, measured and reduced with CodeLaser

All four projects were parsed in full and every written reference from any of their own source files to a third-party type was recorded: declarations, calls, constructions, inheritance, annotations, casts and type tests. Main and test sources alike, because taking a library out means editing every file that writes its name. The counts are what the engine would change. On all four projects the change was also carried out: 28 libraries were put behind an interface, and every result was compiled and its tests run.

ISSUED	MEASURED	CONTACT
2026-09-24	2026-09-24	tom.tourwe@codelaser.io

IN SHORT

What the four projects show

- **What was measured.** Every reference from a project's own source to a third-party type, in four open-source Java projects, and for each one whether a tool can change it or a developer has to decide. Caffeine: 49 libraries, 26,235 references. Timefold Solver: 99, 42,294. Trino: 303, 361,438. QuestDB: 26, 191,332.
- **What was done.** On all four projects the engine then put libraries behind a project-owned interface, one library at a time: 28 attempted, 28 compiled, and not one broke a test that had been passing.
- **Every count is a resolved rewrite.** The edit figures are what the engine would change, worked out against the model rather than estimated from the reference count. A library can have more references than edits or more edits than references, and this report explains why.
- **What an interface costs to own.** Every library method the project calls becomes a method the project maintains. The last column of every table divides references freed by methods taken on.
- **Where the manual work is.** Annotations and inheritance. In Timefold, jspecify accounts for 1,431 references across 155 packages and none of them can be changed by a tool.
- **What the figures do not say.** Whether the change is worth making. slf4j is among the easiest libraries in Timefold to extract, and this report explains why nobody should.

WHAT WAS MEASURED

Exposure, and what reducing it means

Exposure to a library is the set of places in a project's own source that write one of that library's names. Reducing it means putting an interface of the project's own in front of the library, so that only a bounded part of the codebase writes the library's names and the rest writes the project's.

For each library, the project tables report how many of those places a tool can change on its own and how many need a developer, together with the number of library methods an interface would have to carry. "What these figures do not cover" states the limit. The two sections after it define the two classes of reference and the three conditions under which a declaration cannot be changed.

THE SETUP

Four things produced these figures

Four things produced the figures in this report: the four project sources, the CodeLaser engine, the census operation it exposes, and the record the operation wrote.

The project sources

Caffeine at `07c6e370c`, Timefold Solver at `290c87fc53`, Trino at `27e3b9c8d62` and QuestDB at `714b750076`, each taken from its public repository and built with its own build files, so that the compile classpath is the one the project itself resolves. Each was worked on in a sandbox of its own and reset to its pinned commit; the public repositories were never written to.

The CodeLaser engine

A refactoring and modernization engine. It reads the whole project once and builds a model of it: every type and member, every reference from one to another, the source set each belongs to, and the classpath behind them. "What the extraction rests on" says what that model has to hold for this measurement to be possible at all.

The census operation

`extract.libraryCensus` walks every reference in the model, decides for each one which class-path jar it lands in, and works out what putting that jar behind an interface of the project's own would take. **This report counts all of a project's own source, main and test alike**, because the engine rewrites both; the table below shows how much of the work is in the test sources.

The record

The engine writes one record per project holding every count in this report, so that any figure here can be traced to the run that produced it. The record is internal; we will walk anyone evaluating CodeLaser through it. The four projects were measured on 24 September 2026.

Why test sources are counted

Every figure here counts all of a project’s own source, main and test alike. Taking a library out means editing every file that writes its name, and the engine rewrites test sources as well, so a count that left them out would understate the work by the difference below — and would not show where the library is really used:

PROJECT	ALL SOURCES	MAIN SOURCES ONLY
Caffeine	26,235	3,782
Timefold Solver	42,294	8,938
QuestDB	191,332	94,185
Trino	361,438	137,390

A MAIN MODULE MAY STILL BE ABOUT TESTING

The split above is by source set, not by what the code is for. Trino publishes `trino-testing`, `trino-testing-containers` and `trino-testing-services` as artifacts other projects compile against, so its references to JUnit, AssertJ and Testcontainers fall on the main side of that table. QuestDB’s benchmark harness is a `benchmarks` module of its own, and every one of its references to JMH does too. Either way both sides are counted.

THE TOOL

What the extraction rests on

A text search for a library’s name finds strings. It cannot say which of them are references, what each reference does, or whether changing one is safe. Four properties of the model are what let the engine rewrite them instead, and count what it rewrote.

Every name is resolved

The engine resolves each name the way a compiler does, so it knows which declaration a call reaches and which type a name denotes. That is what makes a reference attributable to one jar rather than to a word that happens to match.

The classpath is part of the model

A call into a library resolves to a real declaration, read from that library's bytecode. Without it, a library method and a project method of the same name are indistinguishable, and the per-library counts could not be separated at all.

Source sets are part of the model

Every type is identified by its source set as well as its name. That is what allows main and test code to be told apart, and it is why the table above can exist.

Each reference carries what happens to it

For every reference the model holds the kind of position it sits in — return type, field type, annotation, construction, cast — and, for the ones that cannot be changed, the reason. The two sections that follow are that classification written out. The edit count is arrived at the same way, one rewrite at a time, which is why it does not match the count of references.

It refuses rather than guesses

Where the engine cannot show a rewrite is correct it declines it and names the reason, and the reason is in the tables. That is what makes a planned edit worth counting: what a person would have had to decide is never counted as something the tool can do.

THREE COUNTS

A reference, a planned edit, an actual edit

Three counts appear in this report, and they are not interchangeable.

A reference is a place the project's own source writes one of a library's names. The tables count these per library.

An edit is a place the engine would change characters to put the library behind an interface. It is computed by resolving the rewrite against the model, without touching the project. Every project in this report has these figures.

An actual edit is a line a run really rewrote, on a copy of the project, followed by a compile and a test run. All four projects have these, for the libraries named in *What was carried out*.

References and edits do not correspond one to one. javapoet in Caffeine has 703 references and 320 edits; cache-api has 1,649 references and 1,467 edits. Neither count stands in for the other, and one method from Caffeine's own code generator shows why.

Why most references need no edit

This method is in Caffeine's code generator, in `AddValue.java`. The names belonging to javapoet are the type `MethodSpec` and the methods `methodBuilder`, `addModifiers`, `returns`, `addStatement` and `build`. `NodeContext`, `vTypeVar` and `handle` are Caffeine's own.

```
private static MethodSpec makeGetValue(NodeContext context) {
    var getter = MethodSpec.methodBuilder("getValue")
        .addModifiers(context.publicFinalModifiers())
        .returns(vTypeVar);
    String handle = varHandleName("value");
    ...
    getter.addStatement("return ($T) $L.getAcquire(this)", vTypeVar, handle);
    return getter.build();
}
```

Putting javapoet behind an interface changes **two** of those lines. Here they are, before and after; every other line in the method is left exactly as written.

```
before: private static MethodSpec makeGetValue(NodeContext context) {
after:   private static IMethodSpec makeGetValue(NodeContext context) {

before:     var getter = MethodSpec.methodBuilder("getValue")
after:     var getter = MethodSpecs.methodBuilder("getValue")
```

The return type becomes the generated interface. The static call is sent to the generated factory, `MethodSpecs`.

The four calls on the `getter` object — `addModifiers`, `returns`, `addStatement` and `build` — are untouched, because the generated interface declares all four with the same names and the same parameters. Each of those lines is still a reference, since the project reaches javapoet there, and each costs nothing to change.

That is the single biggest reason the two counts differ. Across the whole of Caffeine, **472 of javapoet's 703 references are calls on a javapoet object**, and all 472 are free.

Why a library can need more edits than it has references

cache-api runs the other way: **1,649 references, 1,467 edits**. Its mixture is nothing like javapoet's. 575 of its references are calls on a library object, the free kind. Against that, 208 are constructions — each becomes a factory call *and* an added `import`, so two edits for one reference — 54 are `extends` or `implements`, 141 are reads of a static field, 109 are class literals, and the rest are casts, type tests and method references.

Some of its edits also land on lines that write no cache-api name at all, and so were never counted as refer-

ences. In outline, illustrative rather than taken from Caffeine:

```
// the project's own method; its return type names Cache, so the tool retypes it
private ICache<K, V> build(String name) { ... }

// elsewhere; register is cache-api's method and still wants cache-api's own Cache.
// This line writes no cache-api name, so it was never counted as a reference --
// but it now needs one, to hand the library back its own object:
manager.register(build("users").wrapped());
//                               ^^^^^^^^^^^ added
```

What follows from all this. A library the project mostly *calls* shows many references and few edits. A library the project *constructs, extends and passes around* shows fewer references and more edits. Use the reference count to judge how widely the library has spread through the codebase, and the edit count to judge the size of the change itself. Neither count stands in for the other.

READING THE TABLES

What each column counts

References freed per method is the price of the trade

Putting a library behind a boundary of the project's own means writing — and then maintaining for as long as the project lives — a replacement for every method the project calls on it. Call 29 different methods of a library, and the project owns 29 method signatures of its own.

That is the price. The benefit is the number of references that stop naming the library. The last column divides one by the other: **how many references are freed for each method taken on**.

- **A high number is a good trade.** typesafe-config frees 473 references for 30 methods — about 16 references per method owned.
- **A low number is a poor one.** guava frees 2,044 references, but the project calls 293 different guava methods across 94 guava classes: 7 per method.
- **Around 1 is no trade at all.** jfreechart frees 77 references for 50 methods.

guava is among the most referenced of Caffeine's dependencies and 92% of them are changeable, which makes it look like an obvious candidate. This column is what says otherwise: guava is used broadly rather than deeply, and breadth is what the project would pay for.

Libraries used only through annotations call no methods at all, so the column is empty for them; nothing can be taken on, and nothing is freed.

WHAT WAS CARRIED OUT

28 libraries put behind an interface, on all four projects

`extract.abstractLibrary` generates the interface and the adapters, routes constructions and static calls through a generated factory, and retypes every declaration that named the library. It was run on all four projects, one library at a time, each attempt from a pristine copy, compiled, and its tests run before and after the change:

PROJECT		LIBRARIES ATTEMPTED	CAME OUT CLEAN
Caffeine		4	4
Timefold Solver		8	8
QuestDB		9	9
Trino		7	7

PROJECT	LIBRARY	GENERATED	EDITED	OUTCOME
Caffeine	fastutil-core	54	40	tests ran and passed
	jctools-core	23	4	compiled; tests failed to run within a two-hour cap
	cache2k-api	15	1	tests ran and passed
	stream	5	1	tests ran and passed
Timefold Solver	gizmo	42	14	tests ran and passed
	maven-plugin-testing-harness	0	1	tests ran and passed

PROJECT	LIBRARY	GENERATED	EDITED	OUTCOME
	jboss-logging	5	9	tests ran and passed
	spring-boot	5	4	tests ran and passed
	vertx-core	8	2	compiled; there are no tests to run
	maven-model	4	2	tests ran and passed
	jcl-over-slf4j	5	3	tests ran and passed
	xml-path	3	2	tests ran and passed
QuestDB	parquet-hadoop	12	2	tests ran and passed
	influxdb-java	22	5	tests ran and passed
	okhttp	36	1	tests ran and passed
	simpleclient	24	3	compiled; there are no tests to run
	parquet-common	8	2	tests ran and passed
	influxdb-client-java	7	2	tests ran and passed

PROJECT	LIBRARY	GENERATED	EDITED	OUTCOME
Trino	gson	7	2	tests ran and passed, 3 already failing
	log4j-api	5	2	compiled; there are no tests to run
	reactive-streams	5	1	tests ran and passed
	google-api-services-sheets-v4	43	2	tests ran and passed
	google-cloud-bigquerystorage	22	12	tests ran and passed
	clickhouse-client	5	3	tests ran and passed
	google-http-client	29	2	tests ran and passed
	json	8	3	tests ran and passed
	datafaker	8	4	tests ran and passed, 1 already failing
	ulidj	4	2	tests ran and passed

generated is the number of interfaces and adapters the engine wrote; **edited** is the number of the project's own files it changed. **All twenty-eight compiled, and not one broke a test that had been passing.** Four carry a qualifier and it is always about the project, never the change: three touch modules that carry no test source, so the compile is the whole of what could be checked, and one is a suite that did not finish

inside a two-hour cap. Two projects have tests that were already failing before anything was touched — three in QuestDB's `utils`, one in Trino's `trino-faker` — and the same ones fail afterwards.

WHICH LIBRARIES ARE ELIGIBLE AT ALL

Three things stop a library being extracted, and the tables report all three. A library that needs a developer cannot be finished without one. A library whose type no declaration anywhere writes down — not a field, not a parameter, not a return type — has nothing to retype and nowhere to put an interface, however many of its objects the project creates. And a library used from source sets with no common ancestor has nowhere to put the generated package, so the operation refuses the write and says so. 68 libraries across the four projects pass all three tests. Eligible is not the same as attempted: the 28 attempts above cover every eligible library of Caffeine, Timefold Solver and QuestDB. Trino, the largest, has 47. Seven of them were run as a sample.

THE LIMIT OF THE FIGURES

What these figures do not cover

Every figure in this report measures **the size and shape of the change**: how many places there are, how many need a person, and how large an interface the project would own afterwards. That a change was made and the tests still passed says it was possible, not that it was worth making. **A library can be easy to extract without the extraction being useful, and these figures cannot tell the two apart.** They should therefore be read with care: a high changeable share and little manual work say that a tool could do the work, and nothing more. One library shows the gap.

Why slf4j is not worth extracting

On the figures this is among the least work of any library in the project: all but one of its 373 references is one a tool changes. `slf4j` is also an interface already, whose implementation is selected when the program is deployed by which other jar is placed on the class path. Changing the logging implementation therefore costs no source edit today. An interface in front of it would carry the 37 `slf4j` methods the project calls.

So extracting `slf4j` would not be worth doing. The project would take on 37 method signatures of its own, permanently, and gain a freedom to swap implementations that it already has for nothing. On the figures alone `slf4j` is among the easiest libraries in Timefold to extract. It is the last one anybody should.

Every other row in every table can be misread the same way, in either direction. The figures say what the change would cost, not what it would achieve.

CLASSIFICATION

The two classes of reference

The tables give two counts per library. The second splits into two kinds, so there are three labels below. Every project table gives both counts for every library.

Changed by the tool. **Type positions** — field, parameter, return, local and type argument — are retyped to the interface. **Instance calls** are left exactly as written: the interface declares them. **Constructions and static calls** that yield a library instance are routed through a generated factory. **Casts** are rewritten where the value being cast provably comes from the interface.

Needs a developer. Cannot be changed. The class extends or implements a library type; or the reference is a library annotation; or it is a written type in a declaration that may not be changed, for one of the three reasons that follow. Java permits one superclass, and an annotation type has no interface equivalent, so those two are permanent.

Judgement. `instanceof` tests and casts where the analysis cannot determine which type the value will actually hold at that point; class literals passed to reflection or a registry; instance field reads; method references. Each can be rewritten, but not provably correctly, so the choice is made per site.

REFUSALS

Why a place may not be changed

A **declaration** is one place where the project writes a library type down: the type of a field, the type a method returns, or the type of a method's parameter. Those are the places a project-owned interface would be written instead. A local variable inside a method body is not one of them: no code outside the method can see a local variable's type, so it always follows whatever value it is given and is always free to change.

Three things stop a declaration from being changed. They do not cost the same, and only the second is caught by the Java compiler.

First reason: a library reads the declared type while the program is running

Timefold's service layer uses two libraries that pass events between parts of a running program: CDI, which supplies the `@Inject` annotation, and SmallRye Reactive Messaging, which supplies `@Channel`. This field is written in `AbstractModelAPIResource.java`:

```
@Inject
@Channel(SolverChannels.DATASET_EVENTS)
Multi<Metadata<Score_>> datasetLifeCycleEvents;
```

When the program starts, those libraries search the compiled code for fields marked this way and put a value into each one. What they put in is decided by the type the field declares — here `Multi`, a type belonging to a third library, Mutiny. Change this field to a project-owned interface and the search matches nothing: the field is never filled in, and the program fails the first time it reads that field.

Nothing in the Java language connects an annotation to a search carried out at startup, so **the compiler cannot detect this**. The project builds and then misbehaves when run, which is why the analysis refuses such a declaration unless a person says otherwise. It records which annotation caused each refusal, because that decides whether the refusal is real: an annotation whose library reads the declared type does block the change, while an annotation that merely records something about a value — that it must not be null, say — does not.

Second reason: Java requires the types the library declared

SmallRye Reactive Messaging publishes an interface named `MutinyEmitter`. One of the methods that interface declares is `Uni<Void> send(T payload)`, where `Uni` is another Mutiny type. Timefold writes a class that implements that interface:

```
public final class RecordingMutinyEmitter<T> implements MutinyEmitter<T> {  
    @Override  
    public Uni<Void> send(T payload) { ... }  
}
```

Java requires a class that implements an interface to declare each method with exactly the types the interface declared. `MutinyEmitter` declares that `send` returns a `Uni`, so this class's `send` must return a `Uni`. Put a project-owned interface there instead and the class no longer implements `MutinyEmitter`. The compiler reports exactly that:

```
error: RecordingMutinyEmitter is not abstract and does not override  
      abstract method send(T) in MutinyEmitter  
error: send(T) in RecordingMutinyEmitter cannot implement send(T) in MutinyEmitter  
      return type IUni<Void> is not compatible with Uni<Void>
```

No option or setting permits this, because it is a rule of the Java language rather than a choice this analysis makes. Such a declaration can change only if the class stops implementing the library's interface altogether, which is a different and larger piece of work.

A POSITION THE LIBRARY LEFT OPEN

Not every type written next to a library's name belongs to the library. Look at the `<T>` in `MutinyEmitter<T>` above. `T` is a **type parameter**: a placeholder the library writes in place of a real type name, meaning "this interface works with some type, and whoever implements it decides which". The library never names that type, so nothing the library does depends on it.

Compare two interfaces a library might publish. The first names a type of its own; the second writes a placeholder:

```
interface Visitor { void visit(Money m); } // Money is the library's own type
interface Sink<T> { void accept(T t); }    // T is a placeholder, filled in by the implementer
```

A class implementing `Visitor` must write `Money`, because the library wrote `Money`. That is the second reason above, and the declaration is stuck. A class implementing `Sink` chooses for itself what stands in for `T`. So if the project wrote `class Prices implements Sink<Money>`, the word `Money` on that line is the project's own choice, not the library's, and it may be replaced: `Sink<IMoney>` still implements the library's interface, and the compiler accepts it. Such a declaration is *not* refused.

Third reason: the library calls back with the type the project filled in

This is the case just described, the project choosing what stands in for a placeholder, with one addition: the project hands its object to the library, and the library then calls back using the very type that was filled in. Trino writes this class in `trino-testing-containers`, one of the testing-support libraries it publishes. It prints the log output of a container:

```
public final class PrintingLogConsumer extends BaseConsumer<PrintingLogConsumer>
{
    @Override
    public void accept(OutputFrame outputFrame)
    { ... }
}
```

Both `BaseConsumer` and `OutputFrame` belong to Testcontainers, the library Trino uses to start containers. Trino registers a `PrintingLogConsumer` with that library; the library then calls `accept` itself, passing an `OutputFrame` object that the library created. So the type of that parameter is not a choice Trino can make: it is the type of the object the library will hand over. Writing a project-owned interface there produces a method the library never calls, because the library calls the method that accepts an `OutputFrame`.

The same situation arises when a library searches for a class to use rather than calling it directly. A JSON library asks which of the classes registered with it handles values of one particular type; a class registered as handling a project-owned interface is not found when a value of the library's own type has to be written, so the library falls back to its default and produces different output, again without the compiler objecting. This is the rarest of the three reasons: seven places in Timefold, two in Trino, one in Caffeine, none in QuestDB.

PROJECT

Caffeine

Caching library for Java. 49 dependencies referenced from its own source, 26,235 references.

“What each column counts” explains the columns. The other project tables use the same ones.

Ten of the 49 dependencies: the five with the least manual work and the five with the most

Sorted by the share of references that can be changed without a developer, and where that share is equal, by how many references are freed, so the libraries needing the least manual work stand at the top.

		CHANGED BY THE TOOL	NEEDS A DEVELOPER	INTERFACE THE PROJECT WOULD OWN	
LIBRARY	REFERENCES	REFERENCES	COUNT	LIBRARY METHODS	REFERENCES FREED PER METHOD
javapoet	703	703	0	59	11.9
awaitility	360	360	0	10	36.0
fastutil-core	357	357	0	33	10.8
ehcache	20	20	0	15	1.3
ascii-table	18	18	0	7	2.6
...					
cache-api	1,649	1,353	296	117	11.6
error_prone_annotations	310	0	310	0	—
jspecify	512	0	512	0	—
junit-jupiter-api	2,030	939	1,091	22	42.7
junit-jupiter-params	2,424	108	2,316	6	18.0

Dependencies with no references

NONE

All 49 dependencies on the compile path are named at least once in the project’s own source.

Observations

- **javapoet: 703 references, none requiring a developer.** Confined to 3 packages, with no manual work attached.
- **typesafe-config: 532 references, 59 requiring a developer,** across 33 packages. 9 library classes and 30 library methods, a small surface for a dependency of this size.
- **cache-api: 1,649 references, 296 requiring a developer:** 161 judgement calls and 135 references in declarations that cannot be retyped, because this project implements the library's own interfaces and the library wrote those signatures. Spread over 12 packages: the highest manual cost in the project.
- **guava: 2,212 references, 2,044 of them changeable,** but 94 library classes and 293 library methods. Low manual cost, largest abstraction surface in the project.
- **jspecify and error_prone: 822 references, every one requiring a developer.** Annotation-only, spanning 83 packages each. No part of this is mechanical.

PROJECT

Timefold Solver

99 referenced libraries, 42,294 references.

Ten of the 99 dependencies: the five with the least manual work and the five with the most

Columns as described above. Sorted by the share of references that can be changed without a developer, and where that share is equal, by how many references are freed, so the libraries needing the least manual work stand at the top.

		CHANGED BY THE TOOL	NEEDS A DEVELOPER	INTERFACE THE PROJECT WOULD OWN	
LIBRARY	REFERENCES	REFERENCES	COUNT	LIBRARY METHODS	REFERENCES FREED PER METHOD
rest-assured	80	80	0	22	3.6
jboss-logging	43	43	0	7	6.1
vertx-core	24	24	0	2	12.0
arc-processor	21	21	0	8	2.6
smallrye-open-api-core	19	19	0	6	3.2
...					
jackson-databind	1,526	1,310	216	80	16.4
jakarta.xml.bind-api	403	56	347	17	3.3
jakarta.ws.rs-api	514	140	374	20	7.0
jspecify	1,431	0	1,431	0	—
junit-jupiter-api	4,807	371	4,436	20	18.6

Dependencies with no references

NONE

All 99 dependencies on the compile path are named at least once in the project's own source.

Observations

- **jspecify**: 1,431 references, every one requiring a developer, across 155 packages. Annotation-only. The largest single block of manual work in this project, and none of it is reducible.
- **jandex**: 1,195 references, 23 requiring a developer, across 7 packages. 23 library classes, 103 library methods.
- **slf4j**: 373 references, one requiring a developer, though spread over 51 packages. 5 library classes, 37 library methods.

- **jackson-databind: 1,526 references, 216 requiring a developer**, across 35 packages: 108 judgement calls and 108 references in declarations that cannot be retyped. It hits all three reasons at once: signatures the library wrote, framework annotations, and places where Jackson finds the class by the type written in it.
- **jakarta.ws.rs and microprofile-openapi: 1,147 references, 553 requiring a developer**, most of them annotations that cannot be changed.

PROJECT

Trino

Distributed SQL query engine for Java. 304 dependencies on the compile path, 303 of them referenced, 361,438 references.

Ten of the 303 dependencies: the five with the least manual work and the five with the most

Columns as described above. Sorted by the share of references that can be changed without a developer, and where that share is equal, by how many references are freed, so the libraries needing the least manual work stand at the top.

A row can show no changeable references and still show edits. Those libraries are reached only by `implements` clauses and annotations, neither of which can be retyped; the edits are the imports and factory calls the rest of the extraction would still make, on lines that write no library name.

		CHANGED BY THE TOOL	NEEDS A DEVELOPER	INTERFACE THE PROJECT WOULD OWN	
LIBRARY	REFERENCES	REFERENCES	COUNT	LIBRARY METHODS	REFERENCES FREED PER METHOD
configuration-testing	609	609	0	5	121.8
testing	545	545	0	12	45.4
log-manager	238	238	0	5	47.6
java-driver-query-builder	139	139	0	17	8.2
commons-math3	132	132	0	21	6.3
...					
jmh-core	2,270	164	2,106	26	6.3
configuration	3,657	933	2,724	18	51.8
antlr4-runtime	19,155	14,211	4,944	115	123.6
jackson-annotations	4,949	3	4,946	1	3.0
junit-jupiter-api	23,202	222	22,980	23	9.7

Dependencies with no references

1 OF 304 DEPENDENCIES IS NEVER NAMED IN THE PROJECT'S OWN SOURCE

`jetty-io-12.1.12.jar`: 0 references, 0 files, 0 packages. On the compile path, with no source file of the project writing any of its type names. Removal candidate, subject to a check that nothing reaches it by reflection or class loading.

Observations

- **Ten libraries account for most of the exposure.** Of the 250, the ten most-referenced are named in 97,038 of the 137,390 places, guava alone in 36,816. At the other end, 76 are named ten times or fewer.
- **guava: 36,816 references, 1,059 requiring a developer.** In 3,609 files across 402 of Trino's 471 packages. 189 library classes and 755 library methods, the largest surface any library has in these four projects, and 67,087 edits.
- **antlr4-runtime: 19,153 references, 4,937 requiring a developer** — the most manual work of any library in the project. It is not spread out: 11 packages, 50 files. 1,469 of those references sit in signatures ANTLR itself declared, and there are 214 places where Trino extends an ANTLR type.
- **jackson-annotations: 4,876 references, 4,873 requiring a developer**, across 115 packages. Annotation-only, so almost none of it is mechanical.
- **configuration and guice: 9,408 references, 4,252 requiring a developer.** Both bind declarations at run time rather than at compile time. Every one of the 203 packages that names guice contains manual work.

PROJECT

QuestDB

Time-series database for Java. 27 dependencies on the compile path, 26 of them referenced, 191,332 references.

Ten of the 26 dependencies: the five with the least manual work and the five with the most

Columns as described above. Sorted by the share of references that can be changed without a developer, and where that share is equal, by how many references are freed, so the libraries needing the least manual work stand at the top.

A row can show no changeable references and still show edits. Those libraries are reached only by `implements` clauses and annotations, neither of which can be retyped; the edits are the imports and factory calls the rest of the extraction would still make, on lines that write no library name.

		CHANGED BY THE TOOL	NEEDS A DEVELOPER	INTERFACE THE PROJECT WOULD OWN	
LIBRARY	REFERENCES	REFERENCES	COUNT	LIBRARY METHODS	REFERENCES FREED PER METHOD
parquet-hadoop	188	188	0	13	14.5
influxdb-java	128	128	0	30	4.3
simpleclient	45	45	0	13	3.5
parquet-common	22	22	0	0	—
reactor-test	15	15	0	4	3.8
...					
postgresql	65	45	20	14	3.2
questdb-client	12,046	11,792	254	418	28.2
annotations	6,444	0	6,444	0	—
jmh-core	87,688	52,694	34,994	46	1145.5
junit	84,146	46,104	38,042	60	768.4

Dependencies with no references

1 OF 27 DEPENDENCIES IS NEVER NAMED IN THE PROJECT'S OWN SOURCE

`okio-jvm-3.6.0.jar`: 0 references, 0 files, 0 packages. On the compile path, with no source file of the project writing any of its type names. Removal candidate, subject to a check that nothing reaches it by reflection or class loading.

The last column is meaningless for `jmh-core`: its 52,694 changeable references are overwhelmingly in benchmark code a generator wrote, so dividing them by the 46 methods the project calls says nothing about a trade anyone would make.

Observations

- **Almost none of this is in the database.** Of the seven libraries, only the annotations jar is written anywhere in `core`. Everything else is named from the benchmark harness or the utilities. Leaving those two aside, QuestDB's production code names five libraries in 411 places.
- **jmh-core: 87,688 references, 34,994 of them requiring a developer.** All of it is the benchmark harness, a module of its own. 373 of the 501 files naming JMH are generated by its annotation processor, which is why the count is what it is. This row describes the benchmarks, not the database.
- **annotations: 6,086 references, every one requiring a developer.** The only library the core names, and annotation-only, so nothing here is mechanical.
- **questdb-client: 355 references, 5 requiring a developer.** 27 library classes and 65 methods. The project's own client library, used by the benchmarks and the utilities.