AIRFLOW. SUMMIT

From JAR to Dag

Running Java tasks natively with the Airflow Java SDK.

Airflow 3.3 • experimental

org.apache.airflow:airflow-sdk:1.0.0-beta1

Jason(Zhe-You) Liu

Astronomer • AIP-108 • github.com/jason810496 • linkedin.com/in/zhe-you-liu

Why a Java SDK at all.

JVM teams that need orchestration.

Observability, retries, secrets. Without leaving the ecosystem.

Existing Java code, already in production.

Plug it in directly. No shell-out, no container wrapper, no second system.

Author the whole Dag in Java.

Zero Python. Compile-checked wiring. One language, one repo.

One orchestrator, any language. Java is the first shipped runtime.

Three ways you run Java from a Dag today.

EVERYONE HAS SHIPPED ONE

BashOperator
+ spark-submit

Kubernetes
PodOperator

A REST
microservice

Not mistakes. This is what you do when the orchestrator speaks one language.

Same four questions, four ways to run Java.

	SPARK-SUBMIT	POD OPERATOR	REST SERVICE	JAVA SDK
Logs in the task log	scraped	pod logs	✗	✓ native
XCom in and out	✗ regex	out only	✗	✓
Connection / Variable by id	✗ argv	✗	✗	✓ read
Retries owned by Airflow	✗	pod-level	✗	✓ on the stub

Java stopped being a payload. And that last column ships in 3.3 today.

01 / 05

Write a Java task, step by step

A Python Dag, a Java class, and a bundle.

WRITE IT | HOW IT RUNS | PRIMITIVES | IN FLIGHT | DEMO & CLOSE

In 3.3 a pipeline is two files.

PYTHON

`dags/sales_pipeline.py`

+

JAVA

`SalesPipeline.java`

Python declares the Dag. Java implements the tasks.

Plus Main.java and build.gradle.kts.

Declare the Java tasks as stubs.

dags/sales_pipeline.py — Airflow 3.3

```python
from airflow.sdk import dag, task

@task.stub(queue="java")
def extract(): ...

@task.stub(queue="java", retries=1)
def transform(): ...

@task
def report(v): print(v)

@dag(schedule="@daily")
def sales_pipeline():
    t = transform()
    extract() >> t
    report(t)
```

queue= picks the runtime.

dags/sales_pipeline.py — Airflow 3.3

```python
from airflow.sdk import dag, task

@task.stub(queue="java")
def extract(): ...

@task.stub(queue="java", retries=1)
def transform(): ...

@task
def report(v): print(v)

@dag(schedule="@daily")
def sales_pipeline():
    t = transform()
    extract() >> t
    report(t)
```

Wire them like any other task.

dags/sales_pipeline.py — Airflow 3.3

```python
from airflow.sdk import dag, task

@task.stub(queue="java")
def extract(): ...

@task.stub(queue="java", retries=1)
def transform(): ...

@task
def report(v): print(v)

@dag(schedule="@daily")
def sales_pipeline():
    t = transform()
    extract() >> t
    report(t)
```

Retries stay in the Dag file.

dags/sales_pipeline.py — Airflow 3.3

```python
from airflow.sdk import dag, task

@task.stub(queue="java")
def extract(): ...

@task.stub(queue="java", retries=1)
def transform(): ...

@task
def report(v): print(v)

@dag(schedule="@daily")
def sales_pipeline():
    t = transform()
    extract() >> t
    report(t)
```

Two authoring syntaxes for Java SDK.

Interface

- implements Task, registered by hand
- You get Context: the Dag run and the task instance
- Explicit, no code generation

Annotations

- A plain class with @Builder.* on it
- The processor generates the registry and the XCom fetch
- Less to write, more to know

Same runtime behaviour. Mix both in one bundle.

Syntax one: implement the interface.

FetchTask.java — interface API

```java
public class FetchTask implements Task {

  @Override
  public void execute(Context ctx, Client client) {

    ctx.ti.tryNumber;          // taskId · mapIndex
    ctx.dagRun.logicalDate;    // runId · conf

    client.setXCom(result);
  }
}
```

The current Task Instance context.

FetchTask.java — Context

```java
public class FetchTask implements Task {
  @Override
  public void execute(Context ctx, Client client) {
    ctx.dagRun.dagId;              // "sales_pipeline"
    ctx.dagRun.runId;              // this Dag run
    ctx.dagRun.logicalDate;        // data interval, runType, conf

    ctx.ti.taskId;                 // "fetch"
    ctx.ti.mapIndex;               // -1 when not mapped
    ctx.ti.tryNumber;              // 1, 2, 3 across retries
  }
}
```

Context is two objects: **dagRun** and **ti**.

The injection client reaches Airflow's models.

FetchTask.java — Client

```java
public class FetchTask implements Task {
  @Override
  public void execute(Context ctx, Client client) {
    var conn = client.getConnection("sales_db");
    var threshold = client.getVariable("threshold");
    var upstream = client.getXCom("extract");

    client.setXCom(result);
  }
}
```

Same Client the annotation API injects.

Syntax two: annotate a plain class.

SalesPipeline.java — annotation API

```java
@Builder.Dag(id = "sales_pipeline")
public class SalesPipeline {
  static final System.Logger log = System.getLogger("Sales");

  @Builder.Task(id = "extract")
  public long extract(Client client) {
    var conn = client.getConnection("sales_db");
    log.log(INFO, "reading {0}", conn.host);
    return recordCount;
  }

  @Builder.Task(id = "transform")
  public long transform(Client client,
      @Builder.XCom(task = "extract") long extracted) {
    var threshold = client.getVariable("threshold");
    return extracted * 2;
  }
}
```

Bind methods to tasks.

SalesPipeline.java — annotation API

```java
@Builder.Dag(id = "sales_pipeline")
public class SalesPipeline {
  static final System.Logger log = System.getLogger("Sales");

  @Builder.Task(id = "extract")
  public long extract(Client client) {
    var conn = client.getConnection("sales_db");
    log.log(INFO, "reading {0}", conn.host);
    return recordCount;
  }

  @Builder.Task(id = "transform")
  public long transform(Client client,
      @Builder.XCom(task = "extract") long extracted) {
    var threshold = client.getVariable("threshold");
    return extracted * 2;
  }
}
```

Inject the upstream XCom.

SalesPipeline.java — annotation API

```java
@Builder.Dag(id = "sales_pipeline")
public class SalesPipeline {
  static final System.Logger log = System.getLogger("Sales");

  @Builder.Task(id = "extract")
  public long extract(Client client) {
    var conn = client.getConnection("sales_db");
    log.log(INFO, "reading {0}", conn.host);
    return recordCount;
  }

  @Builder.Task(id = "transform")
  public long transform(Client client,
      @Builder.XCom(task = "extract") long extracted) {
    var threshold = client.getVariable("threshold");
    return extracted * 2;
  }
}
```

Read Connections and Variables.

SalesPipeline.java — annotation API

```java
@Builder.Dag(id = "sales_pipeline")
public class SalesPipeline {
  static final System.Logger log = System.getLogger("Sales");

  @Builder.Task(id = "extract")
  public long extract(Client client) {
    var conn = client.getConnection("sales_db");
    log.log(INFO, "reading {0}", conn.host);
    return recordCount;
  }

  @Builder.Task(id = "transform")
  public long transform(Client client,
      @Builder.XCom(task = "extract") long extracted) {
    var threshold = client.getVariable("threshold");
    return extracted * 2;
  }
}
```

Log the way you already log.

SalesPipeline.java — annotation API

```java
@Builder.Dag(id = "sales_pipeline")
public class SalesPipeline {
  static final System.Logger log = System.getLogger("Sales");

  @Builder.Task(id = "extract")
  public long extract(Client client) {
    var conn = client.getConnection("sales_db");
    log.log(INFO, "reading {0}", conn.host);
    return recordCount;
  }

  @Builder.Task(id = "transform")
  public long transform(Client client,
      @Builder.XCom(task = "extract") long extracted) {
    var threshold = client.getVariable("threshold");
    return extracted * 2;
  }
}
```

Declare what the bundle contains.

STEP 3 · ENTRY POINT

Main.java — the JVM entry point

```java
public class Main implements BundleBuilder {

  @Override
  public Iterable<Dag> getDags() {
    return List.of(SalesPipelineBuilder.build());
  }

  public static void main(String[] args) {
    Server.create(args).serve(new Main().build());
  }
}
```

One entry point, every Dag.

One artifact can serve hundreds of Dags.

Serve it to the supervisor.

STEP 3 · ENTRY POINT

Main.java — the JVM entry point

```java
public class Main implements BundleBuilder {

  @Override
  public Iterable<Dag> getDags() {
    return List.of(SalesPipelineBuilder.build());
  }

  public static void main(String[] args) {
    Server.create(args).serve(new Main().build());
  }
}
```

Gradle config setup.

build.gradle.kts — Maven Central

```
plugins {
    id("org.apache.airflow.sdk") version "1.0.0-beta1"
}

dependencies {
    implementation(platform(
        "org.apache.airflow:airflow-sdk-bom:1.0.0-beta1"))

    implementation("org.apache.airflow:airflow-sdk")
    annotationProcessor(
        "org.apache.airflow:airflow-sdk-processor")
    implementation("org.apache.airflow:airflow-sdk-jpl")
}
```

Fat JAR or thin JAR.

build.gradle.kts — fat JAR (default)

```
airflowSdk {
    mainClass = "com.example.Main"
    // fatJar = true is the default
}
```

build.gradle.kts — thin JAR

```
airflowSdk {
    mainClass = "com.example.Main"
    fatJar = false
}
```

Fat JAR: one file, classpath issues caught at build time. **Thin JAR**: shared libs under jars_root, faster deploys at scale.

Package it.

shell — the deployable

```
$ ./gradlew bundle
$ ls build/bundle/
my-app-all.jar
```

my-app-all.jar — META-INF/MANIFEST.MF

```
Main-Class: com.example.Main
Airflow-Supervisor-Schema-Version: 2026-06-16
```

The bundle declares its own protocol, so it survives an Airflow upgrade.

That is the whole authoring surface.

- `dags/sales_pipeline.py`
  @task.stub, scheduling, the wiring, retries
- `SalesPipeline.java`
  @Builder.Dag, @Builder.Task, @Builder.XCom, Client
- `Main.java`
  BundleBuilder, Server.create(args).serve(…)
- `build.gradle.kts`
  ./gradlew bundle

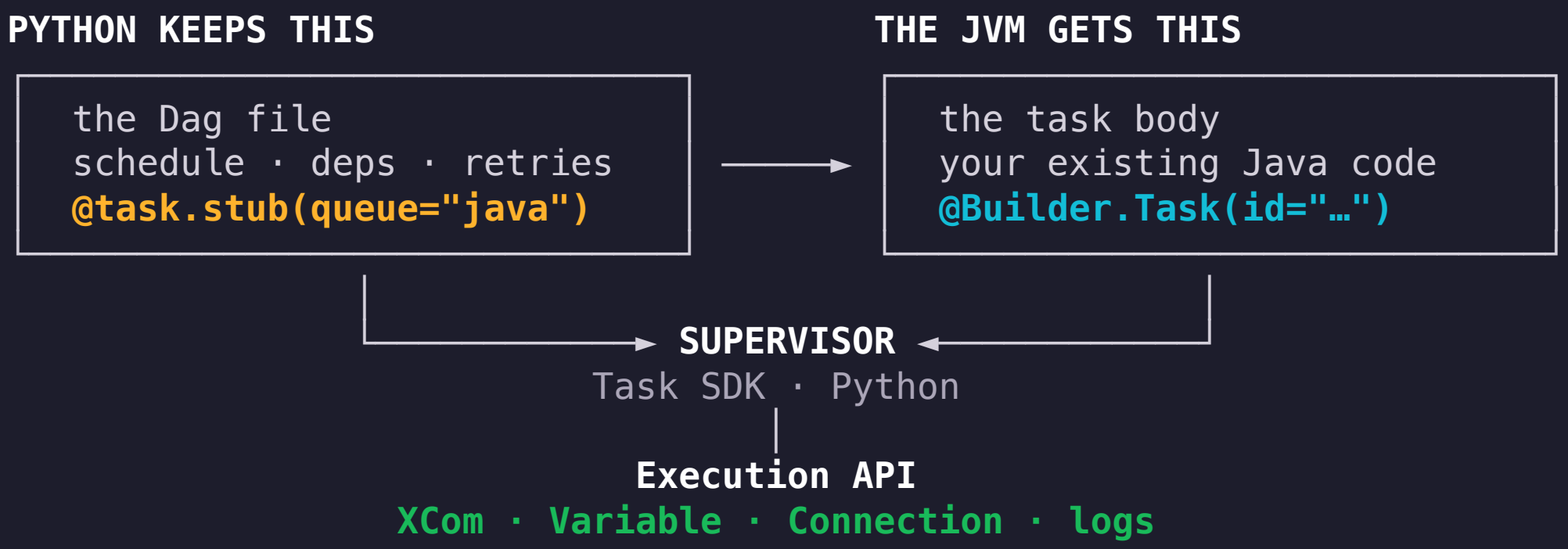
Everything else in this talk is **what Airflow does with those four things**.

02 / 05

How it actually runs

Routing, the seam, the wire protocol, and the coordinator interface.

The seam is queue=.



Python owns the *when*. Any language can own the *what*.

Route the queue to the JVM.

environment [sdk] section

```
export AIRFLOW__SDK__COORDINATORS='{
  "jdk-17": {
    "classpath": "airflow.sdk.coordinators.java.JavaCoordinator",
    "kwargs": {
      "jars_root": ["/opt/airflow/jars"]
    }}}'

export AIRFLOW__SDK__QUEUE_TO_COORDINATOR='{"java": "jdk-17"}'
```

What jars_root actually points at.


```
jars_root: ["/opt/airflow/jars"]
```

```
/opt/airflow/jars/
├── my-app-all.jar          Main-Class: com.example.Main
├── libs/
│   └── shared-utils.jar
└── plugins/
    └── legacy-fetcher.jar
```

Scanned recursively, on every launch. Every JAR under the path joins the classpath.

Two JVMs, two queues, one Airflow.

environment [sdk] section

```
export AIRFLOW__SDK__COORDINATORS='{
  "jdk-11": {"classpath": "...JavaCoordinator",
             "kwargs": {"jars_root": [...],
               "java_executable": "/opt/jdk11/bin/java"}},
  "jdk-17": {"classpath": "...JavaCoordinator",
             "kwargs": {"jars_root": [...],
               "java_executable": "/opt/jdk17/bin/java"}}
}'

export AIRFLOW__SDK__QUEUE_TO_COORDINATOR='{"java11": "jdk-11", "java17": "jdk-17"}'
```

Same coordinator class, twice, each with its own java_executable.

The stub's queue="java11" or queue="java17" picks the JVM.

What the coordinator takes.

JAVACOORDINATOR KWARGS

KWARG	DEFAULT	NOTE
`jars_root`	required	scanned recursively, string or list
`java_executable`	"java" from $PATH	**pin it to an absolute path**
`jvm_args`	[]	e.g. -Xmx1g
`main_class`	auto-detected	set it once you have two executable JARs
`task_startup_timeout`	10.0 seconds	raise it for a large classpath

Config changes need a restart. **A rebuilt JAR does not.**

Three things that follow from it.

No JVM to parse a Dag.

A stub has no body, so the Dag processor never launches Java.

Works with any executor.

Routing lives in the Task SDK, not the executor.

No second data plane.

One Execution API, whichever language asks.

Nothing new on the scheduler, the executor, or the data plane.

Configuring each executor.

WHAT CHANGES, EXECUTOR BY EXECUTOR

Local & Celery

A ONE-TIME HOST CHANGE

- java on the worker host
- or pin java_executable
- jars_root on that host's disk

Kubernetes

A ONE-TIME IMAGE CHANGE

- a JRE in the **worker pod image**
- jars_root resolvable inside the pod
- a pod template for that queue

Same [sdk] config either way. **The pod template is the only extra step.**

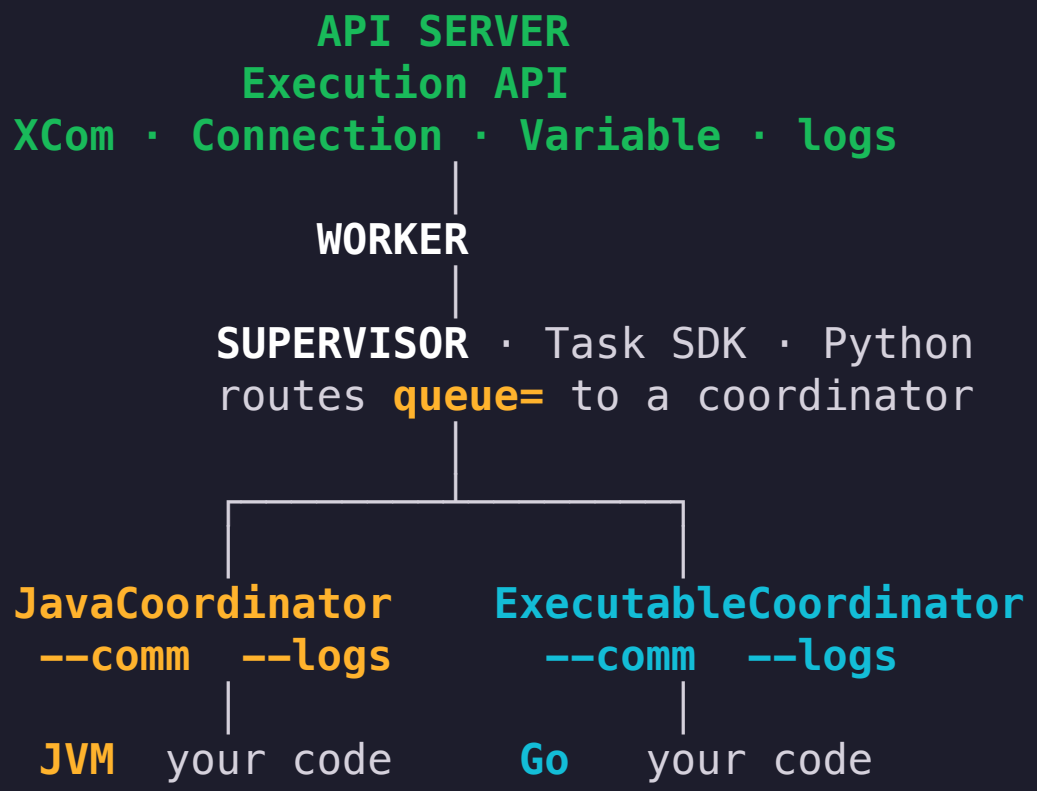
One queue, one pod template, one image.

environment [sdk] coordinators

```
{
  "jdk-17": {
    "classpath": "airflow.sdk.coordinators.java.JavaCoordinator",
    "kwargs": {
      "jars_root": ["/opt/airflow/java-bundles"],
      "java_executable": "/usr/lib/jvm/java-17-openjdk/bin/java"
    },
    "extra": {
      "pod_template_file": "/opt/airflow/pod_templates/java.yaml",
      "worker_container_repository": "apache/airflow",
      "worker_container_tag": "3.3.0"
    }
  },
  "go-sdk": {
    "classpath": "airflow.sdk.coordinators.executable.ExecutableCoordinator",
    "kwargs": {"executables_root": ["/opt/airflow/executable-bundles"]}
  }
}
```

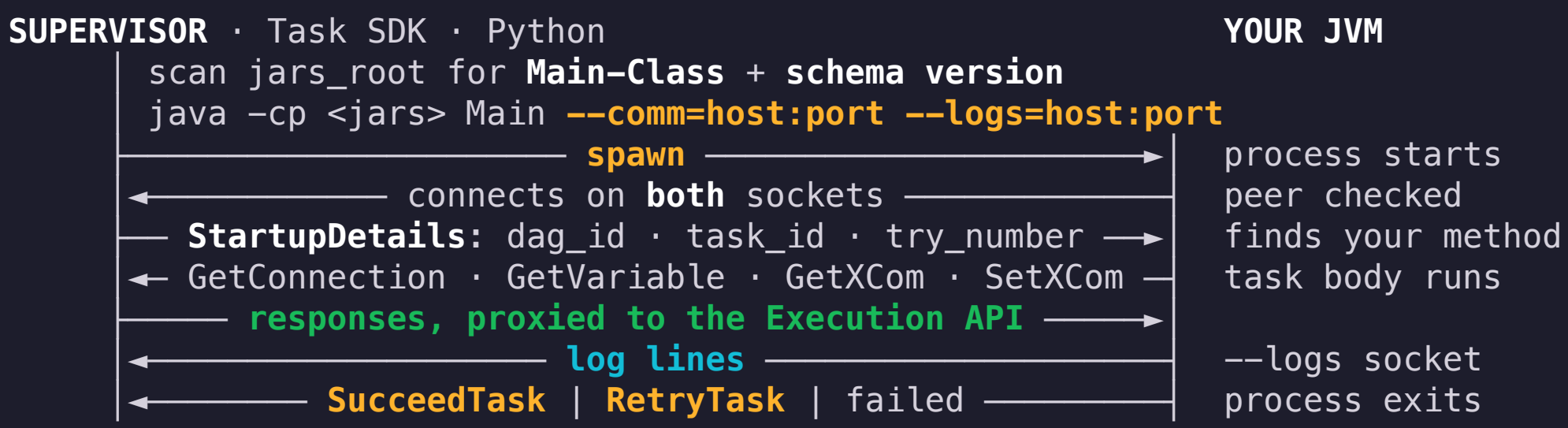
The full picture.

ZOOMED OUT



One supervisor per task, one Execution API for all languages.

Two sockets, then your method.



The JVM holds no DB connection and no API token.

Length-prefixed MessagePack on --comm, JSON log records on --logs.

The coordinator interface.

airflow/sdk/coordinators/ — Airflow 3.3

```python
class BaseCoordinator:
    def execute_task(self, *, what, ...) -> ExecutionResult:
        # ExecutionResult(exit_code, final_state)
        raise NotImplementedError


class SubprocessCoordinator(BaseCoordinator):
    # binds both sockets, spawns the child, frames
    # msgpack, drains logs, records terminal state

    def _build_execute_task_command(self, *, what):
        # -> (argv, schema_version)
        raise NotImplementedError
```

Java is the first runtime, not the only one.

ANY LANGUAGE SDK

Airflow 3.3

JavaCoordinator

Airflow 3.3

ExecutableCoordinator

after 3.3

NodeCoordinator

yours

subclass the base

Any process that speaks the protocol can run Airflow tasks.

Java is the one you already have in production.

03 / 05

The primitives, in detail

XCom, Connections, Variables and logs, across the language boundary.

XCom is JSON, so the mapping is fixed.

PYTHON	JSON	JAVA, FROM GETXCOM
`int`	number (integer)	`Long / BigInteger`
`float`	number (decimal)	`Double`
`str · bool · None`	string · boolean · null	`String · Boolean · null`
`list`	array	`List<Object>`
`dict`	object	`Map<String, Object>`

Read a mapped upstream with mapIndex = null and you get the aggregated list.

Boxed, whenever the XCom can be absent.

THE ONE REAL TRAP

the failure — MissingXComException

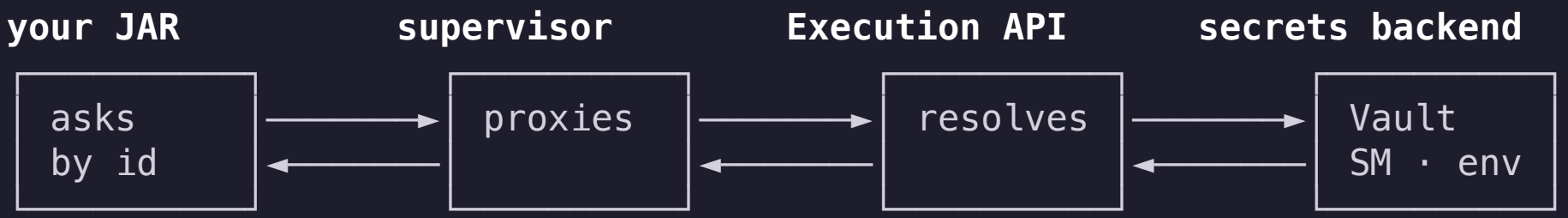
```
@Builder.XCom(task = "maybe") long    n   // fails

@Builder.XCom(task = "maybe") Long    n   // null
```

Loud failure, one-keystroke fix.

Ask by id. Airflow resolves the secret.

SalesPipeline.java — Connections and Variables

```java
// conn.host · conn.login · conn.password · conn.port · conn.extra
Connection conn = client.getConnection("sales_db");
Object threshold = client.getVariable("threshold");
```



not in the JAR · not in the argv · not in the image: the credential

Rotating the credential does not touch your bundle.

Variables are read-only from Java today: getVariable only.

Your existing logger, in the task log.

WHAT YOUR CODE ALREADY USES	ARTIFACT	SETUP
System.Logger (JEP 264)	`airflow-sdk-jpl`	none, ServiceLoader finds it
SLF4J 2.x or Logback	`airflow-sdk-slf4j`	none, ServiceLoader finds it
Log4j 2	`airflow-sdk-log4j2`	declare <AirflowAppender/>
java.util.logging	`airflow-sdk-jul`	call AirflowJulHandler.setup()

One bridge on the classpath, never two.

What is in flight

Bundle deployment, TaskFlow syntax, and native Java Dags. All three have open PRs.

Ship the JARs inside the Dag bundle.

#70805 OPEN

The JARs ride in the same bundle as the Dag that declares the stubs.

TaskFlow syntax, native to every SDK.

#69757 OPEN · ANY LANGUAGE

TODAY

```
Dag file
  transform()
  no arguments allowed
```

wiring, written twice

```
Java
  @Builder.XCom(
    task="extract")
```

WITH #69757

```
Dag file
  transform("uk",
            extract())
```

arg-binding spec

```
Any SDK
  the runtime binds
  the parameters
```

One Python-side capability, every language SDK gets it.

The Dag file already knows which task feeds which.

Dynamic Task Mapping, for every SDK.

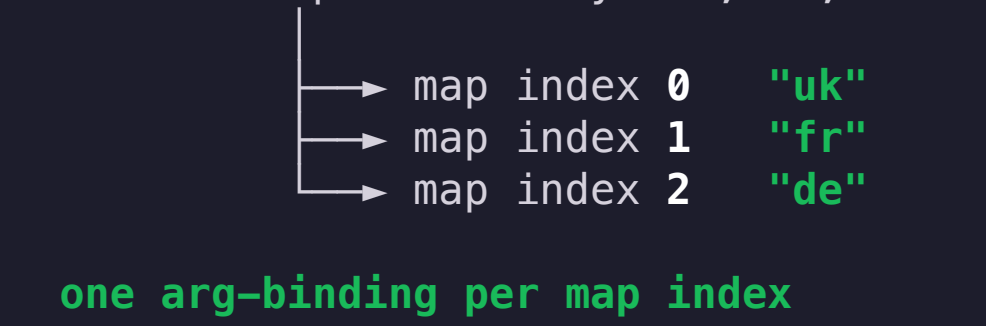
#70570 DRAFT · ANY LANGUAGE

TODAY

```
transform.expand(country=[uk, fr, de])
```

every map index gets the **whole list**, and **picks its own element**

WITH #70570



A mapped stub gets one arg-binding per map index.

Expand over a literal, an unmapped upstream, or another mapped task.

Two ways to build a Dag.

SHIPPED IN 3.3

Mixed-language Dag

A **Python Dag** with @task.stub placeholders; Java implements the task bodies.

Everything in section 1 was this.

WORK IN PROGRESS · #71057

Native Java Dag

No Python file alongside. The Dag itself is declared in Java with @Dag, @Task and a compile-checked depends().

Draft, stacked on #69757.

The task code is the same either way.

A whole Dag, authored in Java.

#71057 DRAFT

EtlPipeline.java — no Python file

```java
@Builder.Dag(id = "java_etl", schedule = "@daily")
public class EtlPipeline {
  @Builder.Task(id = "extract", retries = 2)
  public ExtractResult extract(Client client) { ... }
  @Builder.Task
  public TransformResult transform(Client client,
      ExtractResult in) { ... }
  @Builder.Task
  public void load(Context ctx, TransformResult in) { ... }

  static void depends() {
    load(transform(extract()));
  }
}
```

Dag-level arguments, in Java.

#71057 DRAFT

EtlPipeline.java — no Python file

```java
@Builder.Dag(id = "java_etl", schedule = "@daily")
public class EtlPipeline {
  @Builder.Task(id = "extract", retries = 2)
  public ExtractResult extract(Client client) { ... }
  @Builder.Task
  public TransformResult transform(Client client,
      ExtractResult in) { ... }
  @Builder.Task
  public void load(Context ctx, TransformResult in) { ... }

  static void depends() {
    load(transform(extract()));
  }
}
```

Task-level arguments, in Java.

#71057 DRAFT

EtlPipeline.java — no Python file

```java
@Builder.Dag(id = "java_etl", schedule = "@daily")
public class EtlPipeline {
  @Builder.Task(id = "extract", retries = 2)
  public ExtractResult extract(Client client) { ... }
  @Builder.Task
  public TransformResult transform(Client client,
      ExtractResult in) { ... }
  @Builder.Task
  public void load(Context ctx, TransformResult in) { ... }

  static void depends() {
    load(transform(extract()));
  }
}
```

@Builder.Task carries id and retries, same as the stub did.

Dependency wiring, compile-checked.

#71057 DRAFT

EtlPipeline.java — no Python file

```java
@Builder.Dag(id = "java_etl", schedule = "@daily")
public class EtlPipeline {
  @Builder.Task(id = "extract", retries = 2)
  public ExtractResult extract(Client client) { ... }
  @Builder.Task
  public TransformResult transform(Client client,
      ExtractResult in) { ... }
  @Builder.Task
  public void load(Context ctx, TransformResult in) { ... }

  static void depends() {
    load(transform(extract()));
  }
}
```

A typo in the dependency graph is a compile error, not a parse error at deploy time.

Interface form: TaskRef into a DagRef. Next slide.

Or the interface form, same Dag.

#71057 DRAFT

EtlPipeline.java — interface form

```java
var extract = new TaskRef("extract", Extract.class).config("retries", 2);
var transform = new TaskRef("transform", Transform.class).dependsOn(extract);

var dag = new DagRef("java_etl")
    .config("schedule", "@daily")
    .addTask(extract)
    .addTask(transform);
```

Same two syntaxes as section 1, one level up: a Dag instead of a task.

.config(k, v) works Spark-style on both TaskRef and DagRef.

05 / 05

Demo, limits, and where to get it

What it costs, and where to get it.

java_examples.py 1 | nExample.java | ExampleBundleBuilder.java | docker-compose.override.yml | README

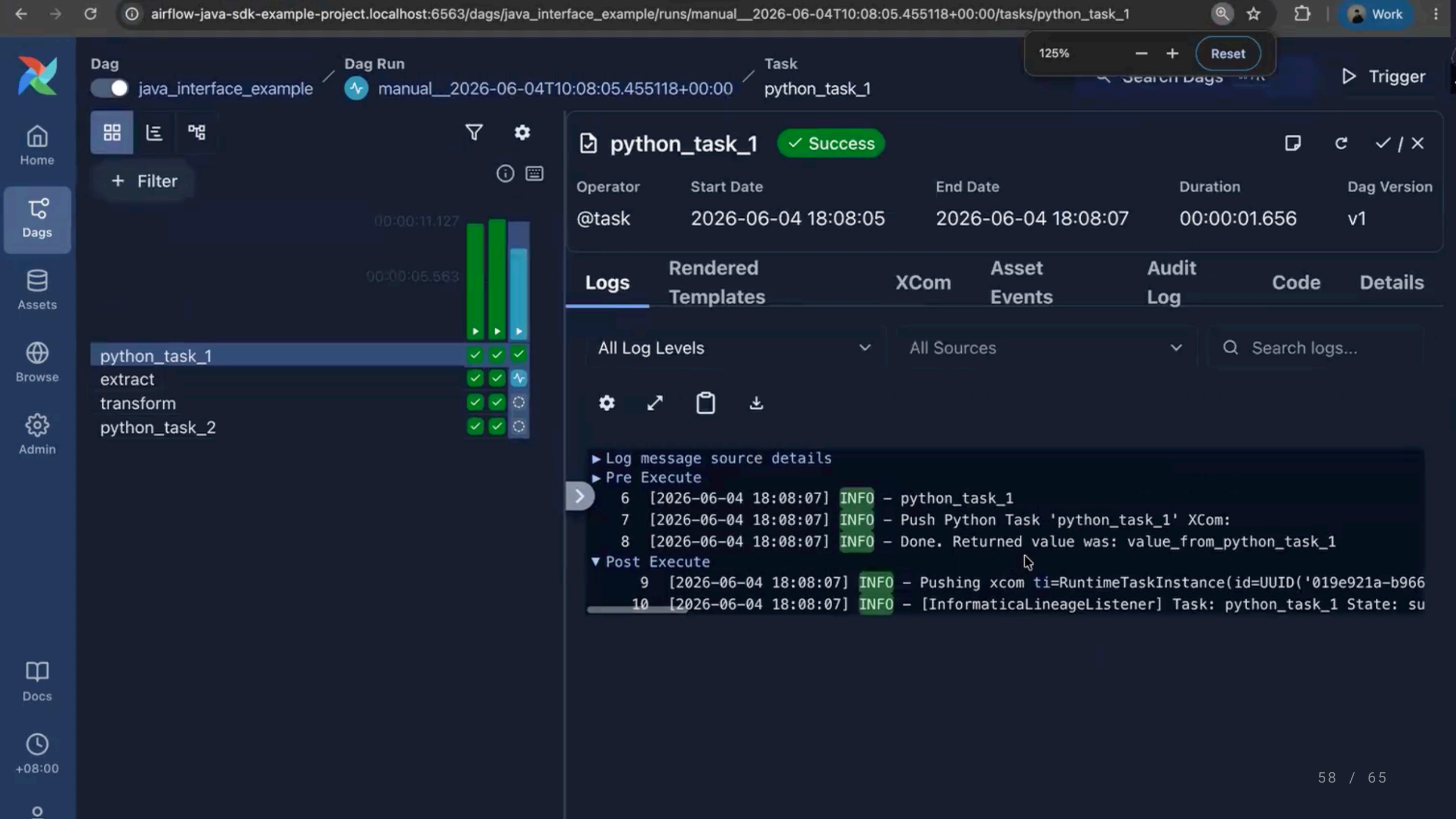
dags > java_examples.py > extract

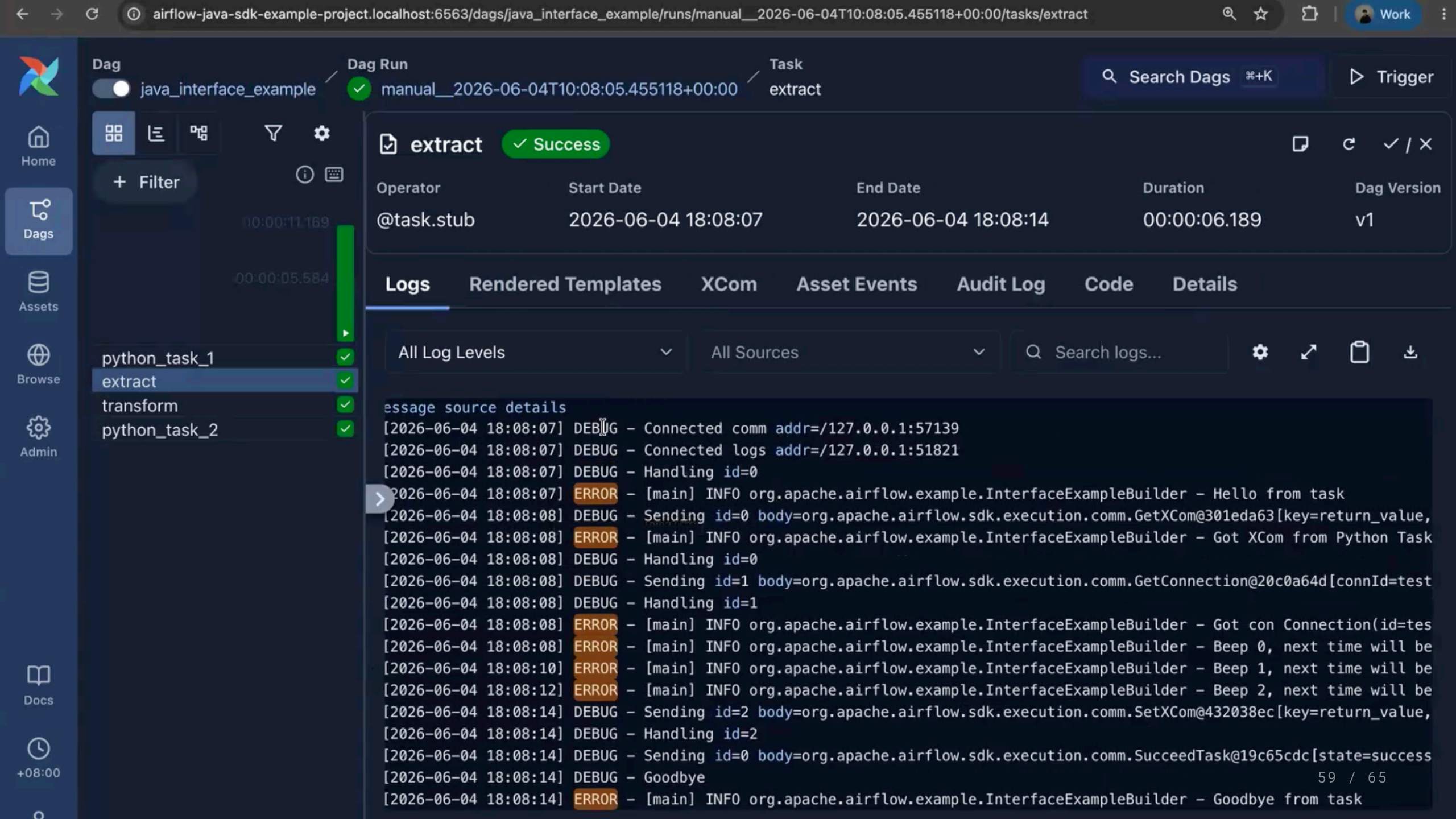
```python
@task()
def python_task_1():
    print("python_task_1")
    print("Push Python Task 'python_tas
    return "value_from_python_task_1"


@task.stub(queue="java")
def extract(): ...


@task.stub(queue="java")
def transform(): ...


@task()
def python_task_2(transformed):
    print("python_task_2")
    print("Pull Java Task 'transform' X
    print(transformed)


@dag(dag_id="java_interface_example")
def java_interface_example():
    transformed = transform()
    python_task_1() >> extract() >> tra
    python_task_2(transformed)


@dag(dag_id="java_annotation_example")
```

docker-compose.override.yml

```yaml
#
# Astro's local stack runs the LocalExecutor, so tasks execute inside th
# `scheduler` container — that is the component that launches the JavaCo
# (there is no separate `worker` under LocalExecutor). We therefore atta
# routing config to `scheduler` only. If you switch the stack to Celeryl
# move this block to the `worker` service instead.
services:
  scheduler:
    environment:
      # Register a coordinator instance named "java" pointing at the bur
      # baked into the image (see Dockerfile), ...
      AIRFLOW__SDK__COORDINATORS: >-
        {"java-sdk": {"classpath": "airflow.sdk.coordinators.java.JavaCo
        "kwargs": {"jars_root": ["/usr/local/airflow/java-bundle/lib"]}}
      # ... and route the "java" task queue to it.
      AIRFLOW__SDK__QUEUE_TO_COORDINATOR: '{"java": "java-sdk"}'
```

```java
@SuppressWarnings("DuplicatedCode")
public class InterfaceExampleBuilder {
  private static final Logger logger = LoggerFactory.getLogger(InterfaceExampleBuilder.class);

  public static class Extract implements Task {
    public void execute(@NotNull Context context, Client client) throws Exception {
      logger.info("Hello from task");

      var pythonInput = client.getXCom("python_task_1");
      logger.info("Got XCom from Python Task 'python_task_1' {}", pythonInput);

      var connection = client.getConnection("test_http");
      logger.info("Got con {}", connection);

      for (var i = 0; i < 3; i++) {
        logger.info("Beep {}, next time will be {}", i, new Date());
        Thread.sleep(2 * 1000);
      }

      client.setXCom(new Date().getTime());
      logger.info("Goodbye from task");
    }
  }

  public static class Transform implements Task {
    public void execute(@NotNull Context context, Client client) {
      var extracted = client.getXCom("extract");
      logger.info("Got XCom from 'extract' {}", extracted);

      var variable = client.getVariable("my_variable");
```


airflow-java-sdk-example-project.localhost:6563/dags/java_interface_example/runs/manual__2026-06-04T10:08:05.455118+00:00/tasks/python_task_1
125%
Reset
Work
Dag
java_interface_example
Dag Run
manual__2026-06-04T10:08:05.455118+00:00
Task
python_task_1
Trigger
Home
Dags
Assets
Browse
Admin
Docs
+08:00
Filter
00:00:11.127
00:00:05.563
python_task_1
extract
transform
python_task_2
python_task_1
Success
Operator
@task
Start Date
2026-06-04 18:08:05
End Date
2026-06-04 18:08:07
Duration
00:00:01.656
Dag Version
v1
Logs
Rendered Templates
XCom
Asset Events
Audit Log
Code
Details
All Log Levels
All Sources
Search logs...
▶ Log message source details
▶ Pre Execute
6 [2026-06-04 18:08:07] INFO - python_task_1
7 [2026-06-04 18:08:07] INFO - Push Python Task 'python_task_1' XCom:
8 [2026-06-04 18:08:07] INFO - Done. Returned value was: value_from_python_task_1
▼ Post Execute
9 [2026-06-04 18:08:07] INFO - Pushing xcom ti=RuntimeTaskInstance(id=UUID('019e921a-b966
10 [2026-06-04 18:08:07] INFO - [InformaticaLineageListener] Task: python_task_1 State: su


airflow-java-sdk-example-project.localhost:6563/dags/java_interface_example/runs/manual__2026-06-04T10:08:05.455118+00:00/tasks/extract
Work
Dag
java_interface_example
Dag Run
manual__2026-06-04T10:08:05.455118+00:00
Task
extract
Search Dags ⌘+K
Trigger
Home
Dags
Assets
Browse
Admin
Docs
+08:00
Filter
00:00:11.169
00:00:05.584
python_task_1
extract
transform
python_task_2
extract
Success
Operator
@task.stub
Start Date
2026-06-04 18:08:07
End Date
2026-06-04 18:08:14
Duration
00:00:06.189
Dag Version
v1
Logs
Rendered Templates
XCom
Asset Events
Audit Log
Code
Details
All Log Levels
All Sources
Search logs...
essage source details
[2026-06-04 18:08:07] DEBUG - Connected comm addr=/127.0.0.1:57139
[2026-06-04 18:08:07] DEBUG - Connected logs addr=/127.0.0.1:51821
[2026-06-04 18:08:07] DEBUG - Handling id=0
2026-06-04 18:08:07] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Hello from task
[2026-06-04 18:08:08] DEBUG - Sending id=0 body=org.apache.airflow.sdk.execution.comm.GetXCom@301eda63[key=return_value,
[2026-06-04 18:08:08] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Got XCom from Python Task
[2026-06-04 18:08:08] DEBUG - Handling id=0
[2026-06-04 18:08:08] DEBUG - Sending id=1 body=org.apache.airflow.sdk.execution.comm.GetConnection@20c0a64d[connId=test
[2026-06-04 18:08:08] DEBUG - Handling id=1
[2026-06-04 18:08:08] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Got con Connection(id=tes
[2026-06-04 18:08:08] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Beep 0, next time will be
[2026-06-04 18:08:10] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Beep 1, next time will be
[2026-06-04 18:08:12] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Beep 2, next time will be
[2026-06-04 18:08:14] DEBUG - Sending id=2 body=org.apache.airflow.sdk.execution.comm.SetXCom@432038ec[key=return_value,
[2026-06-04 18:08:14] DEBUG - Handling id=2
[2026-06-04 18:08:14] DEBUG - Sending id=0 body=org.apache.airflow.sdk.execution.comm.SucceedTask@19c65cdc[state=success
[2026-06-04 18:08:14] DEBUG - Goodbye
[2026-06-04 18:08:14] ERROR - [main] INFO org.apache.airflow.example.InterfaceExampleBuilder - Goodbye from task
59 / 65

airflow-java-sdk-example-project.localhost:6563/dags/java_interface_example/runs/manual__2026-06-04T10:08:05.455118+00:00/tasks/extract

Work

Dag: java_interface_example / Dag Run: manual__2026-06-04T10:08:05.455118+00:00 / Task: extract

Search Dags ⌘+K

Trigger

Home

Dags

Assets

Browse

Admin

Docs

+08:00

+ Filter

00:00:11.169

00:00:05.584

- python_task_1
- extract
- transform
- python_task_2

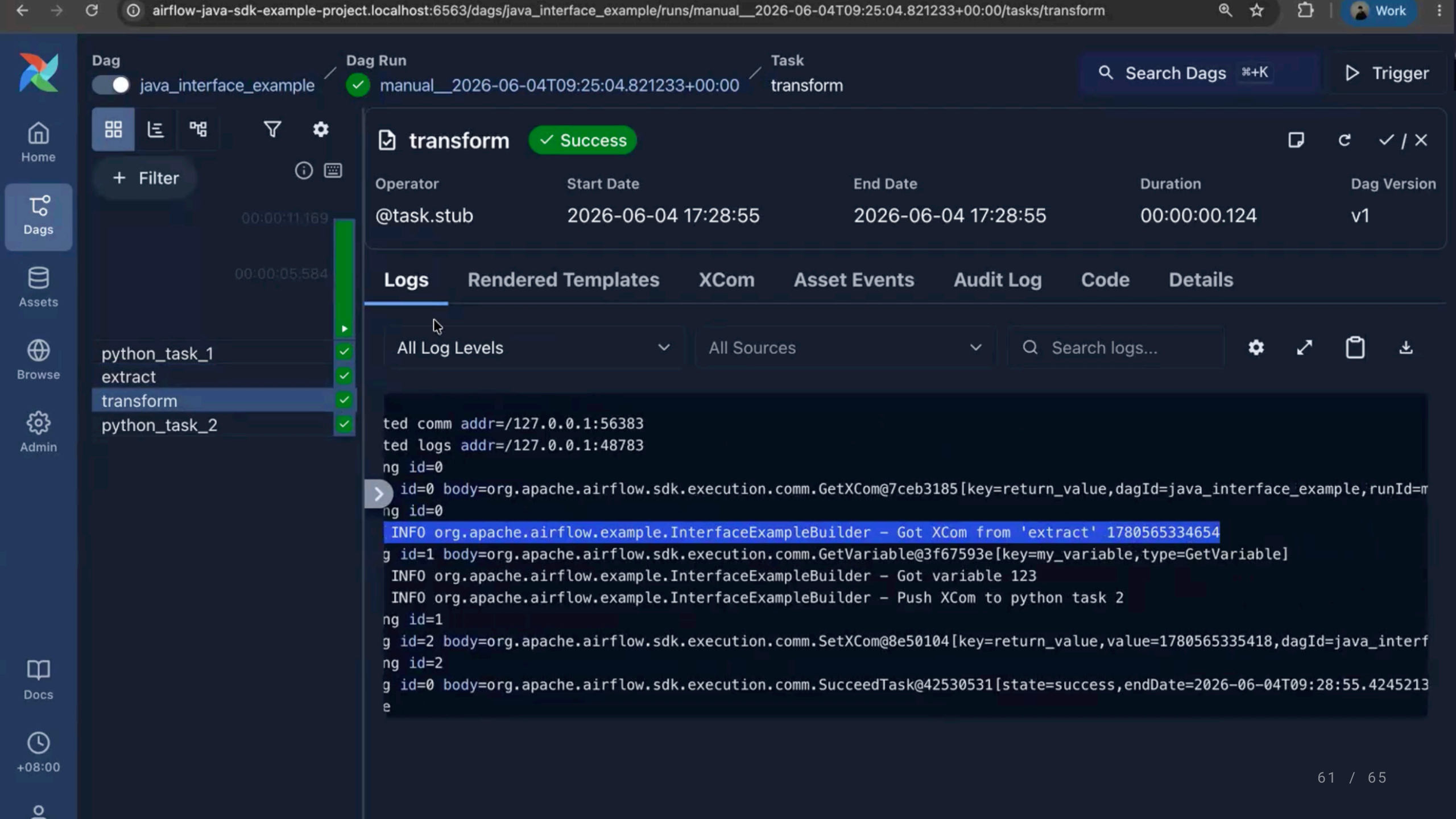
extract ✓ Success

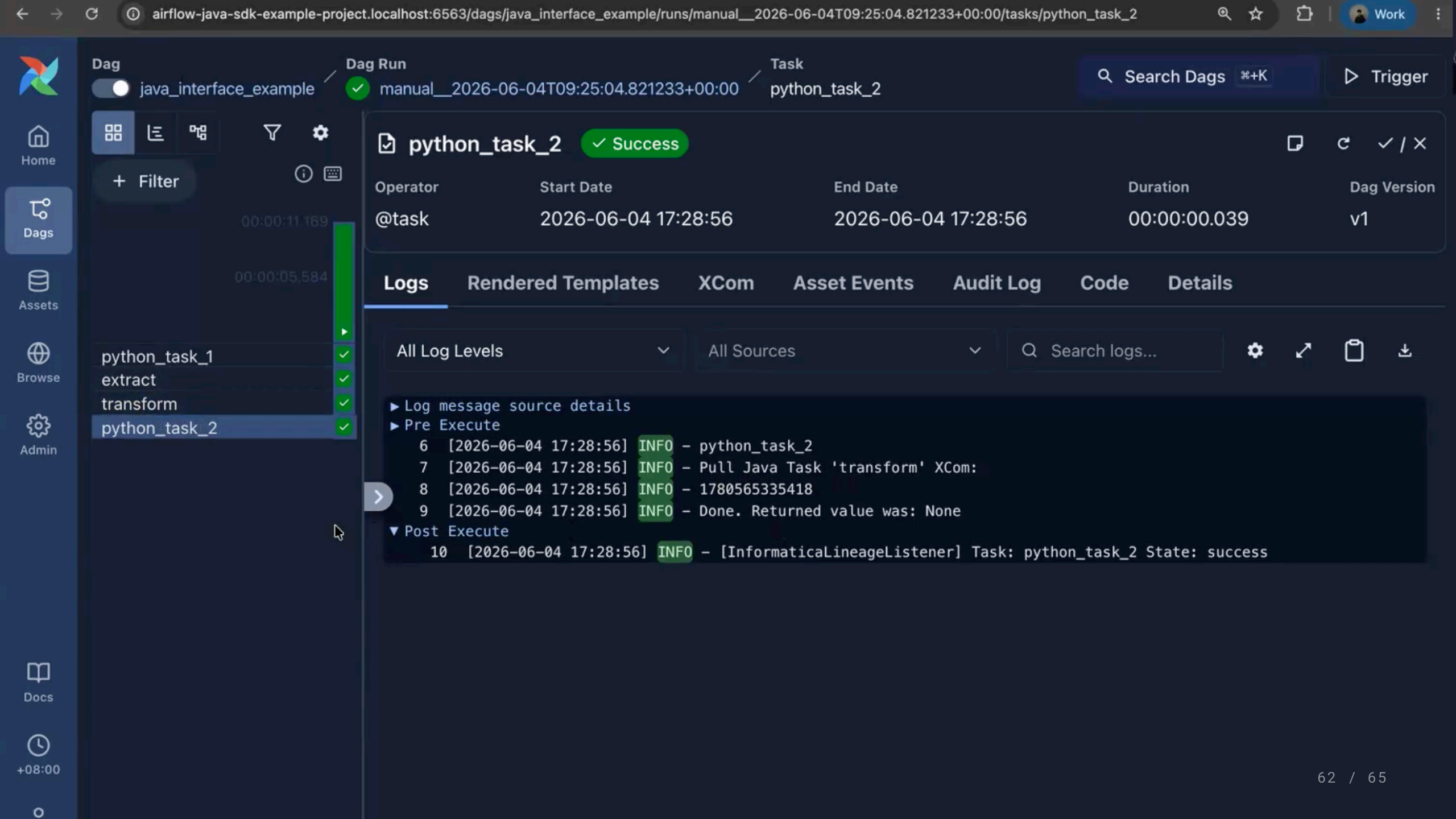
Operator	Start Date	End Date	Duration	Dag Version
@task.stub	2026-06-04 18:08:07	2026-06-04 18:08:14	00:00:06.189	v1

Logs | Rendered Templates | XCom | Asset Events | Audit Log | Code | Details

All Log Levels | All Sources | Search logs...

```
cted comm addr=/127.0.0.1:57139
cted logs addr=/127.0.0.1:51821
ing id=0
 INFO org.apache.airflow.example.InterfaceExampleBuilder - Hello from task
ng id=0 body=org.apache.airflow.sdk.execution.comm.GetXCom@301eda63[key=return_value,dagId=java_interface_example,runId=
] INFO org.apache.airflow.example.InterfaceExampleBuilder - Got XCom from Python Task 'python_task_1' value_from_python_
ing id=0
ng id=1 body=org.apache.airflow.sdk.execution.comm.GetConnection@20c0a64d[connId=test_http,type=GetConnection]
ing id=1
] INFO org.apache.airflow.example.InterfaceExampleBuilder - Got con Connection(id=test_http, type=http, host=example.com
] INFO org.apache.airflow.example.InterfaceExampleBuilder - Beep 0, next time will be Thu Jun 04 10:08:08 UTC 2026
] INFO org.apache.airflow.example.InterfaceExampleBuilder - Beep 1, next time will be Thu Jun 04 10:08:10 UTC 2026
] INFO org.apache.airflow.example.InterfaceExampleBuilder - Beep 2, next time will be Thu Jun 04 10:08:12 UTC 2026
ng id=2 body=org.apache.airflow.sdk.execution.comm.SetXCom@432038ec[key=return_value,value=1780567694048,dagId=java_inte
ing id=2
ng id=0 body=org.apache.airflow.sdk.execution.comm.SucceedTask@19c65cdc[state=success,endDate=2026-06-04T10:08:14.05999
ye
] INFO org.apache.airflow.example.InterfaceExampleBuilder - Goodbye from task
```


airflow-java-sdk-example-project.localhost:6563/dags/java_interface_example/runs/manual__2026-06-04T09:25:04.821233+00:00/tasks/transform
Work
Dag
java_interface_example
Dag Run
manual__2026-06-04T09:25:04.821233+00:00
Task
transform
Search Dags ⌘+K
Trigger
Home
Dags
Assets
Browse
Admin
Docs
+08:00
Filter
00:00:11.169
00:00:05.584
python_task_1
extract
transform
python_task_2
transform
Success
Operator
@task.stub
Start Date
2026-06-04 17:28:55
End Date
2026-06-04 17:28:55
Duration
00:00:00.124
Dag Version
v1
Logs
Rendered Templates
XCom
Asset Events
Audit Log
Code
Details
All Log Levels
All Sources
Search logs...
ted comm addr=/127.0.0.1:56383
ted logs addr=/127.0.0.1:48783
ng id=0
id=0 body=org.apache.airflow.sdk.execution.comm.GetXCom@7ceb3185[key=return_value,dagId=java_interface_example,runId=m
ng id=0
INFO org.apache.airflow.example.InterfaceExampleBuilder - Got XCom from 'extract' 1780565334654
g id=1 body=org.apache.airflow.sdk.execution.comm.GetVariable@3f67593e[key=my_variable,type=GetVariable]
INFO org.apache.airflow.example.InterfaceExampleBuilder - Got variable 123
INFO org.apache.airflow.example.InterfaceExampleBuilder - Push XCom to python task 2
ng id=1
g id=2 body=org.apache.airflow.sdk.execution.comm.SetXCom@8e50104[key=return_value,value=1780565335418,dagId=java_interf
ng id=2
g id=0 body=org.apache.airflow.sdk.execution.comm.SucceedTask@42530531[state=success,endDate=2026-06-04T09:28:55.4245213
e


airflow-java-sdk-example-project.localhost:6563/dags/java_interface_example/runs/manual__2026-06-04T09:25:04.821233+00:00/tasks/python_task_2
Work
Dag
java_interface_example
Dag Run
manual__2026-06-04T09:25:04.821233+00:00
Task
python_task_2
Search Dags ⌘+K
Trigger
Home
Dags
Assets
Browse
Admin
Docs
+08:00
Filter
00:00:11.169
00:00:05.584
python_task_1
extract
transform
python_task_2
python_task_2
Success
Operator
@task
Start Date
2026-06-04 17:28:56
End Date
2026-06-04 17:28:56
Duration
00:00:00.039
Dag Version
v1
Logs
Rendered Templates
XCom
Asset Events
Audit Log
Code
Details
All Log Levels
All Sources
Search logs...
Log message source details
Pre Execute
6 [2026-06-04 17:28:56] INFO - python_task_2
7 [2026-06-04 17:28:56] INFO - Pull Java Task 'transform' XCom:
8 [2026-06-04 17:28:56] INFO - 1780565335418
9 [2026-06-04 17:28:56] INFO - Done. Returned value was: None
Post Execute
10 [2026-06-04 17:28:56] INFO - [InformaticaLineageListener] Task: python_task_2 State: success

What it costs.

One JVM per task, no warm pool.

Noise for a Spark driver. A bad trade for a 10-second task.

A worker slot for the whole run.

No deferral yet, so keep long waits in Python.

Experimental in 3.3.0

Off by default, and the SDK is at beta.

The same model that costs you startup is what makes a JAR swap free.

Get started and tell us what breaks.

COPYABLE TODAY

build.gradle.kts — Maven Central

```
implementation(platform(
  "org.apache.airflow:airflow-sdk-bom:1.0.0-beta1"))

implementation("org.apache.airflow:airflow-sdk")
```

AIRFLOW 3.3+ | JRE 11+ | EXPERIMENTAL

- **Docs:** Non-Python Task SDKs
- **API:** `airflow.apache.org/docs/java-sdk/`
- **Example:** `java-sdk/example/`

Feedback on the SDK and the roadmap is welcome in #sig-lang-sdks.

Questions?

We are looking forward to any feedback on the Language SDKs and the roadmap.

Jason(Zhe-You) Liu • Astronomer

jason.liu@astronomer.io

github.com/jason810496

linkedin.com/in/zhe-you-liu