1. Prerequisites

- JDK 11 or 17 installed (match cluster/runtime). Verify: `java -version`
- sbt installed (`sbt sbtVersion`)
- A Spark distribution available for cluster runs; for local development you can run with `spark-submit` or run in `local[*]` from code.
- An IDE like IntelliJ IDEA (recommended) with Scala plugin.

> Tip: **Make Scala version match your Spark build** (many Spark builds are compiled for Scala 2.12; if you use a Spark build for Scala 2.13 or Scala 3 adjust accordingly).

---

2. Project scaffold (sbt layout)

Create files and folders:

```
spark-etl/
  build.sbt
  project/
    plugins.sbt
  src/
    main/
      scala/
        com/example/etl/
          SalesEtl.scala
          SalesTransforms.scala
    test/
      scala/
        com/example/etl/
          SalesTransformsSpec.scala
  data/
    input/  (put sample CSVs here)
```

build.sbt (adjust Spark & Scala versions to match your environment):

```
name := "spark-etl"
version := "0.1.0"
ThisBuild / scalaVersion := "2.12.18" // set to the Scala version
compatible with your Spark build

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "3.4.1"  % "provided",
  "org.apache.spark" %% "spark-sql"  % "3.4.1"  % "provided",
  // for unit tests
  "org.scalatest"   %% "scalatest"  % "3.2.15" % "test"
)
```

project/plugins.sbt (optional — for assembly if you want a fat jar):

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "1.2.0")
```

Notes:

- `provided` scope for Spark deps is recommended when submitting with `spark-submit` (cluster provides Spark). For pure local runs you can remove `provided`.
- Adjust versions to match your cluster.

---

3. ETL code (clean, transform, aggregate) — keep logic testable

Split logic into small pure functions (makes testing easy).

src/main/scala/com/example/etl/SalesTransforms.scala

```
package com.example.etl

import org.apache.spark.sql.{DataFrame, functions => F}
import org.apache.spark.sql.types.DoubleType
```

```
object SalesTransforms {

  /** normalize and basic cleaning */
  def clean(df: DataFrame): DataFrame = {
    df
      .withColumn("order_date", F.to_date(F.col("order_date"),
"yyyy-MM-dd"))
      .withColumn("amount", F.col("amount").cast(DoubleType))
      .filter(F.col("amount").isNotNull && F.col("amount") > 0)
  }

  /** add derived columns */
  def enrich(df: DataFrame): DataFrame = {
    df.withColumn("amount_usd", F.round(F.col("amount") *
F.lit(0.013), 2))
  }

  /** aggregate by region */
  def aggregateByRegion(df: DataFrame): DataFrame = {
    df.groupBy("region")
      .agg(
        F.sum("amount").as("total_amount"),
        F.count("*").as("orders"),
        F.round(F.avg("amount"), 2).as("avg_amount")
      )
  }
}
```

src/main/scala/com/example/etl/SalesEtl.scala

```
package com.example.etl

import org.apache.spark.sql.SparkSession

object SalesEtl extends App {
  val spark = SparkSession.builder()
```

```
    .appName("Sales ETL")
    // default to local for dev; override with -Dspark.master or use
spark-submit
    .master(sys.props.getOrElse("spark.master", "local[*]"))
    .getOrCreate()

  import spark.implicits._

  val inputPath  = args.headOption.getOrElse("data/input/sales.csv")
  val outputPath = args.lift(1).getOrElse("data/output/region_agg")

  val raw = spark.read
    .option("header", "true")
    .option("inferSchema", "false")   // we cast manually in
transforms
    .csv(inputPath)

  val cleaned    = SalesTransforms.clean(raw)
  val enriched   = SalesTransforms.enrich(cleaned)
  val aggregated = SalesTransforms.aggregateByRegion(enriched)

  aggregated.write.mode("overwrite").parquet(outputPath)

  spark.stop()
}
```

Notes:

- Keep IO (read/write) in `main`, keep transformations pure functions in `SalesTransforms` — that makes unit testing trivial.

---

4. Unit testing with ScalaTest (local SparkSession)

Use a test trait to create a local SparkSession once per test class.

src/test/scala/com/example/etl/SparkSessionTestWrapper.scala

```
package com.example.etl

import org.apache.spark.sql.SparkSession
import org.scalatest.BeforeAndAfterAll
import org.scalatest.Assertions._

trait SparkSessionTestWrapper extends BeforeAndAfterAll {
  lazy val spark: SparkSession = SparkSession.builder()
    .master("local[2]")
    .appName("spark-test")
    .getOrCreate()

  override def afterAll(): Unit = {
    spark.stop()
    super.afterAll()
  }
}
```

src/test/scala/com/example/etl/SalesTransformsSpec.scala

```
package com.example.etl

import org.scalatest.funsuite.AnyFunSuite
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

class SalesTransformsSpec extends AnyFunSuite with
SparkSessionTestWrapper {
  import spark.implicits._

  private def assertDataFrameEquals(expected:
org.apache.spark.sql.DataFrame,
                                    result:
org.apache.spark.sql.DataFrame): Unit = {
    assert(expected.schema === result.schema)
```

```
    assert(expected.collect().toSet === result.collect().toSet)
  }

  test("clean removes invalid rows and parses dates") {
    val input = Seq(
      ("1","2025-01-01","North","100.0"),
      ("2","2025-01-02","South","-10"),
      ("3","bad-date","East","50")
    ).toDF("id","order_date","region","amount")

    val cleaned = SalesTransforms.clean(input)
    assert(cleaned.count() == 1)
    val row =
cleaned.select("id","region","amount").as[(String,String,Double)].coll
ect().head
    assert(row._2 == "North")
    assert(row._3 == 100.0)
  }

  test("aggregateByRegion computes totals") {
    val input = Seq(
      ("North", 100.0),
      ("North", 50.0),
      ("South", 25.0)
    ).toDF("region","amount")

    val aggregated = SalesTransforms.aggregateByRegion(input)

    val expected = Seq(
      ("North", 150.0L, 2L, 75.0),
      ("South", 25.0L, 1L, 25.0)
    ).toDF("region","total_amount","orders","avg_amount")

    // cast types consistently or use custom comparator; simple set
compare here:
    val aggNormalized =
aggregated.select("region","total_amount","orders","avg_amount")
```

```
    val expectedNormalized =
expected.select("region","total_amount","orders","avg_amount")

    assertDataFrameEquals(expectedNormalized, aggNormalized)
  }
}
```

Testing tips:

- Compare DataFrames using sets (order independent). For floats, consider rounding before comparison.
- Avoid hitting disk in unit tests — use small in-memory DataFrames.
- For integration tests that read/write files, use temp directories (Java `Files.createTempDirectory`) and clean up.

---

5. Build, run, package, and submit

Local dev:

```
sbt compile
sbt test
```

Create a fat jar (if you want to run with `spark-submit` without cluster-provided Spark libs):

- Configure `sbt-assembly` plugin and run: `sbt assembly`
- This creates `target/scala-*/spark-etl-assembly-0.1.0.jar`.

Run locally with `spark-submit`:

```
spark-submit \
  --class com.example.etl.SalesEtl \
  --master local[*] \
  target/scala-2.12/spark-etl-assembly-0.1.0.jar \
  data/input/sales.csv data/output/region_agg
```

or run via sbt (dev):

```
sbt "run data/input/sales.csv data/output/region_agg"
```

If your `build.sbt` uses `provided` for Spark, use `spark-submit` (cluster provides Spark). If you want the jar to contain Spark, remove `provided` and assembly will bundle them (but beware of version conflicts on cluster).

---

6. CI (quick GitHub Actions example)

`.github/workflows/ci.yml`:

```
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          java-version: '17'
          distribution: 'temurin'
      - name: Setup sbt
        uses: olafurpg/setup-scala@v13
        with:
          java-version: '17'
      - name: sbt test
        run: sbt test
```

This runs unit tests quickly (no real Spark cluster). For integration tests that require `spark-submit`, you'd add a job that downloads Spark and runs the jar.

---

7. Common troubleshooting

- **No main class detected** — your `object` must have `def main(args: Array[String]): Unit` or `extends App`. Ensure `mainClass` setting is correct in `build.sbt` when running via certain SBT tasks.
- **Scala/Spark version mismatch** — use a Scala version compatible with Spark's build. If Spark is built for Scala 2.12, your project must use Scala 2.12.
- **UnsupportedClassVersionError / JNI** — Java runtime/compile mismatch. Make sure compile and runtime Java versions are compatible.
- **Tests failing because of ordering** — compare DataFrame contents as sets after normalizing schema and rounding floats.
