[Applications | Apache Flink](https://flink.apache.org/what-is-flink/flink-applications/" \l "state)

1. Terminology:

* Batch processing (bounded stream): can ingest the entire dataset to sort the data, compute global statistics, or produce a final report that summarizes all of the input.
* Stream processing (unbounded stream): the input may never end, and so you are forced to continuously process the data as it arrives.
* State: Operations remember information across multiple events (for example window operators). These operations are called **stateful**.
* **Processing time:** Processing time refers to the system time of the machine that is executing the respective operation.
* **Event time:** Event time is the time that each individual event occurred on its producing device.

1. Layerd APIs
2. The ProcessFunction: [Process Function | Apache Flink](https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/process_function/)

The ProcessFunction is a low-level stream processing operation, giving access to the basic building blocks of all (acyclic) streaming applications:

* State: lưu thông tin tạm thời **gắn liền với mỗi key.**
* Event (stream element): Mảnh dữ liệu chạy qua luồng stream.
* Timer: Cơ chế hẹn giờ để xử lý logic trong tương lai (hỗ trợ cả processing time và event time

Example:

public class InactivityDetector extends KeyedProcessFunction<String, String, String> {

    // Lưu timestamp của lần hoạt động cuối

    private ValueState<Long> lastActivityState;

    @Override

    public void open(Configuration parameters) {

        ValueStateDescriptor<Long> descriptor =

            new ValueStateDescriptor<>("lastActivity", Long.*class*);

        lastActivityState = getRuntimeContext().getState(descriptor);

    }

    @Override

    public void processElement(String event, Context ctx, Collector<String> out) throws Exception {

        long currentEventTime = ctx.timestamp();  // Lấy timestamp của event

        // Cập nhật state

        lastActivityState.update(currentEventTime);

        // Đăng ký timer để kiểm tra inactiveness sau 10 giây (10\_000 ms)

        ctx.timerService().registerEventTimeTimer(currentEventTime + 10\_000);

    }

    @Override

    public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out) throws Exception {

        Long lastActivity = lastActivityState.value();

*if* (lastActivity != null && timestamp >= lastActivity + 10\_000) {

            out.collect("⚠ User " + ctx.getCurrentKey() + " is inactive!");

        }

    }

}

**Event**: mỗi lần user hoạt động (gửi dữ liệu lên)

**State**: lưu lại timestamp của lần hoạt động cuối cùng

**Timer**: sau 10 giây kể từ lần hoạt động, nếu không có gì mới → cảnh báo

1. DataStream API: [Overview | Apache Flink](https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/datastream/overview/)

Immutable collections of data that can contain duplicates. This data can either be finite or unbounded. Can not simply inspect the elements inside but only work on them using the DataStream API operations, which are also called transformations. (Spark RDD Reference)

Using API methods such as map(), filter(), and so on.

Create an initial DataStream by adding a source

Example:

*// a stream of website clicks*

DataStream**<**Click**>** clicks **=** ...

DataStream**<**Tuple2**<**String, Long**>>** result **=** clicks

*// project clicks to userId and add a 1 for counting*

.map(

*// define function by implementing the MapFunction interface.*

**new** MapFunction**<**Click, Tuple2**<**String, Long**>>**() {

**@Override**

**public** Tuple2**<**String, Long**>** **map**(Click click) {

**return** Tuple2.of(click.userId, 1L);

}

})

*// key by userId (field 0)*

.keyBy(0)

*// define session window with 30 minute gap*

.window(EventTimeSessionWindows.withGap(Time.minutes(30L)))

*// count clicks per session. Define function as lambda function.*

.reduce((a, b) **->** Tuple2.of(a.f0, a.f1 **+** b.f1));

1. SQL / Table API

* Table API:

The [Table API](https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/table/overview/) follows the (extended) relational model: Tables have a schema attached (similar to tables in relational databases) and the API offers comparable operations, such as select, project, join, group-by, aggregate, etc.

Can seamlessly convert between tables and *DataStream*, allowing programs to mix the *Table API* with the *DataStream API*.

* SQL: similar to the *Table API* both in semantics and expressiveness, but represents programs as SQL query expressions. The [SQL](https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/table/overview/#sql) abstraction closely interacts with the Table API, and SQL queries can be executed over tables defined in the *Table API*.
* Table API:

Table eventsTable = tableEnv.fromDataStream(stream);

// Table API

Table result = eventsTable

    .filter($("score").isGreater(10))

    .window(Tumble.over(lit(10).seconds()).on($("timestamp")).*as*("w"))

    .groupBy($("user"), $("w"))

    .select($("user"), $("score").sum().*as*("total\_score"));

// Chuyển kết quả thành DataStream

DataStream<Row> resultStream = tableEnv.toDataStream(result);

resultStream.print();

* Flink SQL:

// Đăng ký DataStream thành Table

tableEnv.createTemporaryView("events", stream);

// SQL Query

String sqlQuery = "SELECT user, SUM(score) as total\_score " +

                  "FROM events " +

                  "WHERE score > 10 " +

                  "GROUP BY user, TUMBLE(timestamp, INTERVAL '10' SECOND)";

Table result = tableEnv.sqlQuery(sqlQuery);

// Chuyển kết quả thành DataStream

DataStream<Row> resultStream = tableEnv.toDataStream(result);

resultStream.print();