
Stream Juggler Platform

Release 1.1

Jan 09, 2018

| | | |
|------|---|-----|
| 1 | Tutorial | 3 |
| 2 | Creating Modules | 5 |
| 3 | API and UI documentation | 7 |
| 3.1 | Stream Juggler Platform Overview | 7 |
| 3.2 | Tutorial | 10 |
| 3.3 | SJ-Platform Architecture | 49 |
| 3.4 | Modules: Types, Structure, Pipeline | 54 |
| 3.5 | Engines: types, workflow | 70 |
| 3.6 | Custom Module Development Guide | 78 |
| 3.7 | Testing Modules on Simulators | 93 |
| 3.8 | Streams in SJ-Platform | 104 |
| 3.9 | Platform Deployment | 106 |
| 3.10 | UI Guide | 117 |
| 3.11 | REST API Guide | 141 |
| 3.12 | Glossary | 202 |
| 4 | Indices and tables | 205 |

The Stream Juggler Platform (SJ-Platform) is an open source, scalable solution for stream and micro-batched processing. The system fits for building both simple and complex event processing systems (CEP) and allows a developer to construct pipelines for analyzing data streams.

The Stream Juggler Platform uses Apache Mesos, Apache Kafka and T-streams to construct scalable and flexible processing algorithms. It enables exactly-once processing and provides an integrated solution with a RESTful interface, JavaScript UI and an ad hoc repository for modules, services, streams and other data processing pipeline components.

Thus, the SJ-Platform is a system that allows high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from different sources like Apache Kafka, or via TCP connections, and can be processed using complex algorithms. Finally, processed data can be pushed out to filesystems, external databases.

The documentation presented here corresponds to SJ-Platform Release 1.1.1. It gives a complete understanding of the system, its components, basic features fulfilled in it.

ГЛАВА 1

Tutorial

A detailed [Tutorial](#) provides real-life example tasks resolved with SJ-Platform as well as detailed platform deployment instructions.

Creating Modules

The [Custom Module Development Guide](#) section explains how to write a module using a simple hello-world example.

SJ-Platform provides REST API and Web UI for users to easily manage the platform. The [UI Guide](#) and the [REST API Guide](#) may be of utmost importance for platform administrators.

SJ-Platform 1.1.1 documentation general structure:

3.1 Stream Juggler Platform Overview

3.1.1 Introduction to Stream Juggler Platform

Stream Juggler Platform (SJ-Platform) is an open source, scalable solution for stream and micro-batch processing of unbounded data streams. The system fits for building an event processing systems and allows a developer to construct connected pipelines for handling data streams. A stream is an unbound sequence of events processed sequentially from the oldest ones to the newest ones. SJ-Platform is built to be smooth and easy to understand and learn for an average developer who knows Scala language. The system doesn't require any specific knowledge of mathematical concepts like some competing systems require. Thus, it makes it easy for a common type engineer to solve stream processing tasks.

Basically, SJ-Platform is inspired by [Apache Samza](#), but it has a lot of features which Samza doesn't provide, like exactly-once processing capability, integrated RESTful API and Web UI and lots of others.

Stream Processing Systems are widely used in the modern world. There are a lot of cases where developers should use stream processing systems. All those cases usually involve large data streams which can not be handled by a single computer effectively, specific requirements are applied to computations like idempotent processing, exactly-once processing and predictable behavior in case of crashes. Every stream processing platform is a framework which enforces certain code restrictions guaranteeing that the processing is stable and results are reproducible if a developer follows the restrictions.

There are many systems which can compete today - Apache Spark, Apache Flink, Apache Storm are the most famous. Every system has its own strengths and weaknesses, making it better or worse for certain cases. SJ-Platform also has such features. But we developed it to be universal and convenient for a broad range of tasks. We hope the features of SJ-Platform make developers solve and support tasks faster and system engineers operate clusters easier.

Every stream processing system must be fast and scalable. This is the most important requirement. SJ-Platform is fast and scalable as well. It is written in [Scala](#) language - well known JVM language which is fast and provides an access to lots of open-source libraries and frameworks. On the other hand, horizontal scaling is vital for stream processing systems which require the capability to distribute computations between compute nodes. SJ-Platform achieves that goal with the help of well-known distributed scheduler system - Apache Mesos.

SJ-Platform stands on shoulders of well-known technologies which simplify the deployment and operation and support best industrial practices. Core SJ-Platform technologies are mentioned in the following list:

1. [Apache Mesos](#) - a universal distributed computational engine;
2. [Apache Zookeeper](#) - a distributed configuration and a coordination broker;
3. [Apache Kafka](#) - a high-performance message broker;
4. [Mesosphere Marathon](#) - a universal framework for executing tasks on Mesos;
5. [MongoDB](#) - a highly available document database;
6. [Hazelcast](#) - a leading in-memory grid.

Further, in the documentation, we explain how, why and when technologies mentioned above are used in the system.

3.1.2 Documentation Structure

The documentation corresponds to the Stream Juggler Platform Release 1.1.1. It is organized in two big parts. The first one is a tutorial part which guides a reader through the system in a way which motivates him/her to observe, try and explain every step in practice. The second part is a referential one, which explains specific topics and lists system specifications for administrative and programming API, RESTful interface and Web UI.

3.1.3 Preliminary Requirements to The Reader

SJ-Platform is a quite complex system, but the tutorial tries to guide the reader as smooth as possible. So, there is quite a small amount of requirements to a reader. To achieve the success the reader must be familiar with:

1. scala programming language and generic data structures;
2. basics of Docker.

Also, the reader should have working Linux host with 4-8GB of RAM and 4 CPU cores with Docker installed (in the tutorial the installation of Docker for Ubuntu 16.04 OS will be explained).

In the tutorial, we will demonstrate the functionality of SJ-Platform, the cases for using modules of different types and their interoperation in a pipeline.

Two demo projects will train the reader to develop the modules for the platform using the example of solving issues, which are listed further:

- raw data transformation into a data stream,
- data processing, filtering, aggregation,
- result data storing and visualizing.

The demonstrated example tasks are not trivial but they are illustrative enough in terms of the platform usage. The first example is intended to collect the aggregated information on the accessibility of nodes. In this case, the data source is a fping sensor. The Elasticsearch is chosen as a data storage.

The second example performs data collecting from a sFlow reporter on network monitoring and traffic computation. The result is stored in PostgreSQL.

The problem is not a case of a “heavy” task but it includes some problems which are very specific to stream processing tasks and introduces all SJ-Platform functionality step-by-step without deep knowledge requirements of specific problem domains.

3.1.4 Short Features List for Impatient

Major features implemented in SJ-Platform are listed in the following list:

Processes data exactly-once. This is a very critical requirement which is important for many systems. SJ-Platform supports exactly-once processing mode in each module and across the pipeline.

Two kinds of processing - per-event and micro-batch. These modes are widely used and cover requirements of all stream processing tasks.

Stateful and stateless processing. Developers can use special state management API implementing their algorithms. That API supports resilient and reliable state behavior during system failures and crashes.

Distributed synchronized data processing. Micro-batch processing mode provides developers with the capability to synchronize computations across the nodes which is sometimes required.

Custom context-based batching methods. Micro-batch processing mode provides developers with API to implement custom algorithms to determine batch completeness which is important feature required in many real-life tasks.

Use of Apache Kafka, T-streams or TCP as an input source of events. External systems feed SJ-Platform with events via a list of supported interfaces. Right now it supports several of them.

The first is TCP. The method allows developers to design a custom protocol to receive events from external systems, deduplicate them and place into processing pipeline.

The second is Apache Kafka. Apache Kafka is the de-facto standard for message queueing, so we support it in SJ-Platform providing 3rd party applications with common integration interface.

The third is T-streams. T-streams is a Kafka-like message broker which is native to SJ-Platform and is used as internal data exchange bus inside the system.

JDBC/Elasticsearch/RESTful interface as an output destination for processing data. Processed data are exported to JDBC-compatible database, Elasticsearch or a datastore with the RESTful interface.

Performance metrics. SJ-Platform supports embedded performance metrics which help system managers to observe the runtime performance of the system.

Extensive simulator development framework. SJ-Platform provides developers with special “mock” infrastructure which helps to develop and test modules without actual deployment to the runtime.

These features will be explained in the documentation in depth.

To find more about the platform, please, visit the pages below:

[Tutorial](#) - a quick example to demonstrate the platform in action.

[SJ-Platform Architecture](#) - the architecture of the Stream Juggler Platform is presented, its components, connections between them, necessary services and other prerequisites for the Platform operation are described.

[Modules: Types, Structure, Pipeline](#) - more information on modules is given: what module types are supported in Stream Juggler Platform, how they work, etc.

[REST API Guide](#) - the REST API service is described here to work with the platform without the UI.

UI Guide - the section is devoted to the UI and its basic features to configure and monitor the platform.

3.2 Tutorial

Contents

- Tutorial
 - Introduction
 - SJ-Platform Overview
 - * Data Processing in SJ-Platform
 - Examples Introduction
 - Fping Example Task
 - * Step 1. Deployment
 - * Step 2. Configurations and Engine Jars Uploading
 - * Step 3. Module Uploading
 - * Step 4. Creating Streaming Layer
 - * Step 5. Create Output Destination
 - * Step 6. Creating Instances
 - * Launching Instances
 - * See the Results
 - * Customization
 - * Instance Shutdown
 - * Deleting Instance
 - Sflow Example Task
 - * Step 1. Deployment
 - * Step 2. Configurations and Engine Jars Uploading
 - * Step 3. Module Uploading
 - * Step 4. Creating Streaming Layer
 - * Step 5. Output SQL Tables Creation
 - * Step 6. Creating Instances
 - * Launching Instances
 - * See the Results
 - * Instance Shutdown
 - * Deleting Instances
 - More Information

3.2.1 Introduction

This tutorial is aimed to present SJ-Platform and give a quick start for a user to see the platform at work.

The demo projects presented below are example tasks introduced to demonstrate how to run user's first SJ module. A step-by-step guidance will help to deploy the system in a local mode (minimesos) or at a cluster (Mesos) and to implement SJ-Platform to a real-life task. A user will get to know the system structure, its key components and general concepts of the platform workflow.

3.2.2 SJ-Platform Overview

Stream Juggler Platform (SJ-Platform) provides a solution for stream and micro-batched processing of unbounded data streams. A processor transforms and processes data streams in SJ-Platform. Configurations uploaded to the system determine the mode of data processing in the pipeline. The result data are exported to an external storage.

To configure and monitor the system, SJ-Platform provides a user with a comprehensive RESTful API and Web UI.

A simplified structure of SJ-Platform can be presented as at the image below:

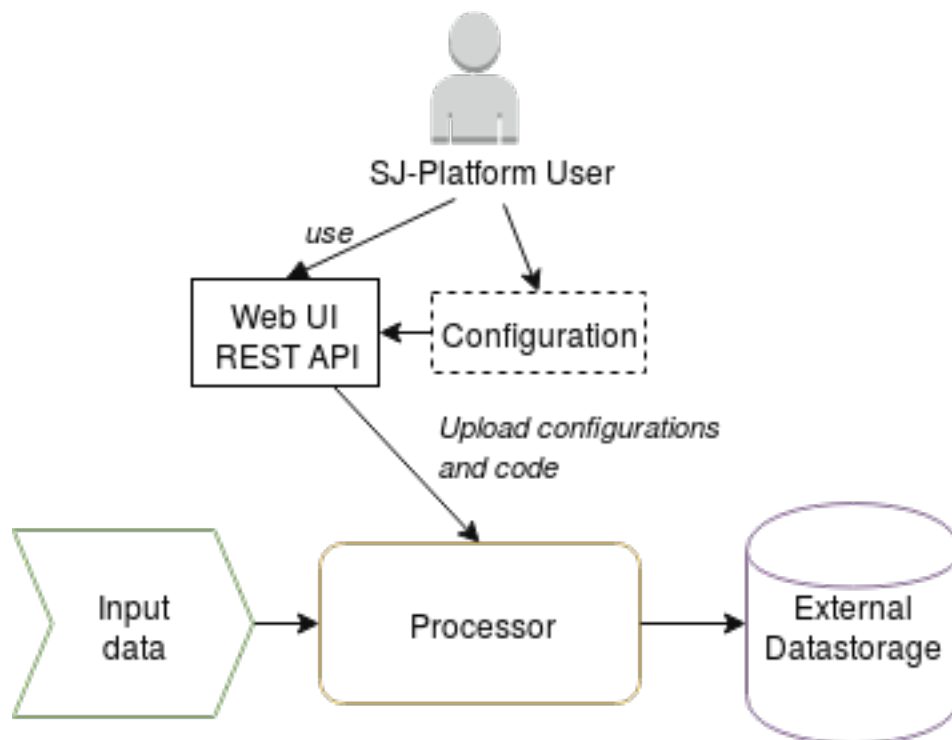


Рис. 3.1: Figure 1.1: General platform structure

Let's have a look at the platform from the perspective of a processing pipeline.

Data Processing in SJ-Platform

A processor can be represented by a processing module or a set of modules. Modules form a pipeline.

Alongside with a processing module the pipeline may include an input module that receives data and transforms them into streams, and an output module that exports the result.

General processing workflow which the system allows implementing is illustrated in the scheme below:

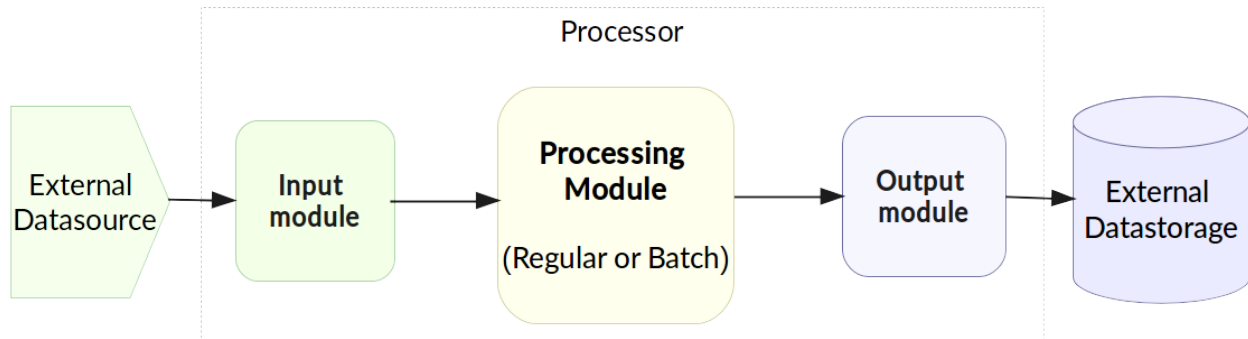


Рис. 3.2: Figure 1.2: General processing workflow

Green, yellow and purple blocks displayed in a rectangular area are managed and evaluated by SJ-Platform. They represent an input module, a processing module and an output module, respectively. The blocks outside the rectangular area represent external systems (a data source and a data storage).

The input module receives raw data and transforms them into a data stream of a proper type compatible with the processing module type. The processing module performs data aggregation, transformations, filtering and enriching, and sends the result to the output module. In the output module, the processed data are transformed into entities appropriate for storing into an external storage of a specified type. It can be Elasticsearch, RESTful endpoint or JDBC-compatible data storages. The illustrated pipeline is a common scenario for a lot of different tasks.

But the platform allows implementation of more complicated processing pipelines. So the pipeline can be extended. Several input modules can be included in the pipeline to receive the raw data and transform them for passing further to the processing stage. You can launch more than a single processing module. Data streams can be distributed among them in various ways. A few output modules may receive processed data and put them into a storage/storages. This case is described in the [Sflow Example Task](#).

To configure and monitor the system, SJ-Platform provides a user with a comprehensive RESTful API and Web UI. Further we will go through a couple of real-life tasks to demonstrate the platform workflow. It will help you to understand how the platform processes data. This tutorial will provide you with a ready-to-use problem solution on the base of SJ-Platform. Perform the steps for the example tasks to get acquainted with the platform functionality.

If you would like to continue studying the platform, proceed with reading the documentation. There you will find instructions on development, deployment and customization of your own code for your specific aims.

3.2.3 Examples Introduction

The example tasks that will be presented are different. But the steps we will perform to solve the tasks are common for both of them (see Figure 1.3).

What we are going to do for the examples is:

1. Deploy Mesos and other services. We suggest deploying the platform to Mesos using Marathon. Among other services we will run:
 - Apache Zookeeper - for coordination of task execution;

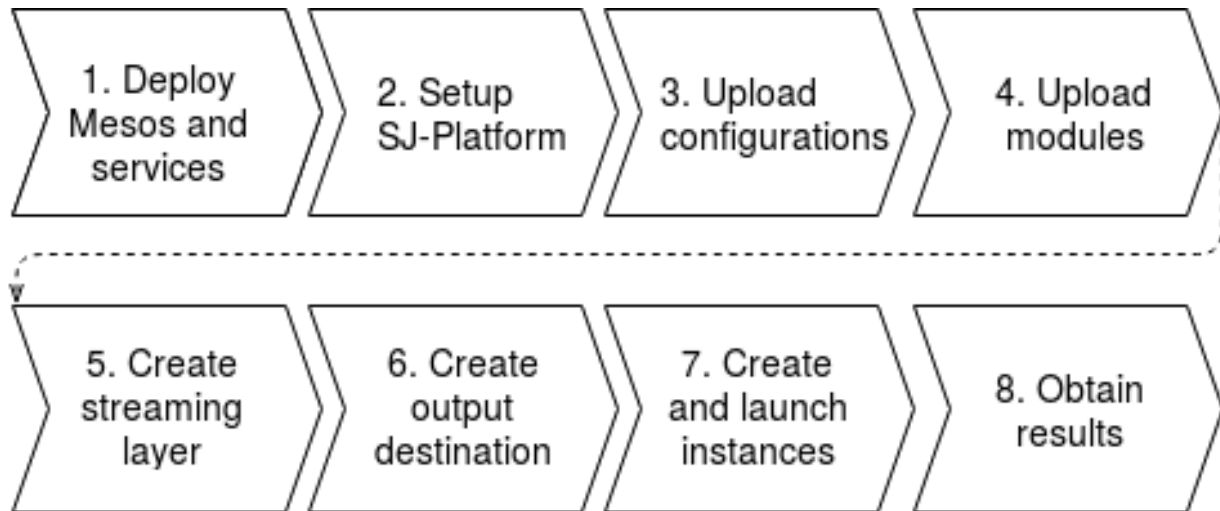


Рис. 3.3: Figure 1.3: Example task solution steps

- Java - a computer software that provides a system for developing application software and deploying it in a cross-platform computing environment;
 - Docker - a software container platform that allows a flexible system configuration;
 - MongoDB - as a database;
 - T-streams - as a message broker ensuring exactly-once data processing;
 - RESTful API - for accessing and monitoring the platform;
 - Elasticsearch, PostgreSQL - as external data storages;
 - Kibana - to visualize Elasticsearch data.
2. Download and set up the platform and demo project. We'll set up the platform and the demo repositories downloading it from GitHub.
 3. Upload configurations and engines. The configurations should be uploaded to determine the system work. The full list of all configurations can be viewed at the [Configuration](#) page.

Engines are necessary for modules as they handle data flow making it into streams.

An engine is required to start a module. A module can not process data without an engine.

It is a base of the system that provides the I/O functionality. It uses module settings for data processing. We will upload an engine jar file per each module in a pipeline.

Note: Find more about engines at the [Engines: types, workflow](#) page.

4. Upload modules. Module is a program module processing data streams. For a module we assemble a JAR file, containing a module specification. Module's executor performs data transformation, aggregation, filtering.

In the example tasks we will upload ready-to-use modules of three types - input modules, processing modules (regular, batch) and output modules.

To solve your tasks, you may upload your custom modules in this step.

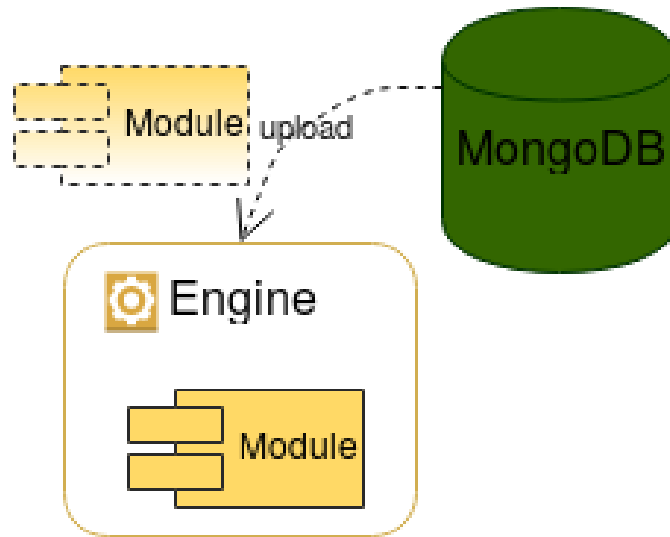


Рис. 3.4: Figure 1.4: An engine in SJ-Platform

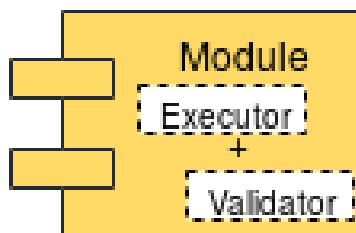


Рис. 3.5: Figure 1.5: Module structure

Note: Find more about modules at the [Modules: Types, Structure, Pipeline](#) page. A hello-world on a custom module can be found at the [Custom Module Development Guide](#) section.

5. Create the streaming layer. Modules exchange data via streams. Within the platform, T-streams are used for message transportation allowing exactly-once data exchange. The result data are exported from SJ-Platform to an external storage with streams of types corresponding to the type of that storage: Elasticsearch, SQL database or RESTful.

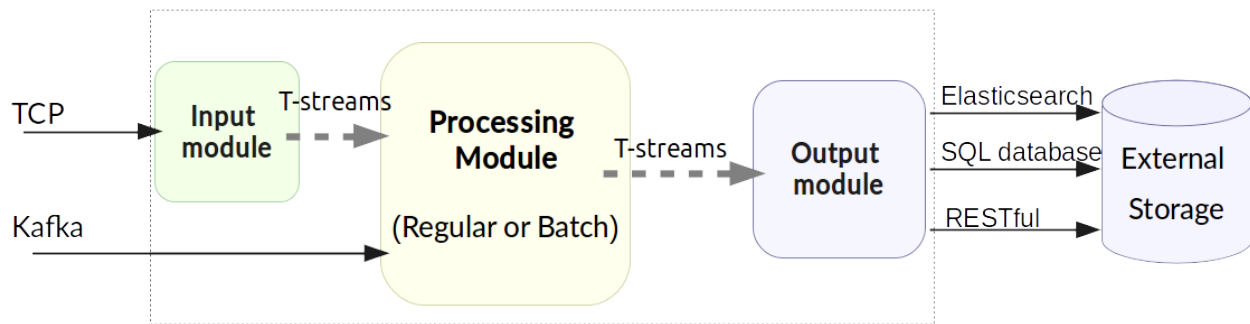


Рис. 3.6: Figure 1.6: Srteams in the system

Streaming requires the infrastructure: providers and services. These are necessary API entities without which streaming will not be so flexible. Streaming flexibility lies in the one-to-many connection between providers and services, streams and modules. One provider works with many services (of various types). One type of streams can be used by different module instances. These streams take necessary settings from the common infrastructure (providers and services). There is no need to duplicate the settings for each individual stream.

For both example tasks we will need Apache Zookeeper, Elasticsearch and SQL-database types of providers, and Apache Zookeeper, Elasticsearch, SQL-database and T-streams types of services. On the base of the infrastructure we will create streams of corresponding types.

Note: Find more about streams and the streaming infrastructure at the [Streams in SJ-Platform](#) section.

6. Create output destination. At this step all necessary tables and mapping should be created for storing the processed result in an external data storage.
7. Create and launch instances. For each module we will create instances. It is a set of settings determining collaborative work of an engine and a module.

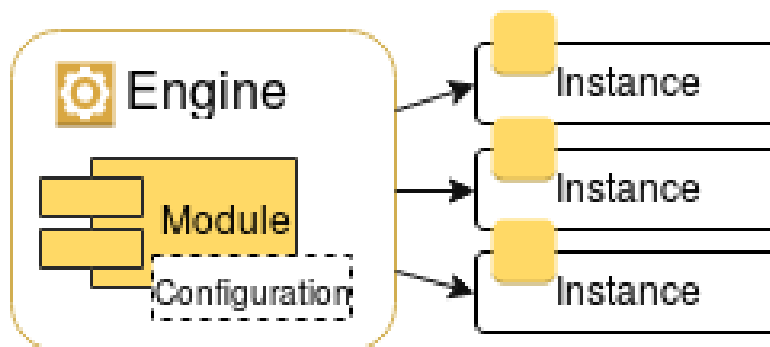


Рис. 3.7: Figure 1.7: Instance in the platform

An instance is created with specific parameters. It will use particular streams specified for it as input and output streams.

Launching instances we will start data processing in the platform.

8. Obtain and store the result. The result of processing will be saved to an external storage. Besides, in the `fping` example we will visualise resulting data using Kibana.

Now as you have general idea of the workscope to do, let's dive into the example tasks.

3.2.4 Fping Example Task

The first example task we'd like to introduce illustrates the platform workflow in the real-world use.

The issue we are going to solve using our platform is to collect aggregated information on the accessibility of nodes using `fping` utility. It checks accessibility of provided IPs sending a 64-bytes packet to each IP and waiting for a return packet. If the node can be accessed, a good return packet will be received. Also it returns the amount of time needed for a package to reach the node and return back. On the basis of this information the processor calculates the average response time for each node per 1 minute. The amount of successful responses by IP per 1 minute is calculated by the processing module as well. The result is exported to an external data storage.

In the example task solution the processing workflow is formed in the following way:

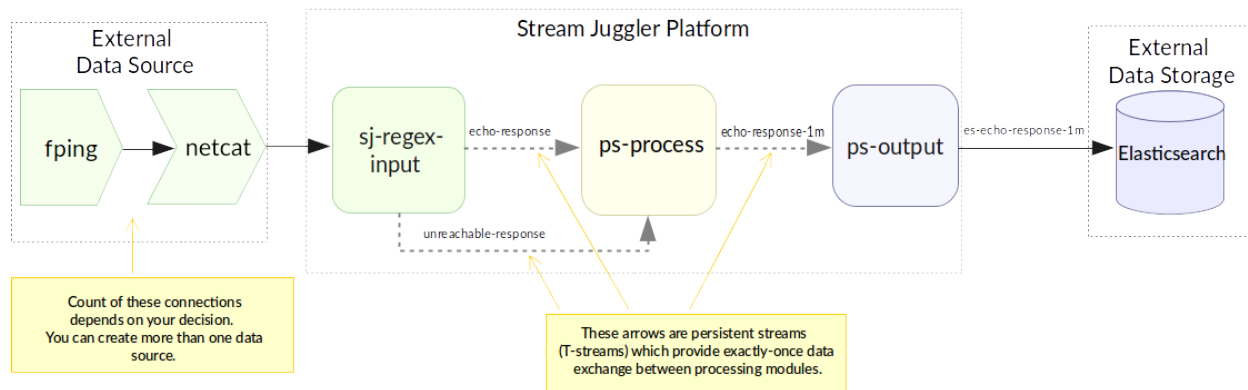


Рис. 3.8: Figure 1.8: Fping example task pipeline

This diagram demonstrates the processing workflow for the example. Green, yellow and purple blocks are executed with SJ-Platform. They are an input module, a processing module and an output module.

The data come to a TCP input module through a pipeline of `fping` and `netcat`. The TCP input module is an input module that performs per-event processing. For the `fping` example task we will use one of two TCP input modules provided by SJ-Platform - a regex input module. It processes an input stream which contains text data using a set of regular expressions, and then serializes them with Apache Avro.

Note: We provide two off-the-shelf input modules - CSV and regex - for two most general input data formats. Find more information about them at the [Modules of Input Type](#) section.

Then the input module parses ICMP echo responses (IP and response time are selected) and ICMP unreachable responses (IPs only are selected) and puts the parsed data into 'echo-response' stream and 'unreachable-response' stream, respectively.

After that, the instance of a processing module aggregates response time and a total amount of echo/unreachable responses by IP per 1 minute and sends aggregated data to 'echo-response-1m' stream. In the fping demonstration example the data aggregation is performed with the processing module of a regular-streaming type.

In the [Customization](#) section we will tell you how to add more instances to the processing module for calculating responses per a different period of time - per 3 minutes and per 1 hour. It is an optional step which allows adjusting the system to a specific aim and making calculations for different periods of time.

Finally, the output module exports aggregated data from echo-response streams to Elasticsearch. The result is visualized using Kibana.

The data are fed to the system, passed from one module to another and exported from the system via streams.

Platform entities can be created via Web UI filling up all needed fields in corresponding forms. In the demonstration task, we suggest adding the entities to the system via REST API as it is the easiest and quickest way. You can use Web UI to see the created entities.

Now, having the general idea on the platform workflow, we can dive into solving an example task on the base of SJ-Platform.

And we start with the system deployment.

Step 1. Deployment

Though SJ-Platform is quite a complex system and it includes a range of services to be deployed, no special skills are required for its setting up.

There are two options to deploy the platform. Please, read the description for each option and choose the most convenient for you.

Option 1. The easiest way to start SJ-Platform is prebuilding [a virtual box image](#). This is the most rapid way to get acquainted with the platform and assess its performance.

We suggest deploying the platform locally via Vagrant with VirtualBox as a provider. It takes up to 30 minutes.

Minimum system requirements in this case are as follows:

- At least 8 GB of free RAM;
- VT-x must be enabled in BIOS;
- Vagrant 1.9.1 installed;
- VirtualBox 5.0.40 installed.

These requirements are provided for deployment on Ubuntu 16.04 OS.

The platform is deployed with all entities necessary to demonstrate the solution for the example task: providers, services, streams, configurations. So the instructions below for creating entities can be omitted. You may read about platform components here in the deployment steps (Step 1 - Step 6) and see the result in the UI.

Option 2. Another option is to deploy the platform on a cluster. Currently, the deployment on [Mesos](#) as a universal distributed computational engine is supported.

Minimum system requirements in this case are as follows:

- working Linux host with 4-8 GB of RAM and 4 CPU cores;
- Docker installed (see [official documentation](#));

- cURL installed;
- sbt installed (see [official documentation](#)).

The platform is deployed with no entities. Thus, the pipeline should be built from scratch.

This tutorial provides step-by-step instructions to deploy the demo project to Mesos using Marathon. At first step, Mesos with all the services will be deployed. Then entities will be created for the platform. Finally, modules will be launched and results will be visualised using Kibana.

For the example task we provide instructions to deploy the platform to Mesos using Marathon.

The deployment is performed via REST API.

So, let's start with deploying Mesos and other services.

1. Deploy Mesos, Marathon, Zookeeper. You can follow the instructions at the official [installation guide](#) .

To deploy Docker follow the instructions at the official [installation guide](#) .

Install Java 1.8. Find detailed instructions [here](#).

Please, note, the deployment described here is for one default Mesos-slave with available ports [31000-32000]. Mesos-slave must support Docker containerizer. The technical requirements to Mesos-slave are the following:

- 2 CPUs,
- 4096 memory.

Start Mesos and the services.

Note: If you are planning to process data in a parallel mode (set the parallelizm parameter to a value greater than 1), you need to increase the `executor_registration_timeout` parameter for Mesos-slave.

2. Create JSON files and a configuration file. Please, name them as specified here.

Replace `<slave_advertise_ip>` with Mesos-slave IP.

Replace `<zk_ip>` and `<zk_port>` according to the Apache Zookeeper address.

mongo.json:

```
{
  "id": "mongo",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "mongo:3.4.7",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 27017,
          "hostPort": 31027,
          "protocol": "tcp"
        }
      ]
    },
    "parameters": [
      {
        "key": "restart",
        "value": "always"
      }
    ]
  }
}
```

```

    }
  },
  "instances":1,
  "cpus":0.1,
  "mem":512
}

```

sj-rest.json:

```

{
  "id":"sj-rest",
  "container":{
    "type":"DOCKER",
    "docker":{
      "image":"bwsj/sj-rest:dev",
      "network":"BRIDGE",
      "portMappings":[
        {
          "containerPort":8080,
          "hostPort":31080,
          "protocol":"tcp"
        }
      ],
      "parameters":[
        {
          "key":"restart",
          "value":"always"
        }
      ]
    }
  },
  "instances":1,
  "cpus":0.1,
  "mem":1024,
  "env":{
    "MONGO_HOSTS":"<slave_advertise_ip>:31027",
    "ZOOKEEPER_HOST":"<zk_ip>",
    "ZOOKEEPER_PORT":"<zk_port>"
  }
}

```

elasticsearch.json:

```

{
  "id":"elasticsearch",
  "container":{
    "type":"DOCKER",
    "docker":{
      "image":"docker.elastic.co/elasticsearch/elasticsearch:5.5.1",
      "network":"BRIDGE",
      "portMappings":[
        {
          "containerPort":9200,
          "hostPort":31920,
          "protocol":"tcp"
        },
        {
          "containerPort":9300,

```

```
        "hostPort":31930,
        "protocol":"tcp"
      }
    ],
    "parameters":[
      {
        "key":"restart",
        "value":"always"
      }
    ]
  }
},
"env":{
  "ES_JAVA_OPTS":"-Xms256m -Xmx256m",
  "http.host":"0.0.0.0",
  "xpack.security.enabled":"false",
  "transport.host":"0.0.0.0",
  "cluster.name":"elasticsearch"
},
"instances":1,
"cpus":0.2,
"mem":256
}
```

config.properties:

```
key=pingstation
active.tokens.number=100
token.ttl=120

host=0.0.0.0
port=8080
thread.pool=4

path=/tmp
data.directory=transaction_data
metadata.directory=transaction_metadata
commit.log.directory=commit_log
commit.log.rocks.directory=commit_log_rocks

berkeley.read.thread.pool = 2

counter.path.file.id.gen=/server_counter/file_id_gen

auth.key=dummy
endpoints=127.0.0.1:31071
name=server
group=group

write.thread.pool=4
read.thread.pool=2
ttl.add-ms=50
create.if.missing=true
max.background.compactions=1
allow.os.buffer=true
compression=LZ4_COMPRESSION
use.fsync=true
```



```

zk.endpoints=<zk_ip>
zk.prefix=/pingstation
zk.session.timeout-ms=10000
zk.retry.delay-ms=500
zk.connection.timeout-ms=10000

max.metadata.package.size=100000000
max.data.package.size=100000000
transaction.cache.size=300

commit.log.write.sync.value = 1
commit.log.write.sync.policy = every-nth
incomplete.commit.log.read.policy = skip-log
commit.log.close.delay-ms = 200
commit.log.file.ttl-sec = 86400
stream.zookeeper.directory=/tts/tstreams

ordered.execution.pool.size=2
transaction-database.transaction-keep-time-min=70000
subscribers.update.period-ms=500

```

tts.json (replace <path_to_conf_directory> with an appropriate path to the configuration directory on your computer and <external_host> with a valid host):

```

{
  "id": "tts",
  "container": {
    "type": "DOCKER",
    "volumes": [
      {
        "containerPath": "/etc/conf/config.properties",
        "hostPath": "<path_to_conf_directory>",
        "mode": "RO"
      }
    ],
    "docker": {
      "image": "bws/tstreams-transaction-server",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 8080,
          "hostPort": 31071,
          "protocol": "tcp"
        }
      ],
      "parameters": [
        {
          "key": "restart",
          "value": "always"
        }
      ]
    }
  },
  "instances": 1,
  "cpus": 0.1,
  "mem": 512,
  "env": {
    "HOST": "<slave_advertise_ip>",

```

```
"PORT0":"31071"
}
}
```

kibana.json:

```
{
  "id":"kibana",
  "container":{"
    "type":"DOCKER",
    "docker":{"
      "image":"kibana:5.5.1",
      "network":"BRIDGE",
      "portMappings":[
        {
          "containerPort":5601,
          "hostPort":31561,
          "protocol":"tcp"
        }
      ],
      "parameters":[
        {
          "key":"restart",
          "value":"always"
        }
      ]
    }
  },
  "instances":1,
  "cpus":0.1,
  "mem":256,
  "env":{"
    "ELASTICSEARCH_URL":"https://<slave_advertise_ip>:31920"
  }
}
```

3. Run the services on Marathon:

Mongo:

```
curl -X POST http://172.17.0.1:8080/v2/apps -H "Content-type: application/json" -d @mongo.json
```

Elasticsearch:

Please, note that command should be executed on Master-slave machine:

```
sudo sysctl -w vm.max_map_count=262144
```

Then launch Elasticsearch:

```
curl -X POST http://172.17.0.1:8080/v2/apps -H "Content-type: application/json" -d
@elasticsearch.json
```

SJ-rest:

```
curl -X POST http://172.17.0.1:8080/v2/apps -H "Content-type: application/json" -d @sj-rest.json
```

T-Streams:

```
curl -X POST http://172.17.0.1:8080/v2/apps -H "Content-type: application/json" -d @tts.json
```

Kibana:

```
curl -X POST http://172.17.0.1:8080/v2/apps -H "Content-type: application/json" -d @kibana.json
```

Via the Marathon interface, make sure the services have a running status.

4. Add the credential settings if Mesos requires the framework must be authenticated:

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name": \"framework-principal\", \"value\": <principal>, \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name": \"framework-secret\", \"value\": <secret>, \"domain\": \"configuration.system\"}"
```

Now make sure you have access to the Web UI. You will see the platform is deployed but there are no entities yet created. We will create them in next steps.

Step 2. Configurations and Engine Jars Uploading

To implement the processing workflow for the example task resolution the following JAR files should be uploaded:

1. a JAR file per each module type: input-streaming, regular-streaming, output-streaming;
2. a JAR file for Mesos framework that starts engines.

Thus, engines should be compiled and uploaded next.

Upload Engine Jars

Please, download the engines' JARs for each module type (input-streaming, regular-streaming, output-streaming) and the Mesos framework:

```
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-mesos-framework.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-input-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-regular-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-output-streaming-engine.jar
```

Now upload the engines' JARs. Please, replace <slave_advertise_ip> with the Mesos-slave IP:

```
address=<slave_advertise_ip>:31080

curl --form jar=@sj-mesos-framework.jar http://$address/v1/custom/jars
curl --form jar=@sj-input-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-regular-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-output-streaming-engine.jar http://$address/v1/custom/jars
```

Now the JARs should appear in the UI under Custom Jars of the “Custom files” navigation tab.

Setup Configurations for Engines

For the example task, we upload the following configurations via REST:

Stream Juggler / Custom files

| Name | Version | Upload date | Size | Actions |
|-----------------------------------|---------|------------------------------|-------|---------|
| com.bwsw.fw | 1.0 | Wed Aug 09 03:45:30 UTC 2017 | 94 MB | |
| com.bwsw.input.streaming.engine | 1.0 | Wed Aug 09 03:45:38 UTC 2017 | 97 MB | |
| com.bwsw.output.streaming.engine | 1.0 | Wed Aug 09 03:45:43 UTC 2017 | 97 MB | |
| com.bwsw.regular.streaming.engine | 1.0 | Wed Aug 09 04:03:38 UTC 2017 | 88 MB | |

Рис. 3.9: Figure 1.9: Uploaded engines list

- session.timeout - use when connect to Apache Zookeeper (ms). Usually when we are dealing with T-streams consumers/producers and Apache Kafka streams.
- current-framework - indicates which file is used to run a framework. By this value, you can get a setting that contains a file name of framework jar.
- crud-rest-host - REST interface host.
- crud-rest-port - REST interface port.
- marathon-connect - Marathon address. Use to launch a framework that is responsible for running engine tasks and provides the information about launched tasks. It should start with 'http://'.
- marathon-connect-timeout - use when trying to connect by 'marathon-connect' (ms).

Send the next requests to upload the configurations. Please, replace <slave_advertise_ip> with the slave advertise IP and <marathon_address> with the address of Marathon:

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"session.timeout\", \"value\": \"7000\", \"domain\": \"configuration.apache-zookeeper\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"current-framework\", \"value\": \"com.bwsw.fw-1.0\", \"domain\": \"configuration.system\"}"

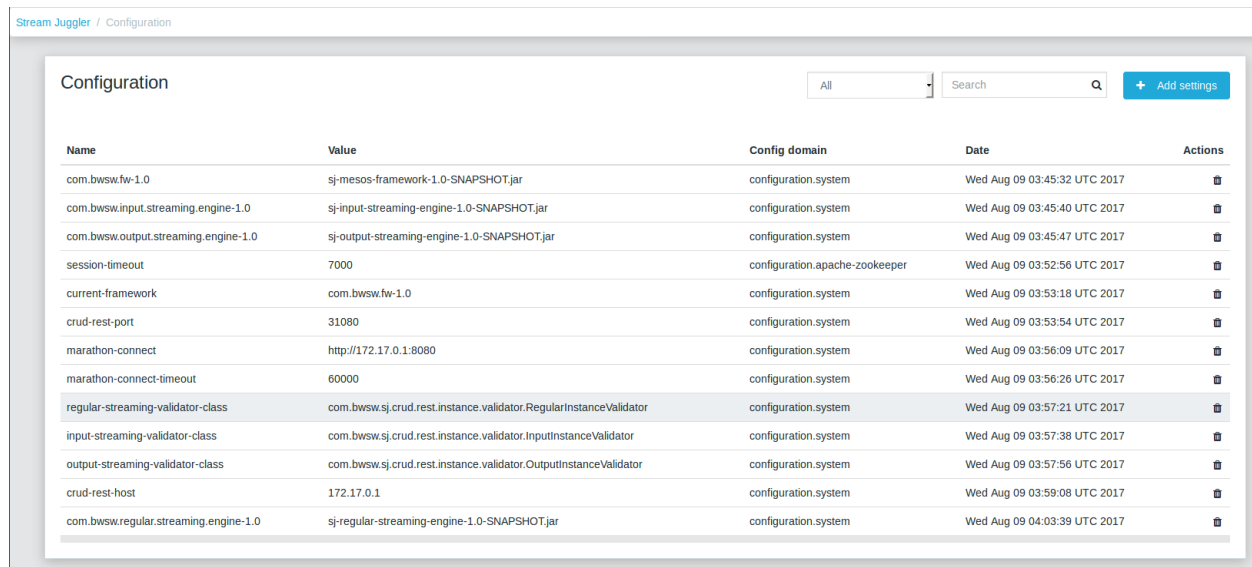
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"crud-rest-host\", \"value\": \"<slave_advertise_ip>\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"crud-rest-port\", \"value\": \"31080\", \"domain\": \"configuration.system\"}"

curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"marathon-connect\", \"value\": \"http://<marathon_address>\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"marathon-connect-timeout\", \"value\": \"60000\", \"domain\": \"configuration.system\"}"
```

Send the next requests to upload configurations for instance validators:

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"regular-streaming-validator-class\", \"value\": \"com.bwsw.sj.crud.rest.instance.validator.RegularInstanceValidator\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"input-streaming-validator-class\", \"value\": \"com.bwsw.sj.crud.rest.instance.validator.InputInstanceValidator\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"output-streaming-validator-class\", \"value\": \"com.bwsw.sj.crud.rest.instance.validator.OutputInstanceValidator\", \"domain\": \"configuration.system\"}"
```

In the UI you can see the uploaded configurations under the “Configuration” tab of the main navigation bar.



The screenshot shows the 'Configuration' tab in the Stream Juggler UI. It features a table with columns: Name, Value, Config domain, Date, and Actions. The table lists various system configurations, including framework versions, streaming engines, session timeouts, and validator classes. A search bar and an 'Add settings' button are located at the top right of the configuration list.

| Name | Value | Config domain | Date | Actions |
|---------------------------------------|---|--------------------------------|------------------------------|---------|
| com.bwsw.fw-1.0 | sj-mesos-framework-1.0-SNAPSHOT.jar | configuration.system | Wed Aug 09 03:45:32 UTC 2017 | |
| com.bwsw.input.streaming.engine-1.0 | sj-input-streaming-engine-1.0-SNAPSHOT.jar | configuration.system | Wed Aug 09 03:45:40 UTC 2017 | |
| com.bwsw.output.streaming.engine-1.0 | sj-output-streaming-engine-1.0-SNAPSHOT.jar | configuration.system | Wed Aug 09 03:45:47 UTC 2017 | |
| session-timeout | 7000 | configuration.apache-zookeeper | Wed Aug 09 03:52:56 UTC 2017 | |
| current-framework | com.bwsw.fw-1.0 | configuration.system | Wed Aug 09 03:53:18 UTC 2017 | |
| crud-rest-port | 31080 | configuration.system | Wed Aug 09 03:53:54 UTC 2017 | |
| marathon-connect | http://172.17.0.1:8080 | configuration.system | Wed Aug 09 03:56:09 UTC 2017 | |
| marathon-connect-timeout | 60000 | configuration.system | Wed Aug 09 03:56:26 UTC 2017 | |
| regular-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.RegularInstanceValidator | configuration.system | Wed Aug 09 03:57:21 UTC 2017 | |
| input-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.InputInstanceValidator | configuration.system | Wed Aug 09 03:57:38 UTC 2017 | |
| output-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.OutputInstanceValidator | configuration.system | Wed Aug 09 03:57:56 UTC 2017 | |
| crud-rest-host | 172.17.0.1 | configuration.system | Wed Aug 09 03:59:08 UTC 2017 | |
| com.bwsw.regular.streaming.engine-1.0 | sj-regular-streaming-engine-1.0-SNAPSHOT.jar | configuration.system | Wed Aug 09 04:03:39 UTC 2017 | |

Рис. 3.10: Figure 1.10: Uploaded configurations list

Step 3. Module Uploading

Now as the system is deployed and necessary engines are added, modules can be uploaded to the system.

For the stated example task we upload the following modules:

- a TCP input module - sj-regex-input module - that accepts TCP input streams and transforms raw data to put them to T-streams and transmit for processing;
- a processing module - ps-process module - which is a regular-streaming module that processes data element-by-element.
- an output module - ps-output module - that exports resulting data to Elasticsearch.

Please, follow these steps to upload the modules.

First, copy the example task repository from GitHub:

```
git clone https://github.com/bwsw/sj-fping-demo.git
cd sj-fping-demo
sbt assembly
```

Then download the sj-regex-input module from Sonatype repository:

```
curl "https://oss.sonatype.org/content/repositories/snapshots/com/bwsw/sj-regex-input_2.12/1.0-SNAPSHOT/sj-regex-input_2.12-1.0-SNAPSHOT.jar" -o sj-regex-input.jar
```

Upload modules

Upload three modules to the system:

```
curl --form jar=@sj-regex-input.jar http://$address/v1/modules
curl --form jar=@ps-process/target/scala-2.11/ps-process-1.0.jar http://$address/v1/modules
curl --form jar=@ps-output/target/scala-2.11/ps-output-1.0.jar http://$address/v1/modules
```

Now in the UI, you can see the uploaded modules under the ‘Modules’ tab.

Stream Juggler / Modules

Modules list

All

Upload module

| Name | Type | Version | Size | Upload date | Actions |
|---------------------|-------------------|---------|-------|------------------------------|---------|
| com.bsw.input.regex | input-streaming | 1.0 | 34 KB | Wed Aug 09 04:07:59 UTC 2017 | |
| pingstation-process | regular-streaming | 1.0 | 32 KB | Wed Aug 09 04:11:44 UTC 2017 | |
| pingstation-output | output-streaming | 1.0 | 10 KB | Wed Aug 09 04:11:52 UTC 2017 | |

Рис. 3.11: Figure 1.11: Uploaded modules list

Step 4. Creating Streaming Layer

The raw data are fed to the platform from different sources. And within the platform, the data are transported to and from modules via streams. Thus, in the next step, the streams for data ingesting and exporting will be created.

Different modules require different stream types for input and output.

In the example task solution the following stream types are implemented:

1. TCP input stream feed the raw data into the system;
2. T-streams streaming passes the data to and from the processing module;
3. output modules export aggregated data and transfer them in streams to Elasticsearch.

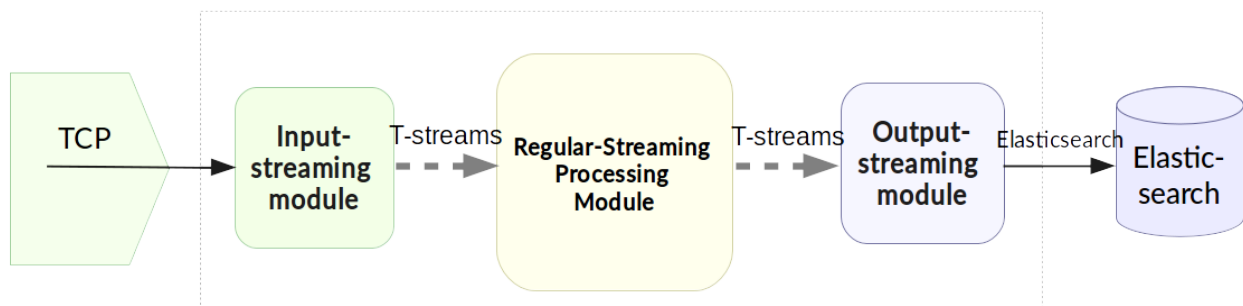


Рис. 3.12: Figure 1.12: Streams in Fping example pipeline

Prior to creating a stream, we need to create infrastructure for the streaming layer. The infrastructure for streams includes providers and services. This is a required presetting.

The types of providers and services are determined by the type of streams. Find more about types of providers and services at the [Streaming Infrastructure](#) section.

Perform the steps below to create streaming infrastructure: providers, services, and streams.

Set Up Streaming Infrastructure

At this step we will create the infrastructure: providers and services.

In the example task pipeline the modules of three types take place - input-streaming, regular-streaming and output-streaming. For all types of modules, the Apache Zookeeper service is necessary. Thus, it is required to create the Apache Zookeeper provider.

Besides, the Apache Zookeeper provider is required for T-streams service. T-streams service is in its turn needed for streams of T-streams type within the system, and for instances of the input-streaming and the regular-streaming modules.

The provider and the service of Elasticsearch type are required by the Elasticsearch output streams to put the result into the Elasticsearch data storage.

As a result, we have the following infrastructure to be created:

- Providers of Apache Zookeeper and Elasticsearch types;
- Services of Apache Zookeeper, T-streams and Elasticsearch types.

1. Set up providers.

Before sending a request, please, note there is a default value of Elasticsearch IP (176.120.25.19) in json configuration files. So we need to change it appropriately via sed app before using.

- Create Apache Zookeeper provider for ‘echo-response’ and ‘unreachable-response’ T-streams used within the platform, as well as for Apache Zookeeper service required for all types of instances:

```
sed -i 's/176.120.25.19:2181/<zookeeper_address>/g' api-json/providers/zookeeper-ps-provider.json
curl --request POST "http://$address/v1/providers" -H 'Content-Type: application/json' --data "@api-
  ↪json/providers/zookeeper-ps-provider.json"
```

- Create Elasticsearch provider for output streaming (all ‘es-echo-response’ streams):

```
sed -i 's/176.120.25.19/elasticsearch.marathon.mm/g' api-json/providers/elasticsearch-ps-provider.json
curl --request POST "http://$address/v1/providers" -H 'Content-Type: application/json' --data "@api-
  ↪json/providers/elasticsearch-ps-provider.json"
```

The created providers are available in the UI under the “Providers” tab.

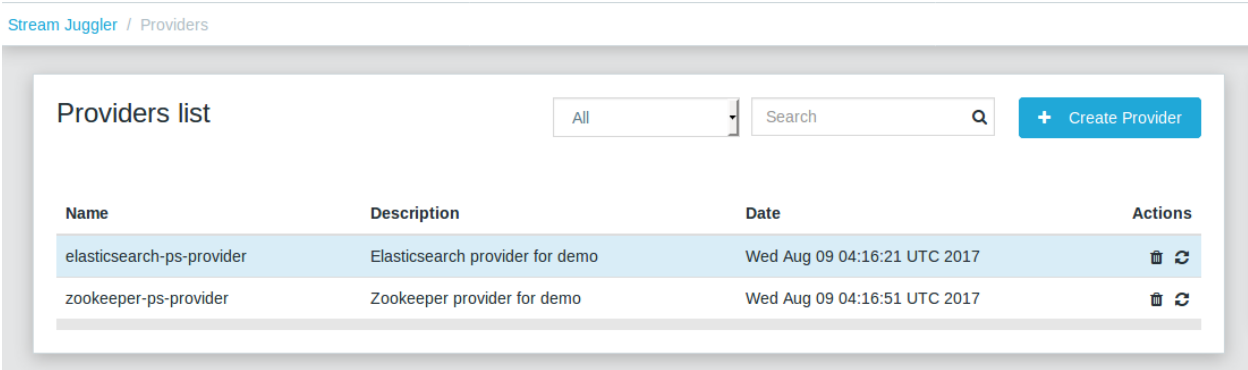


Рис. 3.13: Figure 1.13: Providers list

2. Next, we will set up services:

- Apache Zookeeper service for all modules:

```
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/
↪services/zookeeper-ps-service.json"
```

- T-streams service for T-streams (all ‘echo-response’ streams and the ‘unreachable-response’ stream) within the system and for the instances of the input-streaming and the regular-streaming modules:

```
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/
↪services/tstream-ps-service.json"
```

- Elasticsearch service for output streams (all ‘es-echo-response’ streams) and the output-streaming module:

```
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/
↪services/elasticsearch-ps-service.json"
```

Please, make sure the created services have appeared in the UI under the “Services” tab.

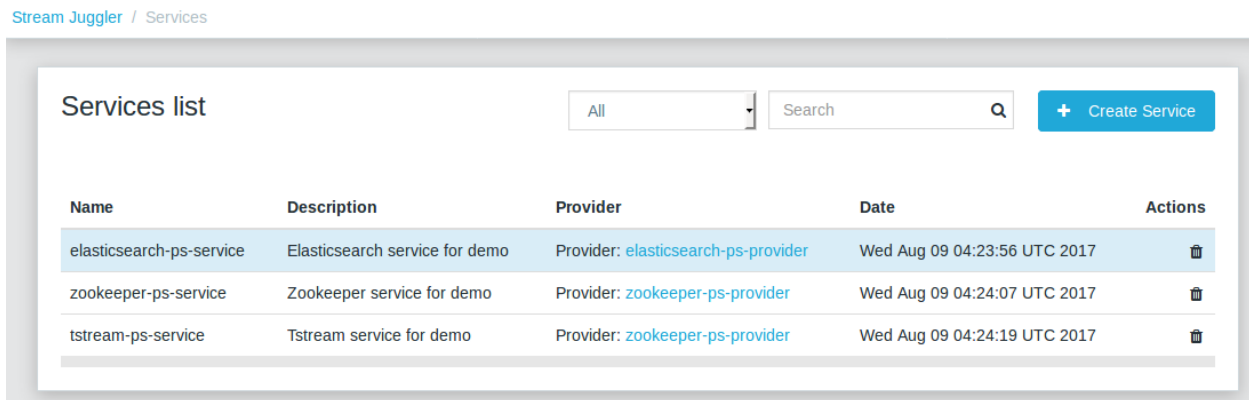


Рис. 3.14: Figure 1.14: Services list

Creating Streams

Once the infrastructure is ready, it is time to create streams.

For sj-regex-input module:

Create an ‘echo-response’ output stream of the input-streaming module (consequently, an input stream of the regular-streaming module). It will be used for keeping an IP and average time from ICMP echo-response and also a timestamp of the event:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/
↪streams/echo-response.json"
```

Create one more output stream - an ‘unreachable response’ output stream - of the input-streaming module. It will be used for keeping an IP from ICMP unreachable response and also a timestamp of the event:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/
↪streams/unreachable-response.json"
```

These streams are of T-streams type.

For ps-process module:

Create an output stream of the regular-streaming module (consequently, an input stream of the output-streaming module) named 'echo-response-1m'. It will be used for keeping the aggregated information about the average time of echo responses, the total amount of echo responses, the total amount of unreachable responses and the timestamp for each IP (per 1 minute):

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/echo-response-1m.json"
```

This stream is of the T-streams type.

For ps-output module:

Create an output stream of the output-streaming module named 'es-echo-response-1m'. It will be used for keeping the aggregated information (per 1 minute) from the previous stream including total amount of responses:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/es-echo-response-1m.json"
```

This stream is of the Elasticsearch type (as the external storage in the pipeline is Elasticsearch).

All the created streams should be available now in the UI under the "Streams" tab.

Stream Juggler / Streams

| Name | Description | Service | Date | Actions |
|----------------------|--------------------|---|------------------------------|---------|
| echo-response | Tstream for demo | Service: tstream-ps-service | Thu Aug 10 04:35:10 UTC 2017 | |
| unreachable-response | Tstream for demo | Service: tstream-ps-service | Thu Aug 10 04:36:18 UTC 2017 | |
| echo-response-1m | Tstream for demo | Service: tstream-ps-service | Thu Aug 10 04:36:34 UTC 2017 | |
| es-echo-response-1m | ES stream for demo | Service: elasticsearch-ps-service | Thu Aug 10 04:36:48 UTC 2017 | |

Рис. 3.15: Figure 1.15: Streams list

Step 5. Create Output Destination

At this step all necessary indexes, tables and mapping should be created for storing the processed result.

In the provided example task the result data are saved to the Elasticsearch data storage.

Thus, it is necessary to create the index and mapping for Elasticsearch.

Create the index and the mapping for Elasticsearch sending the PUT request:

```
curl --request PUT "http://176.120.25.19:9200/pingstation" -H 'Content-Type: application/json' --data "@api-json/elasticsearch-index.json"
```

Step 6. Creating Instances

Once the system is deployed, configurations and modules are uploaded, the streaming layer with necessary infrastructure is created, we can create instances in the next step.

An individual instance should be created for each module.

See the instructions below to create instances for the example task.

To create an instance of the sj-regex-input module send the following request:

```
curl --request POST "http://$address/v1/modules/input-streaming/pingstation-input/1.0/instance" -H 'Content-Type: application/json' --data "@api-json/instances/pingstation-input.json"
```

To create an instance of the ps-process module send the following request:

```
curl --request POST "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance" -H 'Content-Type: application/json' --data "@api-json/instances/pingstation-process.json"
```

To create an instance of the ps-output module send the following request:

```
curl --request POST "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance" -H 'Content-Type: application/json' --data "@api-json/instances/pingstation-output.json"
```

The created instances should be available now in UI under the “Instances” tab. There they will appear with the “ready” status.

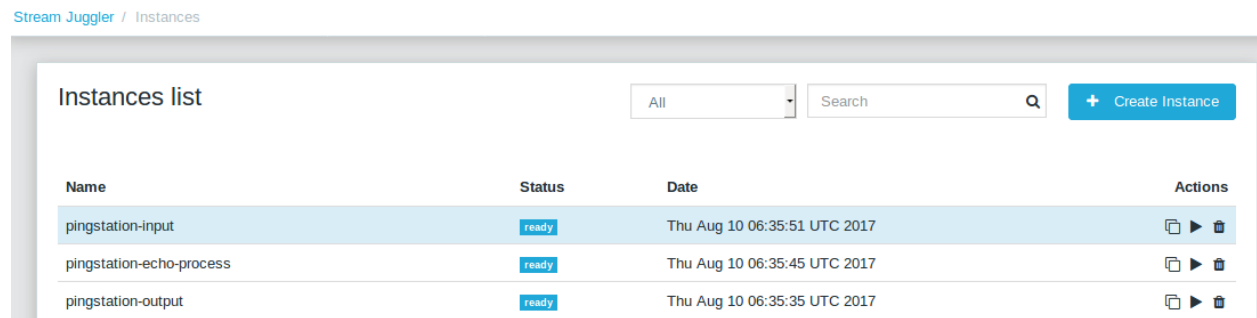


Рис. 3.16: Figure 1.16: Instances list

Ready! The modules can be launched.

Launching Instances

After the streaming layer (with its infrastructure) and instances are ready you can start a module.

The module starts working after its instance is launched. An input module begins to receive data, transforms the data for T-streams to transfer them to the processing module. A processing module begins to process them and put to T-streams to transfer them to the output module. An output module begins to save the result into a data storage.

In the example case, there are three modules and each of them has its own instances. Thus, these instances should be launched one by one.

To launch the input module instance send:

```
curl --request GET "http://$address/v1/modules/input-streaming/pingstation-input/1.0/instance/pingstation-input/start"
```

To launch the processing module instances send:

```
curl --request GET "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance/pingstation-process/start"
```

To launch the output module instances send:

```
curl --request GET "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance/pingstation-
↪output/start"
```

If you take a look at the UI, you will see the launched instances with the “started” status.

Stream Juggler / Instances

| Instances list | | | | All | Search | + Create Instance |
|--------------------------|---------|------------------------------|---------|-----|--------|-------------------|
| Name | Status | Date | Actions | | | |
| pingstation-input | started | Thu Aug 10 10:27:50 UTC 2017 | | | | |
| pingstation-echo-process | started | Thu Aug 10 10:21:09 UTC 2017 | | | | |
| pingstation-output | started | Thu Aug 10 10:21:09 UTC 2017 | | | | |

Pnc. 3.17: Figure 1.17: Started instances

To get a list of ports that are listened by the input module instance send the request:

```
curl --request GET "http://$address/v1/modules/input-streaming/pingstation-input/1.0/instance/pingstation-
↪input"
```

and look at the field named ‘tasks’, e.g. it may look as follows:

```
"tasks": {
  "pingstation-input-task0": {
    "host": "176.120.25.19",
    "port": 31000
  },
  "pingstation-input-task1": {
    "host": "176.120.25.19",
    "port": 31004
  }
}
```

You need to have ‘fping’ installed. If not, please, install it:

```
sudo apt-get install fping
```

And now you can start the processing pipeline. Please, replace values of nc operands with the host and the port of the instance task:

```
fping -l -g 91.221.60.0/23 2>&1 | awk '{printf "%s ", $0; system("echo $(date +%s%N | head -c -7)")} ' | nc 176.
↪120.25.19 31000
```

See the Results

To see the processing results saved in Elasticsearch, please, go to Kibana. There the aggregated data can be rendered on a plot.

The result can be viewed while the module is working. A necessary auto-refresh interval can be set for the diagram to update the graph.

Firstly, click the Settings tab and fill in the data entry field ‘*’ instead of ‘logstash-*’.

Then there will appear another data entry field called ‘Time-field name’. You should choose ‘ts’ from the combobox and press the “Create” button.

After that, click the Discover tab.

Choose a time interval of ‘Last 15 minutes’ in the top right corner of the page, as well as an auto-refresh interval of 45 seconds, as an example. Make a plot.

Select the parameters to show in the graph at the left-hand panel.

The example below is compiled in Kibana v.5.5.1.

It illustrates the average time of echo-responses by IPs per a selected period of time (e.g. 1 min). As you can see, different nodes have different average response times. Some nodes respond faster than others.

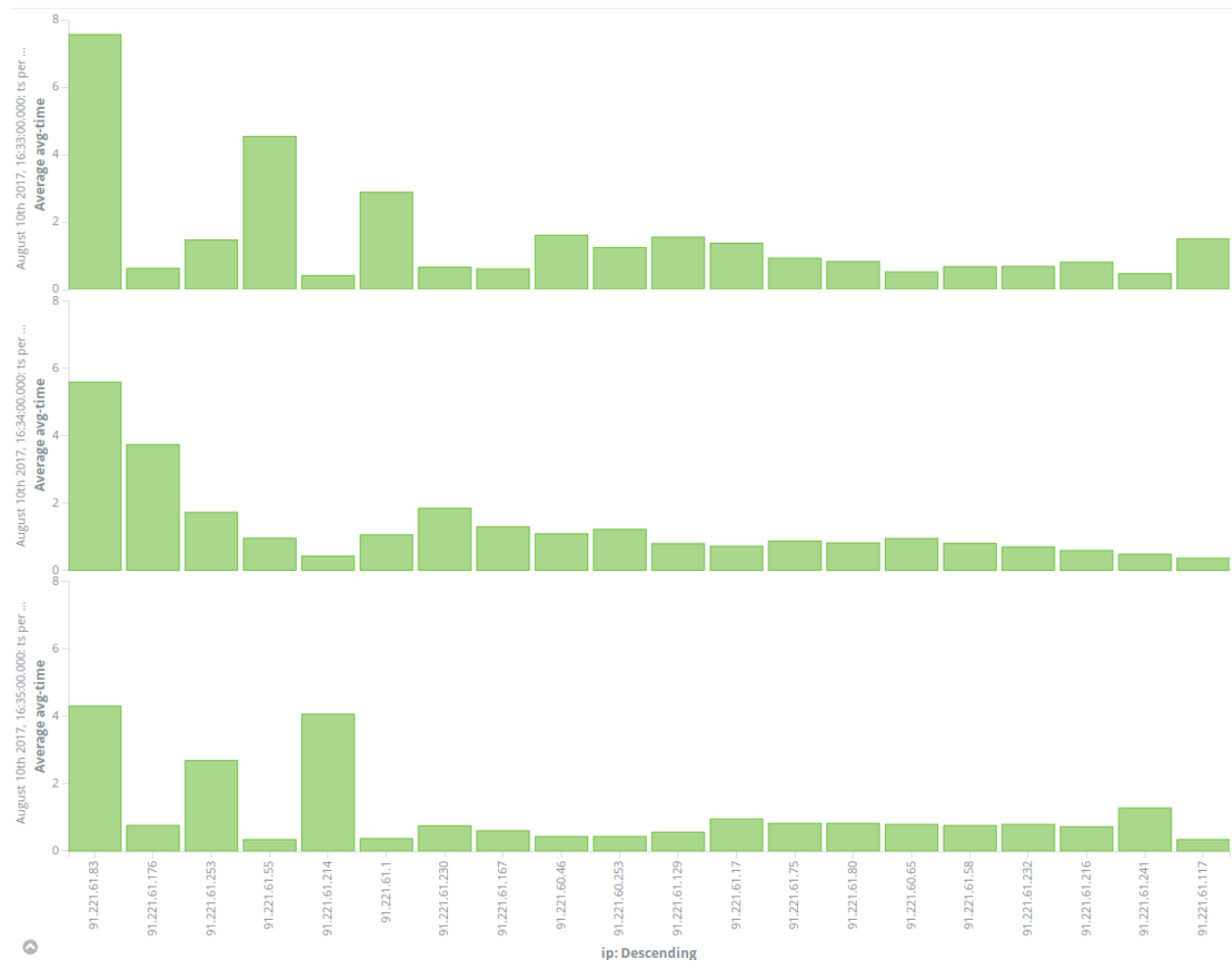


Рис. 3.18: Figure 1.18: View results in Kibana

Many other parameter combinations can be implemented to view the results.

Customization

The platform allows to customize the processing flow. For example, you can change the data aggregation interval adding more instances. You can create one instance with the 3-minute aggregation interval and

another instance with 1-hour aggregation interval.

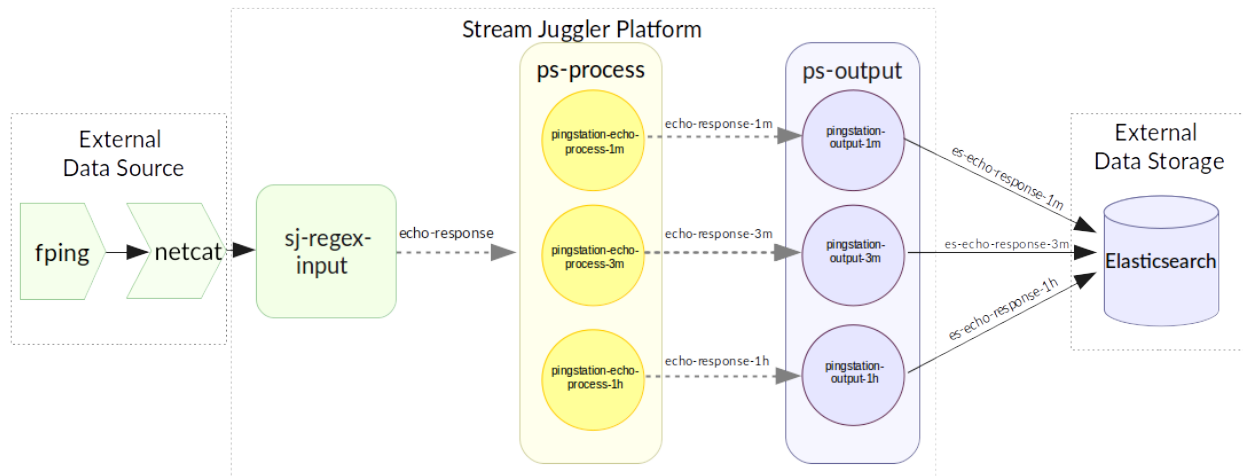


Рис. 3.19: Figure 1.19: Fping example solution customization

We will start from creating input and output streams for the instances.

1. You can create streams on the base of the existing providers and services. Remember to create the JSON files with different names. Create output streams for the ps-process module with the names e.g. 'echo-response-3m' as an output stream aggregating data per 3 minutes, 'echo-response-1h' as an output stream aggregating data per 1 hour:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/echo-response-3m.json"

curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/echo-response-1h.json"
```

Create output streams for the ps-output module with new names, e.g. 'es-echo-response-3m' exporting resulting data aggregated per 3 minutes, 'es-echo-response-1h' exporting resulting data aggregated per 1 hour:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/es-echo-response-3m.json"

curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/es-echo-response-1h.json"
```

2. Create two more instances for the ps-process module with different checkpoint intervals to process data every 3 minutes and every hour. Remember to create them with different names. In our example we create 'pingstation-echo-process-3m' and 'pingstation-echo-process-1h' instances with 'output' values 'echo-response-3m' and 'echo-response-1h' correspondingly:

```
curl --request POST "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance" -H 'Content-Type: application/json' --data "@api-json/instances/pingstation-echo-process-3m.json"

curl --request POST "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance" -H 'Content-Type: application/json' --data "@api-json/instances/pingstation-echo-process-1h.json"
```

3. Create two more instances for the ps-output module with different checkpoint intervals to receive data aggregated for every 3 minutes and for every hour.

Create the JSON files with different names. Change the ‘input’ values to ‘echo-response-3m’ and ‘echo-response-1h’ respectively to receive data from these streams. Change the ‘output’ values to ‘es-echo-response-3m’ and ‘es-echo-response-1h’ correspondingly to put the result data to these streams:

```
curl --request POST "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance" -H
↪ 'Content-Type: application/json' --data "@api-json/instances/pingstation-output-3m.json"

curl --request POST "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance" -H
↪ 'Content-Type: application/json' --data "@api-json/instances/pingstation-output-1h.json"
```

4. Launch the instances as described in the [Launching Instances](#) section. Remember to change the instance names for the new ones:

```
curl --request GET "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance/
↪ pingstation-echo-process-3m/start"
curl --request GET "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance/
↪ pingstation-echo-process-1h/start"

curl --request GET "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance/
↪ pingstation-output-3m/start"
curl --request GET "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance/
↪ pingstation-output-1h/start"
```

Instance Shutdown

Once the task is resolved and necessary data are aggregated, the instances can be stopped.

A stopped instance can be restarted again if it is necessary.

If there is no need for it anymore, a stopped instance can be deleted. On the basis of the uploaded modules and the whole created infrastructure (providers, services, streams) other instances can be created for other purposes.

To stop instances in the example task the following requests should be sent.

To stop the sj-regex-input module instance send:

```
curl --request GET "http://$address/v1/modules/input-streaming/pingstation-input/1.0/instance/pingstation-
↪ input/stop"
```

To stop the ps-process module instance send:

```
curl --request GET "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance/pingstation-
↪ process/stop "
```

To stop the ps-output module instances send:

```
curl --request GET "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance/pingstation-
↪ output/stop"
```

In the UI, you will see the stopped instances with the “stopped” status.

Deleting Instance

A stopped instance can be deleted if there is no need for it anymore. An instance of a specific module can be deleted via REST API by sending a DELETE request (as described below). Or instance deleting action is available in the UI under the “Instances” tab.

Stream Juggler / Instances










| Instances list | | | |
|--------------------------|---------|------------------------------|---|
| Name | Status | Date | Actions |
| pingstation-input | stopped | Thu Aug 10 10:21:07 UTC 2017 |    |
| pingstation-echo-process | stopped | Thu Aug 10 10:21:09 UTC 2017 |    |
| pingstation-output | stopped | Thu Aug 10 10:21:09 UTC 2017 |    |

Рис. 3.20: Figure 1.20: Stopped instances

Make sure the instances to be deleted are stopped and are not with one of the following statuses: «starting», «started», «stopping», «deleting».

The instances of the modules can be deleted one by one.

To delete the sj-regex-input module instance send:

```
curl --request DELETE "http://$address/v1/modules/input-streaming/pingstation-input/1.0/instance/pingstation-
↪input/"
```

To delete the ps-process module instance send:

```
curl --request DELETE "http://$address/v1/modules/regular-streaming/pingstation-process/1.0/instance/
↪pingstation-process/"
```

To delete the ps-output module instance send:

```
curl --request DELETE "http://$address/v1/modules/output-streaming/pingstation-output/1.0/instance/
↪pingstation-output/"
```

In the UI you can make sure the deleted instances are not present.

3.2.5 Sflow Example Task

This is another example of the platform functionality. It represents the processing workflow developed for the demonstration task that is responsible for collecting sFlow information. The aggregated information can be valuable for monitoring the current traffic and predicting of possible problems. The solution represents a scalable system for aggregation of big data in continuous streams. That is extremely important for large computer systems and platforms.

The suggested processing pipeline includes an input module, a batch processing module and an output module. Within the platform, the data are transported with T-streams.

A sFlow reporter is an external data source in our example task. It sends data to the system in CSV format.

The CSV data are transformed by the input module and sent for processing to the batch processing module. The data that can not be parsed by the input module are treated as incorrect and sent straight to the output module without processing.

The processed data are exported via the output module with the streams of SQL-database type and saved in the PostgreSQL database.

A complete pipeline can be rendered as in the diagram below:

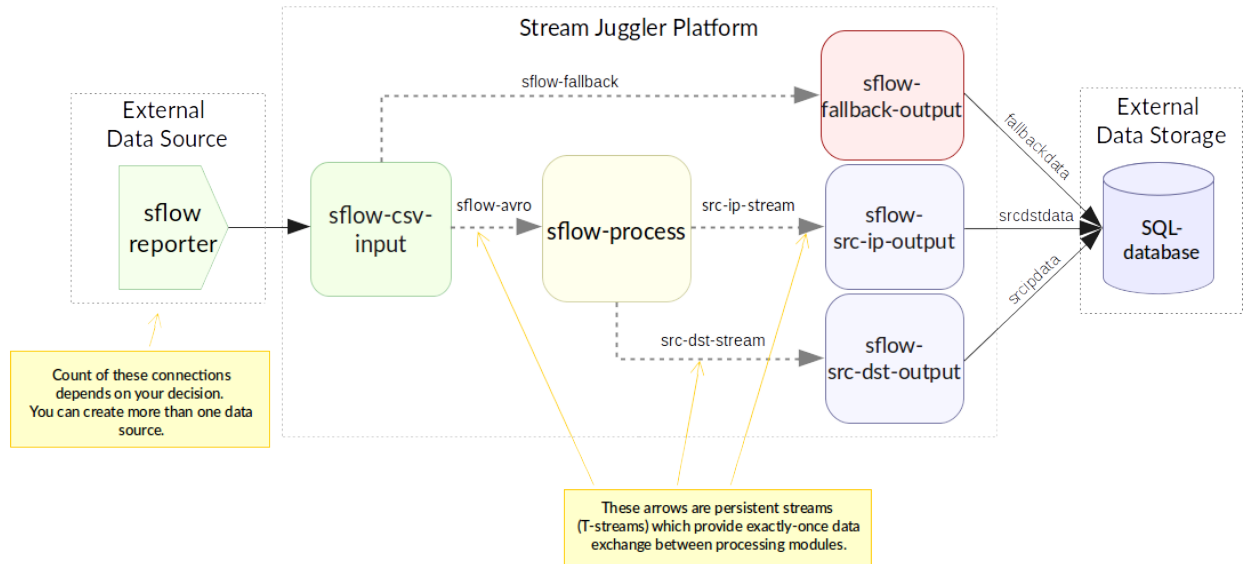


Рис. 3.21: Figure 2.1: sFlow example task pipeline

Green, yellow, purple and red blocks within the SJ-Platform rectangular area are managed and evaluated by SJ-Platform.

These are:

- ‘sflow-csv-input’ module - an input module that transforms CSV data into T-streams;
- ‘sflow-process’ module - a processing module for micro-batch data processing;
- ‘sflow-src-ip-output’ and ‘sflow-src-dst-output’ modules - two output modules that export processed data via T-streams to PostgreSQL;
- ‘sflow-fallback-output’ module - an output module to store incorrect data to a separate table in PostgreSQL.

The blocks beyond the SJ-Platform area represent external systems. Data come to the CSV input module from the sFlow reporter. It sends sFlow records in CSV format to the input module. Then the input module serialises CSV-lines with Apache Avro and puts the data into the ‘sflow-avro’ stream of T-streams type. After that, the batch processing module uses parsed data to:

- computes traffic for the source IP and puts it into ‘src-ip-stream’;
- computes traffic between the source and the destination and puts it into ‘src-dst-stream’.

Finally, the ‘sflow-src-ip-output’ module just displaces data from ‘src-ip-stream’ to the ‘srcipdata’ table in PostgreSQL. The ‘sflow-src-dst-output’ module displaces data from ‘src-dst-stream’ to the ‘srcdstdata’ table.

If the input module cannot parse an input line, then it puts data into the ‘sflow-fallback’ stream. After that the ‘fallback-output’ module moves that incorrect line from ‘sflow-fallback’ to the ‘fallbackdata’ table in PostgreSQL.

Step 1. Deployment

For this demo project the following core systems and services are required:

1. Apache Mesos - a cluster for all computations;

2. Mesosphere Marathon - a framework for executing tasks on Mesos;
3. Apache Zookeeper - to coordinate task executions;
4. Java - a computer software that provides a system for developing application software and deploying it in a cross-platform computing environment;
5. Docker - a software container platform that allows a flexible system configuration;
6. MongoDB - as a database;
7. T-streams - as a message broker ensuring exactly-once data processing;
8. RESTful API - to access and monitor the platform;
9. PostgreSQL - as a destination data storage.

For a start, perform the steps for platform deployment from the [Step1-Deployment](#) section.

1. Deploy Mesos, Apache Zookeeper, Marathon.
2. Create json files for the services and run them:
 - [mongo.json](#)
 - [sj-rest.json](#)
 - [config.properties](#)

For the sFlow demonstrational project the config.properties.json has the following content (remember to replace <zk_ip> with a valid Apache Zookeeper IP):

```
key=sflow
active.tokens.number=100
token.ttl=120

host=0.0.0.0
port=8080
thread.pool=4

path=/tmp
data.directory=transaction_data
metadata.directory=transaction_metadata
commit.log.directory=commit_log
commit.log.rock.directory=commit_log_rock

berkeley.read.thread.pool = 2

counter.path.file.id.gen=/server_counter/file_id_gen

auth.key=dummy
endpoints=127.0.0.1:31071
name=server
group=group

write.thread.pool=4
read.thread.pool=2
ttl.add-ms=50
create.if.missing=true
max.background.compactions=1
allow.os.buffer=true
compression=LZ4_COMPRESSION
use.fsync=true
```

```
zk.endpoints=172.17.0.3:2181
zk.prefix=/sflow
zk.session.timeout-ms=10000
zk.retry.delay-ms=500
zk.connection.timeout-ms=10000

max.metadata.package.size=100000000
max.data.package.size=100000000
transaction.cache.size=300

commit.log.write.sync.value = 1
commit.log.write.sync.policy = every-nth
incomplete.commit.log.read.policy = skip-log
commit.log.close.delay-ms = 200
commit.log.file.ttl-sec = 86400
stream.zookeeper.directory=/tts/tstreams

ordered.execution.pool.size=2
transaction-database.transaction-keep-time-min=70000
subscribers.update.period-ms=500
```

- [tts.json](#)

Via the Marathon interface, make sure the services are deployed and run properly.

Make sure you have access to the Web UI. You will see the platform but there are no entities yet. We will add them further.

Next, we will upload configurations for the platform and engine JARs for modules in the next step.

Step 2. Configurations and Engine Jars Uploading

We upload an engine JAR file for each type of module and for Mesos framework.

Download the engine jars:

```
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-mesos-framework.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-input-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-batch-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-output-streaming-engine.jar
```

And upload them to the system. Please, replace `<slave_advertise_ip>` with Mesos-slave IP:

```
address=<slave_advertise_ip>:31080

curl --form jar=@sj-mesos-framework.jar http://$address/v1/custom/jars
curl --form jar=@sj-input-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-batch-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-output-streaming-engine.jar http://$address/v1/custom/jars
```

Check out in the UI the engines are uploaded:

Setup settings for the engines. Please, replace `<slave_advertise_ip>` with Mesos-slave IP and `<marathon_address>` with the address of Marathon:

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "session-timeout", "value": "7000", "domain": "configuration.apache-zookeeper"}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "current-framework", "value": "com.bwsf.fw-1.0", "domain": "configuration.system"}'
```

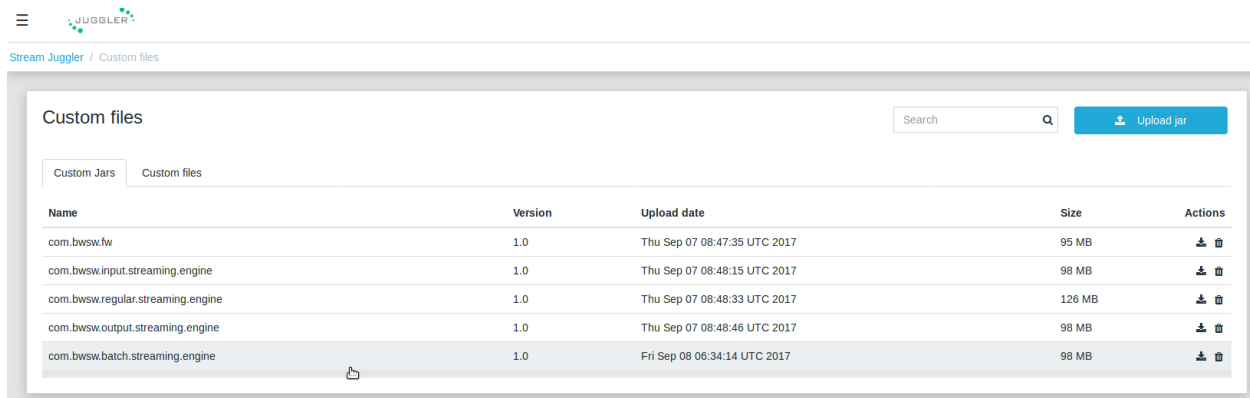


Рис. 3.22: Figure 2.2: Engines list

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"crud-rest-host\"", "value": "\"<slave_advertise_ip>\"", "domain": "\"configuration.system\""}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"crud-rest-port\"", "value": "\"8080\"", "domain": "\"configuration.system\""}'

curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"marathon-connect\"", "value": "\"<marathon_address>\"", "domain": "\"configuration.system\""}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"marathon-connect-timeout\"", "value": "\"60000\"", "domain": "\"configuration.system\""}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"kafka-subscriber-timeout\"", "value": "\"100\"", "domain": "\"configuration.system\""}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"low-watermark\"", "value": "\"100\"", "domain": "\"configuration.system\""}'

curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"batch-streaming-validator-class\"", "value": "\"com.bwsw.sj.crud.rest.instance.validator.\
↪BatchInstanceValidator\"", "domain": "\"configuration.system\""}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"input-streaming-validator-class\"", "value": "\"com.bwsw.sj.crud.rest.instance.validator.\
↪InputInstanceValidator\"", "domain": "\"configuration.system\""}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name\
↪": "\"output-streaming-validator-class\"", "value": "\"com.bwsw.sj.crud.rest.instance.validator.\
↪OutputInstanceValidator\"", "domain": "\"configuration.system\""}'
```

You can see in the UI the configurations are uploaded:

Step 3. Module Uploading

Now let's upload modules for data processing.

First, copy the example project repository from GitHub:

```
git clone https://github.com/bwsw/sj-sflow-demo.git
cd sj-sflow-demo
sbt assembly
```

Then, upload the ready-to-use CSV-input module from Sonatype repository:

Stream Juggler / Configuration

Configuration

+ Add settings

All Search

| Name | Value | Config domain | Date | Actions |
|--------------------------------------|---|--------------------------------|------------------------------|---------|
| com.bwsw.fw-1.0 | sj-mesos-framework.jar | configuration.system | Fri Sep 08 09:55:16 UTC 2017 | |
| com.bwsw.input.streaming.engine-1.0 | sj-input-streaming-engine.jar | configuration.system | Fri Sep 08 09:55:21 UTC 2017 | |
| com.bwsw.output.streaming.engine-1.0 | sj-output-streaming-engine.jar | configuration.system | Fri Sep 08 09:55:40 UTC 2017 | |
| session-timeout | 7000 | configuration.apache-zookeeper | Fri Sep 08 09:56:42 UTC 2017 | |
| current-framework | com.bwsw.fw-1.0 | configuration.system | Fri Sep 08 09:57:09 UTC 2017 | |
| crud-rest-host | sj-rest.marathon.mm | configuration.system | Fri Sep 08 09:57:35 UTC 2017 | |
| crud-rest-port | 8080 | configuration.system | Fri Sep 08 09:57:35 UTC 2017 | |
| marathon-connect | http://marathon.mm:8080 | configuration.system | Fri Sep 08 09:57:49 UTC 2017 | |
| marathon-connect-timeout | 60000 | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| kafka-subscriber-timeout | 100 | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| low-watermark | 100 | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| input-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.InputInstanceValidator | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| output-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.OutputInstanceValidator | configuration.system | Fri Sep 08 09:57:51 UTC 2017 | |
| com.bwsw.batch.streaming.engine-1.0 | sj-batch-streaming-engine.jar | configuration.system | Fri Sep 08 10:00:36 UTC 2017 | |
| batch-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.RegularInstanceValidator | configuration.system | Fri Sep 08 10:02:51 UTC 2017 | |

Рис. 3.23: Figure 2.3: Configurations list

```
curl "https://oss.sonatype.org/content/repositories/snapshots/com/bwsw/sj-csv-input_2.12/1.0-SNAPSHOT/sj-csv-input_2.12-1.0-SNAPSHOT.jar" -o sj-csv-input.jar
curl --form jar=@sj-csv-input.jar http://$address/v1/modules
```

Next, build and upload the batch processing and the output modules of the sFlow demo project.

From the directory of the demo project set up the batch processing module:

```
curl --form jar=@sflow-process/target/scala-2.12/sflow-process-1.0.jar http://$address/v1/modules
```

Next, set up the output modules:

```
curl --form jar=@sflow-output/src-ip/target/scala-2.12/sflow-src-ip-output-1.0.jar http://$address/v1/modules
curl --form jar=@sflow-output/src-dst/target/scala-2.12/sflow-src-dst-output-1.0.jar http://$address/v1/modules
curl --form jar=@sflow-fallback-output/target/scala-2.12/sflow-fallback-output-1.0.jar http://$address/v1/modules
```

Now you can see the uploaded modules in the UI:

Now upload the GeoIP database which is required for the processing module:

```
curl "http://download.maxmind.com/download/geoip/database/asnum/GeoIPASNum.dat.gz" -O
gunzip GeoIPASNum.dat.gz
curl --form file=@GeoIPASNum.dat http://$address/v1/custom/files
```

Then, upload and configure JDBC driver (determine <driver_name> - it can be any name containing letters, digits or hyphens):

```
curl "https://jdbc.postgresql.org/download/postgresql-42.0.0.jar" -O
curl --form file=@postgresql-42.0.0.jar http://$address/v1/custom/files
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name": "\<driver_name>", "value": "postgresql-42.0.0.jar", "domain": "configuration.sql-database"}'
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data '{"name": "\<driver_name>.class", "value": "org.postgresql.Driver", "domain": "configuration.sql-database"}'

```

| Name | Type | Version | Size | Upload date | Actions |
|-----------------------|------------------|---------|-------|------------------------------|---------|
| sflow-process | batch-streaming | 1.0 | 13 MB | Fri Sep 08 10:16:03 UTC 2017 | |
| sflow-src-ip-output | output-streaming | 1.0 | 5 MB | Fri Sep 08 10:16:16 UTC 2017 | |
| sflow-src-dst-output | output-streaming | 1.0 | 5 MB | Fri Sep 08 10:16:16 UTC 2017 | |
| com.bwsw.input.csv | input-streaming | 1.0 | 21 KB | Fri Sep 08 10:16:55 UTC 2017 | |
| sflow-fallback-output | output-streaming | 1.0 | 5 MB | Fri Sep 08 10:17:39 UTC 2017 | |

Рис. 3.24: Figure 2.4: Modules list

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\"name\": \"driver.<driver_name>.prefix\", \"value\": \"jdbc:postgresql\", \"domain\": \"configuration.sql-database\"}"
```

We will use the value of `<driver_name>` in `jdbc-sflow-provider.json` when creating providers in the next step.

Now you can see the settings are added to the configuration list:

Step 4. Creating Streaming Layer

Let's create streams to transfer data from and to the modules.

Creating Infrastructure

The streaming needs the infrastructure - providers and services. We need the following providers for the demonstration task: Apache Zookeeper and SQL database. And the following services: T-streams, Apache Zookeeper and SQL-database.

Providers creation

To create providers you should create json files with the content specified below.

Note: Please, replace the placeholders in the json files: `<login>`, `<password>`, `<host>` and `<port>`. Remove "login" and "password" fields if you do not need authentication to an appropriate server.

`jdbc-sflow-provider.json` (replace `<driver_name>` with the value specified for JDBC driver in the previous step):

Stream Juggler / Configuration

Configuration

[+ Add settings](#)

All

| Name | Value | Config domain | Date | Actions |
|--------------------------------------|---|--------------------------------|------------------------------|---------|
| com.bwsw.fw-1.0 | sj-mesos-framework.jar | configuration.system | Fri Sep 08 09:55:16 UTC 2017 | |
| com.bwsw.input.streaming.engine-1.0 | sj-input-streaming-engine.jar | configuration.system | Fri Sep 08 09:55:21 UTC 2017 | |
| com.bwsw.output.streaming.engine-1.0 | sj-output-streaming-engine.jar | configuration.system | Fri Sep 08 09:55:40 UTC 2017 | |
| session-timeout | 7000 | configuration.apache-zookeeper | Fri Sep 08 09:56:42 UTC 2017 | |
| current-framework | com.bwsw.fw-1.0 | configuration.system | Fri Sep 08 09:57:09 UTC 2017 | |
| crud-rest-host | sj-rest.marathon.mm | configuration.system | Fri Sep 08 09:57:35 UTC 2017 | |
| crud-rest-port | 8080 | configuration.system | Fri Sep 08 09:57:35 UTC 2017 | |
| marathon-connect | http://marathon.mm:8080 | configuration.system | Fri Sep 08 09:57:49 UTC 2017 | |
| marathon-connect-timeout | 60000 | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| kafka-subscriber-timeout | 100 | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| low-watermark | 100 | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| input-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.InputInstanceValidator | configuration.system | Fri Sep 08 09:57:50 UTC 2017 | |
| output-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.OutputInstanceValidator | configuration.system | Fri Sep 08 09:57:51 UTC 2017 | |
| com.bwsw.batch.streaming.engine-1.0 | sj-batch-streaming-engine.jar | configuration.system | Fri Sep 08 10:00:36 UTC 2017 | |
| batch-streaming-validator-class | com.bwsw.sj.crud.rest.instance.validator.RegularInstanceValidator | configuration.system | Fri Sep 08 10:02:51 UTC 2017 | |
| driver.postgresql | postgresql-42.0.0.jar | configuration.sql-database | Fri Sep 08 10:23:34 UTC 2017 | |
| driver.postgresql.class | org.postgresql.Driver | configuration.sql-database | Fri Sep 08 10:24:12 UTC 2017 | |
| driver.postgresql.prefix | jdbc:postgresql | configuration.sql-database | Fri Sep 08 10:24:35 UTC 2017 | |

Рис. 3.25: Figure 2.5: JDBC driver settings are added to the list

```
{
  "name": "jdbc-sflow-provider",
  "description": "JDBC provider for demo",
  "type": "provider.sql-database",
  "login": "<login>",
  "password": "<password>",
  "hosts": [
    "<host>:<port>"
  ],
  "driver": "<driver_name>"
}
```

zookeeper-sflow-provider.json (remember to replace <host>:<port> with a valid Apache Zookeeper IP):

```
{
  "name": "zookeeper-sflow-provider",
  "description": "Zookeeper provider for demo",
  "type": "provider.apache-zookeeper",
  "hosts": [
    "<host>:<port>"
  ]
}
```

Then create providers:

```
curl --request POST "http://$address/v1/providers" -H 'Content-Type: application/json' --data "@api-json/
↪providers/jdbc-sflow-provider.json"
curl --request POST "http://$address/v1/providers" -H 'Content-Type: application/json' --data "@api-json/
↪providers/zookeeper-sflow-provider.json"
```

Check out they have appeared in the UI:

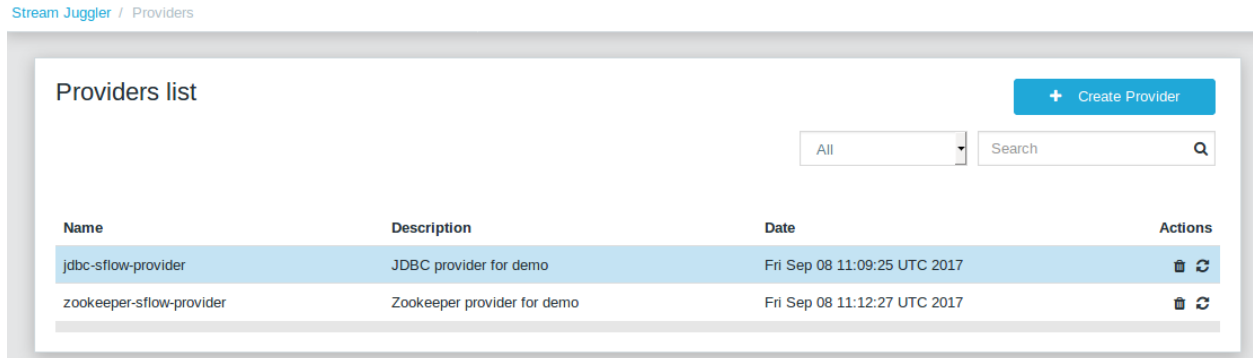


Рис. 3.26: Figure 2.6: Providers list

Once providers are created, we can create services.

Services creation

Services of three types are required: T-streams, Apache Zookeeper and SQL-database.

To create services:

```
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/
↪services/jdbc-sflow-service.json"
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/
↪services/tstream-sflow-service.json"
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/
↪services/zookeeper-sflow-service.json"
```

Check out the services have appeared in the UI:

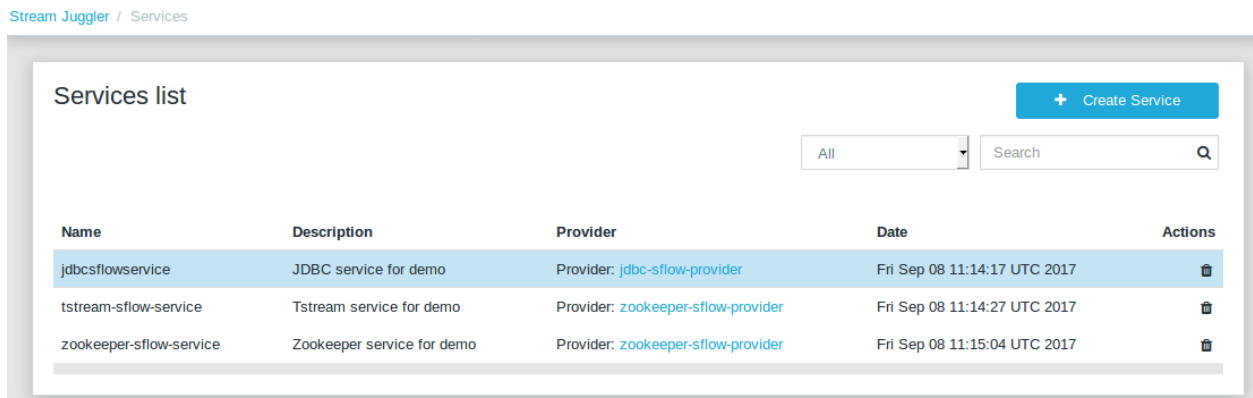


Рис. 3.27: Figure 2.7: Services list

Streams creation

Now you can create streams that will be used by the instances of input, processing, output and fallback-output modules.

First, we create output streams of the input module:

- 'sflow-avro' — the stream for correctly parsed sFlow records;
- 'sflow-fallback' — the stream for incorrect inputs.

Run the following commands:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/sflow-avro.json"
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/sflow-fallback.json"
```

Secondly, we create output streams of the processing module to keep information about:

1. traffic for the source IP,
2. traffic between the source IP and destination:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/src-ip-stream.json"
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/src-dst-stream.json"
```

Thirdly, we create output streams of the output modules to save information to the database. Use the following commands:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/src-ip-data.json"
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/src-dst-data.json"
```

Fourthly, we create an output stream of the fallback-output module to save incorrect inputs to the database. Use the following commands:

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/streams/fallback-data.json"
```

Check out that they have appeared in the UI:

Step 5. Output SQL Tables Creation

At this step you are expected to have PostgreSQL running with sflow database in it.

SQL tables for the output data should be created in the sflow database. To create tables:

```
CREATE TABLE srcipdata (
  id SERIAL PRIMARY KEY,
  src_ip VARCHAR(32),
  traffic INTEGER,
  txn BIGINT
);

CREATE TABLE srcdstdata (
  id SERIAL PRIMARY KEY,
  src_as INTEGER,
  dst_as INTEGER,
  traffic INTEGER,
  txn BIGINT
);
```


| Name | Description | Service | Date | Actions |
|----------------|--|--|------------------------------|---------|
| srcipdata | JDBC stream for demo | Service: jdbcsflobservice | Mon Sep 11 09:13:02 UTC 2017 | |
| srcdstdata | JDBC stream for demo | Service: jdbcsflobservice | Mon Sep 11 09:13:12 UTC 2017 | |
| fallbackdata | JDBC stream for demo | Service: jdbcsflobservice | Tue Sep 12 02:01:46 UTC 2017 | |
| sflow-avro | Tstream for normal input in sflow demo | Service: tstream-sflow-service | Tue Sep 12 02:25:09 UTC 2017 | |
| sflow-fallback | Tstream for fallback in sflow demo | Service: tstream-sflow-service | Tue Sep 12 04:05:42 UTC 2017 | |
| src-ip-stream | Tstream for demo | Service: tstream-sflow-service | Tue Sep 12 04:05:55 UTC 2017 | |
| src-dst-stream | Tstream for demo | Service: tstream-sflow-service | Tue Sep 12 04:06:07 UTC 2017 | |

Рис. 3.28: Figure 2.8: Streams list

```
);
CREATE TABLE fallbackdata (
  id SERIAL PRIMARY KEY,
  line VARCHAR(255),
  txn BIGINT
);
```

Step 6. Creating Instances

An instance should be created for each module. It is a set of settings determining the collaborative work of a module and an engine.

In the demonstrational case, we will create one instance for the input module, one instance for the processing module. As there are three output modules. Thus, we will create three instances for the output.

To create an instance of the input module:

```
curl --request POST "http://$address/v1/modules/input-streaming/com.bws.input.csv/1.0/instance" -H
  'Content-Type: application/json' --data "@api-json/instances/sflow-csv-input.json"
```

To create an instance of the processing module:

```
curl --request POST "http://$address/v1/modules/batch-streaming/sflow-process/1.0/instance" -H 'Content-
  Type: application/json' --data "@api-json/instances/sflow-process.json"
```

To create instances of the output modules:

```
curl --request POST "http://$address/v1/modules/output-streaming/sflow-src-ip-output/1.0/instance" -H
  'Content-Type: application/json' --data "@api-json/instances/sflow-src-ip-output.json"
curl --request POST "http://$address/v1/modules/output-streaming/sflow-src-dst-output/1.0/instance" -H
  'Content-Type: application/json' --data "@api-json/instances/sflow-src-dst-output.json"
```

To create an instance of the fallback-output module:

```
curl --request POST "http://$address/v1/modules/output-streaming/sflow-fallback-output/1.0/instance" -H
  ↪ 'Content-Type: application/json' --data "@api-json/instances/sflow-fallback-output.json"
```

View them in the UI:

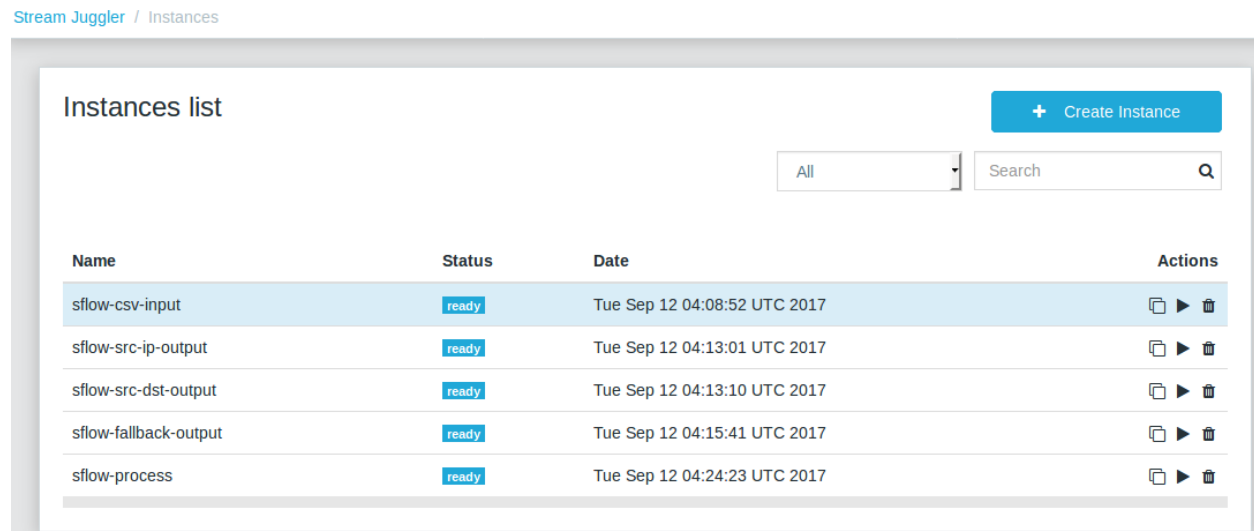


Рис. 3.29: Figure 2.9: Instances list

Launching Instances

Now you can launch each instance.

To launch the input module instance:

```
curl --request GET "http://$address/v1/modules/input-streaming/com.bsw.input.csv/1.0/instance/sflow-csv-
  ↪input/start"
```

To launch the processing module instance:

```
curl --request GET "http://$address/v1/modules/batch-streaming/sflow-process/1.0/instance/sflow-process/start"
```

To launch the output module instances:

```
curl --request GET "http://$address/v1/modules/output-streaming/sflow-src-ip-output/1.0/instance/sflow-src-ip-
  ↪output/start"
curl --request GET "http://$address/v1/modules/output-streaming/sflow-src-dst-output/1.0/instance/sflow-src-
  ↪dst-output/start"
```

To launch the fallback-output module instance:

```
curl --request GET "http://$address/v1/modules/output-streaming/sflow-fallback-output/1.0/instance/sflow-
  ↪fallback-output/start"
```

To get the list of ports the input module listens, send the following command:

```
curl --request GET "http://$address/v1/modules/input-streaming/com.bsw.input.csv/1.0/instance/sflow-csv-
↪input"
```

and look at the field named tasks. It may look as follows:

```
"tasks": {
  "sflow-csv-input-task0": {
    "host": "176.120.25.19",
    "port": 31000
  }
}
```

Or, in the UI, click at the input module instance in the “Instances” section and unfold the Tasks section of the Instance Details panel:

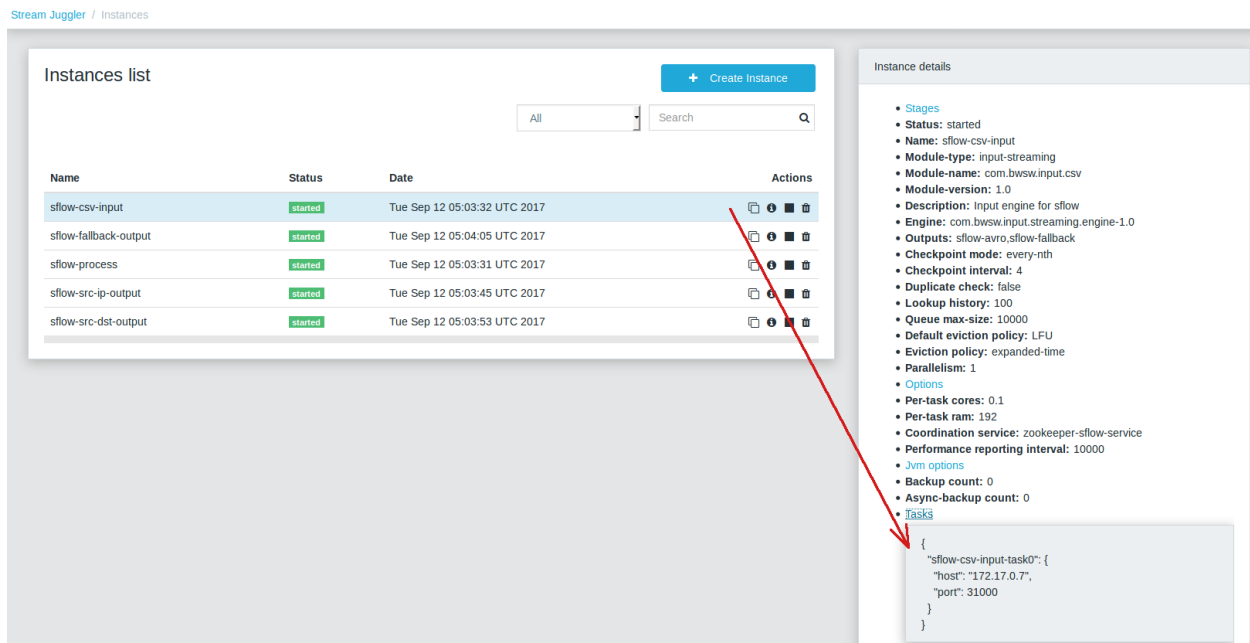


FIG. 3.30: Figure 2.10: Started instances

And now you can start the processing pipeline (please, replace <host> and <port> by the values returned for the instance task of the input module):

```
python send_sflow.py -p <port> -h <host> sflow_example.csv
```

See the Results

To see the results execute the following queries in the output database:

```
SELECT * FROM srcipdata;
SELECT * FROM srcdstdata;
SELECT * FROM fallbackdata;
```

You should see a table similar to the one below:

```
sflow=# SELECT * FROM srcipdata;
```

| id | src_ip | traffic | txn |
|---------------------------------------|--------------|---------|-------------------|
| 84cf5fad-aa64-4081-a9bc-3ce51110953d | 66.77.88.99 | 1055750 | 14918948733750000 |
| 65dcbeeb2-7a6c-4a2b-a622-b030e13ef546 | 11.22.33.44 | 588000 | 14918948733750000 |
| 6b26b6cf-f4a8-4334-839f-86e1404bca16 | 11.73.81.44 | 660500 | 14918948733750000 |
| 5492c762-b536-49b5-8088-1649cc09f0fb | 11.22.33.201 | 310500 | 14918948733750000 |

(4 rows)

```
sflow=# SELECT * FROM srcdstdata;
```

| id | src_as | dst_as | traffic | txn |
|--------------------------------------|--------|--------|---------|-------------------|
| 4b18d026-de4c-43fa-a765-8b308c28f75b | 0 | 0 | 100000 | 14918948736400000 |
| a43f0243-3ba7-4305-9664-3d0938bad394 | 0 | 0 | 1148500 | 14918948736400000 |
| cc326d39-8de5-487b-bfff-87b3202ef645 | 209 | 209 | 293250 | 14918948736400000 |
| 236942d4-a334-4f4f-845f-c288bca6cebd | 0 | 0 | 310500 | 14918948736400000 |
| afca82ab-5f30-4e09-886c-a689554621c7 | 209 | 209 | 172500 | 14918948736400000 |
| d8a34274-db5b-480b-8b6c-bd36b991d131 | 209 | 209 | 590000 | 14918948736400000 |

(6 rows)

```
sflow=# SELECT * FROM fallbackdata;
```

| id | line | txn |
|--------------------------------------|---|-------------------|
| 31652ea0-7437-4c48-990c-22ceab50d6af | 1490234369,sfr6,10.11.12.13,4444,5555,INCORRECT | 14911974375950000 |

(1 row)

Instance Shutdown

To stop the input module instance execute:

```
curl --request GET "http://$address/v1/modules/input-streaming/com.bsw.input.csv/1.0/instance/sflow-csv-
↪input/stop"
```

To stop the processing module instance execute:

```
curl --request GET "http://$address/v1/modules/batch-streaming/sflow-process/1.0/instance/sflow-process/stop"
```

To stop the output module instances execute:

```
curl --request GET "http://$address/v1/modules/output-streaming/sflow-src-ip-output/1.0/instance/sflow-src-ip-
↪output/stop"
curl --request GET "http://$address/v1/modules/output-streaming/sflow-src-dst-output/1.0/instance/sflow-src-
↪dst-output/stop"
```

To stop the fallback-output module instance execute:

```
curl --request GET "http://$address/v1/modules/output-streaming/sflow-fallback-output/1.0/instance/sflow-
↪fallback-output/stop"
```

Deleting Instances

A stopped instance can be deleted if there is no need for it anymore. An instance of a specific module can be deleted via REST API by sending a DELETE request (as described below). An instance deleting action is also available in the UI in the “Instances” section.

Make sure the instances you are going delete are stopped and are not with one of the following statuses: «starting», «started», «stopping», «deleting».

The instances of the modules can be deleted one by one.

To delete the input module instance:

```
curl --request DELETE "http://$address/v1/modules/input-streaming/com.bwsw.input.csv/1.0/instance/sflow-csv-↵input/"
```

To delete the process module instance:

```
curl --request DELETE "http://$address/v1/modules/batch-streaming/sflow-process/1.0/instance/sflow-process/"
```

To delete the output module instances:

```
curl --request DELETE "http://$address/v1/modules/output-streaming/sflow-src-ip-output/1.0/instance/sflow-src-↵ip-output/"
curl --request DELETE "http://$address/v1/modules/output-streaming/sflow-src-dst-output/1.0/instance/sflow-↵src-dst-output/"
```

To delete the fallback-output module instance:

```
curl --request DELETE "http://$address/v1/modules/output-streaming/sflow-fallback-output/1.0/instance/sflow-↵fallback-output/"
```

You can check the UI to make sure the instances are deleted.

3.2.6 More Information

Find more information about SJ-Platform and its entities at:

[Modules: Types, Structure, Pipeline](#) - more about module structure.

[Custom Module Development Guide](#) - how to create a module.

[SJ-Platform Architecture](#) - the structure of the platform.

[UI Guide](#) - the instructions on platform monitoring via the Web UI.

[REST API Guide](#) - the RESTful API to configure and monitor the platform.

3.3 SJ-Platform Architecture

A good data processing system needs to be fault-tolerant and scalable; it needs to support micro-batch and event-by-event data processing and must be extensible. The Stream Juggler Platform architecture fulfills all these aspects.

The Stream Juggler Platform is an integrated processing system. It means the system includes all the parts required to achieve goals: components for computation, administration. These parts can be rearranged in different pipelines. That allows building sophisticated processing graphs to customize the system.

SJ-Platform's architecture is designed so that exactly-once processing is performed not only within a single processing block but throughout the entire platform, starting from the moment streams of events are fed to the system and up to the moment the output data are saved in conventional data storage. This approach based on loosely coupled blocks allows a user to combine the decomposed blocks to create different processing pipelines for a wide variety of tasks. It provides better modularity, performance management and simplicity in development.

In this section, we will take a closer look at the system components, their functions within the data flow pipeline.

But first of all, let's get the general idea of the platform structure and concepts.

3.3.1 Architecture Overview

The platform features presented above have conditioned the architecture developed for SJ-Platform. The diagram below represents the overview of the platform components:

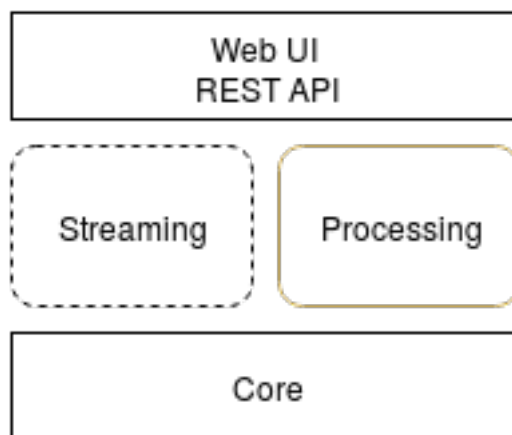


Рис. 3.31: Figure 1.1: Platform structure

1. Processing component for launching data computation,
2. Streaming component for data transportation,
3. Core component for task execution,
4. API/UI component for administration.

The Stream Juggler Platform performs data processing that is fulfilled in modules. The mode of processing in a module is determined by a set of configurations uploaded to the system via the UI.

The events enter the processing module in streams from a list of supported interfaces - TCP, Apache Kafka and T-streams. A result data are placed into an external data storage.

SJ-Platform provides a user with the comprehensive API and UI that allow him/her to develop, customize and manage the event processing pipeline.

The core component is presented with services that simplify the deployment and operation and support best industrial practices.

3.3.2 Platform Components

Now let's have a look at each component in detail.

Core Component

The Core is composed of prerequisites for the platform. These are the services and settings that should be deployed prior to exploring the Stream Juggler Platform features. The services at this layer are responsible for input data ingestion, task execution management, data storage.

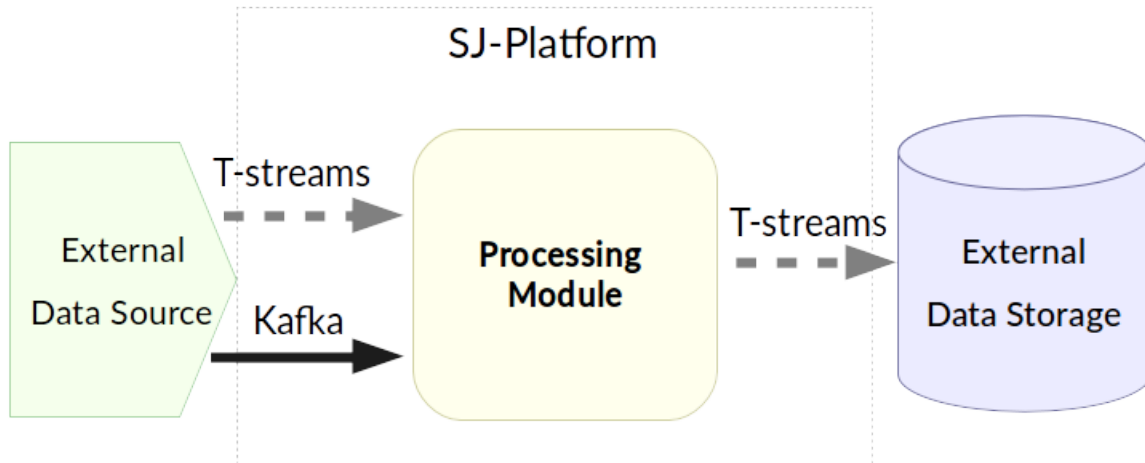


Рис. 3.32: Figure 1.2: Streams in the system

- Resource management is fulfilled via [Apache Mesos](#) that allows to run the system at scale and to support different types of workloads.
- To start applicable services in the Mesos we use [Docker](#)
- [Marathon](#) allows running long-life tasks on Mesos.
- To perform leader election in case the currently leading Marathon instance fails [Apache Zookeeper](#) is used. Zookeeper is also responsible for instance task synchronization for a Batch module.
- Data sources for the platform are [Netty](#) and [T-streams](#) libraries and [Apache Kafka](#).
- The system saves the outcoming data to Elasticsearch, JDBC-compatible or RESTful external storages.
- We use [MongoDB](#) as a document database that provides high performance and availability. All created platform entities (Providers, Services, Streams, Instances, etc.), as well as configurations, are stored here.

SJ-Platform's backend is written in Scala. The UI is based on Angular 4.0. REST API is written in Akka HTTP.

Processing Component

The Processing component is provided by the Stream Juggler Platform. At this layer, the data processing itself is performed via modules. In fact, the platform represents a pipeline of modules.

The major one is the processing module that performs data processing. Two types of processing modules exist in SJ-Platform:

- Regular – the most generic module which receives events, transforms data element by element and sends them to the next processing step.
- Batch – a module where the processing algorithm must observe a range of input messages rather than the current one (as it is in the regular module). For each stream input messages are collected into batches. Then batches are collected in a window. Windows of several streams are transferred to the module for processing. Thus, the module allows processing of data from several streams at the same time. In SJ-Platform the data are processed using sliding window.

The processing module receives data for processing from Apache Kafka and T-streams. You also can use TCP as a source, but you will need an input module in this case. The input module handles external inputs,

does data deduplication, transforms raw data into objects for T-streams.

To receive the result of processing an output module is required. The output module puts the processed data from event processing pipeline to external data destinations (Elasticsearch, SQL-database, RESTful).

So the pipeline may look like at the following scheme:

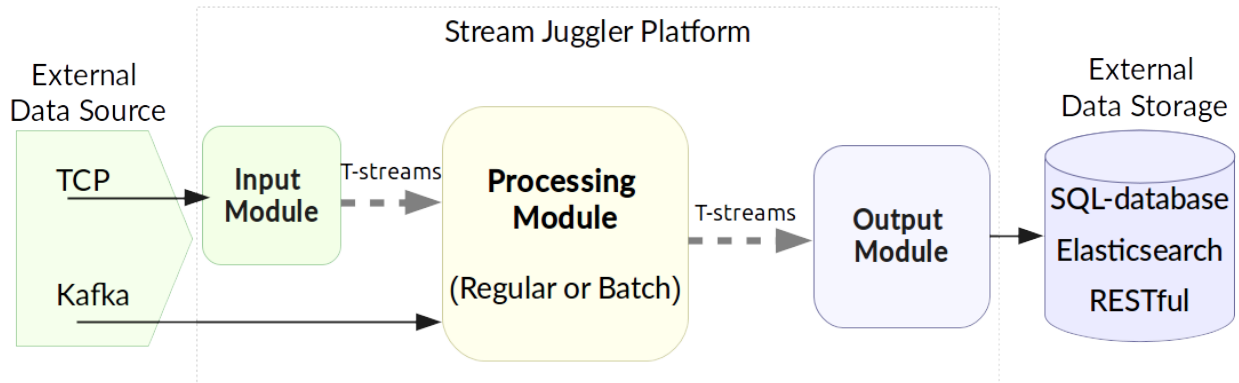


Рис. 3.33: Figure 1.3: Pipeline structure

In the Processing platform component, the ingested data are transformed into streams, processed and sent to an external storage. Data transformation and computation are the two major tasks of this component.

Tip: More information on modules you can find at the [Modules: Types, Structure, Pipeline](#) page.

Streaming Component

The Streaming component is essential in SJ-Platform. The data are fed to the system, transferred between modules and exported to an external storage via streams. Stream usage makes possible such platform features as exactly-once processing and parallelism.

The data can be received from different sources. Currently, the platform supports obtaining data from Apache Kafka and via TCP.

Using TCP as an input source a custom protocol can be used for receiving events, deduplicating them and putting into the processing pipeline.

SJ-Platform supports Apache Kafka as a standard message broker providing a common interface for the integration of many applications.

Within the platform, the data are transported to and from modules via transactional streams or T-streams. It is a message broker which is native to SJ-Platform and designed primarily for exactly-once processing (so it includes a transactional producer, a consumer and a subscriber).

Tip: More information on T-streams as well as streaming infrastructure in SJ-Platform can be found at the [Streams in SJ-Platform](#) page.

API/UI Component

The Web UI allows managing and administrating of the platform. It is based on Angular 4.0.

Also, the platform provides REST API that allows interacting with the platform, monitoring and managing module statuses (its starting or stopping), retrieving configuration information.

Tip: More information about the UI in the platform can be found in the [UI Guide](#) and the [REST API Guide](#).

3.3.3 Platform Features

Each SJ-Platform component contributes to its outstanding features.

SJ-Platform performs stream processing. That means the system can handle events as soon as they are available inside the system without specific delay. Micro-batch data processing can be also performed in the system.

Streams can be very intensive and all events cannot be handled by a single server of arbitrary performance. The system allows scaling the computations horizontally to handle increasing demands.

The events are guaranteed to be processed exactly-once. The key idea of exactly-once processing lies in the checkpoint. That means all producers and consumers of a stream are bunched into a group and do a checkpoint automatically fixing the current state. Moreover, a user can initialize a checkpoint after any stage of the processing cycle.

The idea of parallelism is implemented via multi-partitioning in streams. A partition is a part of a data stream allocated for convenience in stream processing. Upon creation, every stream gets a certain amount of partitions. The parallelism is enabled by dividing existing partitions fairly among the tasks of module instance and thus scaling the data processing.

SJ-Platform fulfills the idea of fault-tolerance as its architecture prevents the whole system from stopping operation completely in case of module failure. In such case when a live data stream processing fails in one module, the module is restarted by Marathon. Within the module, if it runs in a parallel mode (several tasks are set in the parameters of module instance) and one of the tasks fails, the whole system does not stop processing. The task is restarted.

The general structure of SJ-Platform can be rendered as at the scheme below where all the mentioned above components are presented in detail:

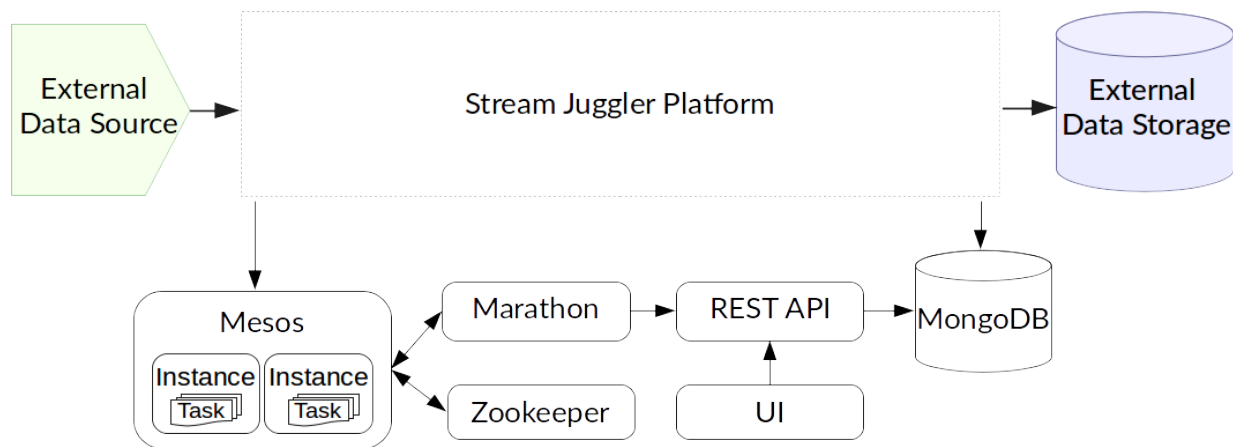


Рис. 3.34: Figure 1.4: SJ-Platform general structure

To understand the interconnections between platform components look at the diagram below.

SJ-Platform uses a range of data sources and data storages. A client operates the platform via UI/REST API. And he/she uploads a custom module to the platform with a set of configurations. The platform runs the module via an “executable” engine on Mesos and Marathon. And the module uses MongoDB as a data store.

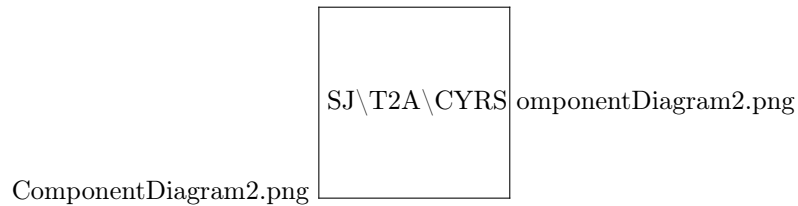


Рис. 3.35: Figure 1.5: Platform component diagram

Every component deployed to the Stream Juggler Platform contributes to the main idea of hitting three V-s of data processing:

- **Volume** The system is scalable and perfectly fits for large data processing.
- **Velocity** The Stream Juggler Platform is the solution for real-time processing that means the system can handle events as soon as they are available inside the system without specific delay. Alongside with the stream processing, the micro-batch processing is supported in the system.
- **Variety** The SJ-Platform components are ready-to-use and can be reorganized in various pipelines. Besides, the system is compatible with different types of data sources, data storages, services and systems. Stream Juggler Platform easily integrates with in-memory grid systems, for example, Hazelcast, Apache Ignite.

The system is available under Apache License v2.

3.4 Modules: Types, Structure, Pipeline

Contents

- **Modules: Types, Structure, Pipeline**
 - Streaming Validator
 - Executor
 - Data Processing Flow in Modules
 - Module Types
 - * Modules of Input Type
 - * Modules of Regular Type
 - * Modules of Batch Type
 - Batch Collector
 - * Modules of Output Type
 - Module Instances
 - * Example of Instance Infrastructure
 - * General Instance Infrastructure

A module is a processor that handles events of data streams. It is a key program component in the system that contains the logic of data processing and settings validation. It works within an engine that receives raw data and sends them to the module executor. A module uses instances as a range of settings determining the collaborative work of an engine and a module.

A module includes:

- an executor that processes data streams,
- a validator to check an instance.

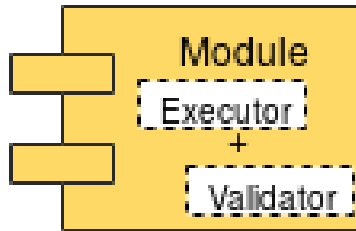


Рис. 3.36: Figure 1.1: Module structure

Below you will find more information on each of these two components.

3.4.1 Streaming Validator

Streaming validator provides two methods:

1. to validate options parameter of the module.
2. to validate an instance of the module. The method is used when you try to create a new instance of a specific module, and if the validation method returns false value the instance will not be created.

Each of these two methods returns a tuple of values that contains:

- The value that indicates if the validation is successful or not;
- The value that is a list of errors in case of the validation failures (it is an empty list by default).

3.4.2 Executor

An executor is a key module component that performs the data processing. It receives data and processes them in correspondence with the requirements of module specification. It utilizes an instance/instances for processing. An instance is a full range of settings for an exact module. It determines the coordinated work of an engine and a module.

3.4.3 Data Processing Flow in Modules

In general, data processing in modules can be described in a simple scheme.

The base of the system is an engine: it provides basic I/O functionality. It is started via a Mesos framework which provides distributed task dispatching and then the information on task execution. The engine performs data processing using an uploaded module.

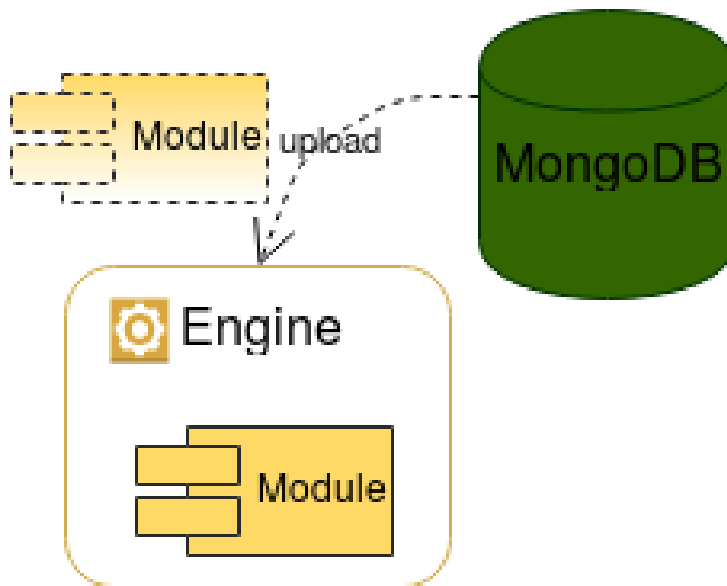


Рис. 3.37: Figure 1.2: An engine in SJ-Platform

After its uploading, the engine receives raw data and sends them to the module executor. The executor starts data processing and returns the resulting data back to the engine where they are deserialized to be passed into the stream or a storage.

3.4.4 Module Types

The system supports 4 types of modules:

1. Input-streaming - handles external inputs, does data deduplication, transforms raw data to objects.
2. Regular-streaming (base type) - the most generic module which receives events, transforms data element by element and sends them to the next processing step.
3. Batch-streaming - a module where the processing algorithm must observe a range of input messages rather than the current one (as it is in the regular-streaming type). For each stream input messages are collected into batches. Then batches are collected in a window. Windows of several streams are transferred to the module for processing. Thus, the module allows processing of data from several streams at the same time. In SJ-Platform the data are processed using sliding window.
4. Output - handles the data outcoming from event processing pipeline to external data destinations (Elasticsearch, SQL database, RESTful).

The modules can be strung in a pipeline as illustrated below:

At this page each module is described in detail. You will find more information on the methods provided by module executors as well as entities' description.

Modules of Input Type

Modules of the input-streaming type handle external input streams, does data deduplication, transforms raw data to objects.

In the SJ-Platform the TCP Input module is currently implemented.

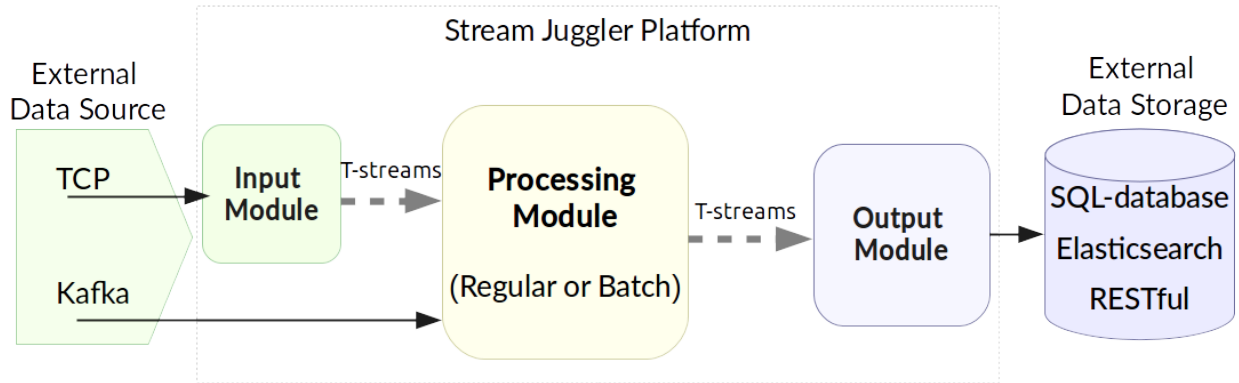


Рис. 3.38: Figure 1.3: Pipeline structure

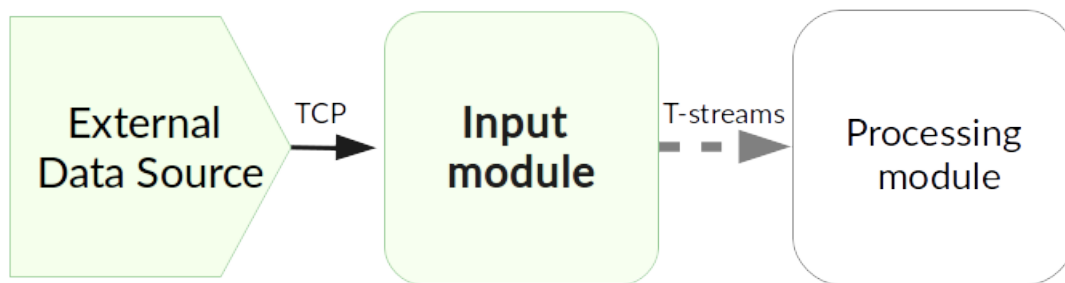


Рис. 3.39: Figure 1.4: Input module direct interaction with pipeline components

It performs the transformation of the streams incoming via TCP into T-streams. T-streams are persistent streams designed for exactly-once processing (so they include a transactional producer, a consumer and a subscriber). Find more information about T-streams [here](#).

In the diagram below you can see the illustration of dataflow for the input module.

All input data elements are going as a flow of bytes to particular interface provided by Task Engine. That flow is going straight to Streaming Executor and is converted to an object called an Input Envelope.

An envelope is a specialized fundamental data structure containing data and metadata. The metadata is required for exactly-once processing.

The Input Envelope then goes to Task Engine which serializes it to a stream of bytes and then sends to T-Streams.

An input module executor provides the following methods with default implementation (which can be overridden)f.

1. **tokenize**: It is invoked every time when a new portion of data is received. It processes a flow of bytes to determine the beginning and the end of the Interval (significant set of bytes in incoming flow of bytes). By default it returns None value (meaning that it is impossible to determine an Interval). If Interval detected, method should return it (indexes of the first and the last elements of the interval in the flow of bytes). The resulting interval can either contain message or not.
2. **parse**: This method is invoked once the “tokenize” method returns an Interval. It processes both a buffer with incoming data (a flow of bytes) and an Interval (an output of “tokenize” method). Its purpose is to define whether the Interval contains a message or meaningless data. Default return value is None. The same value should be returned if Interval contains meaningless data. If Interval contains a message, the “InputEnvelope” value should be returned.

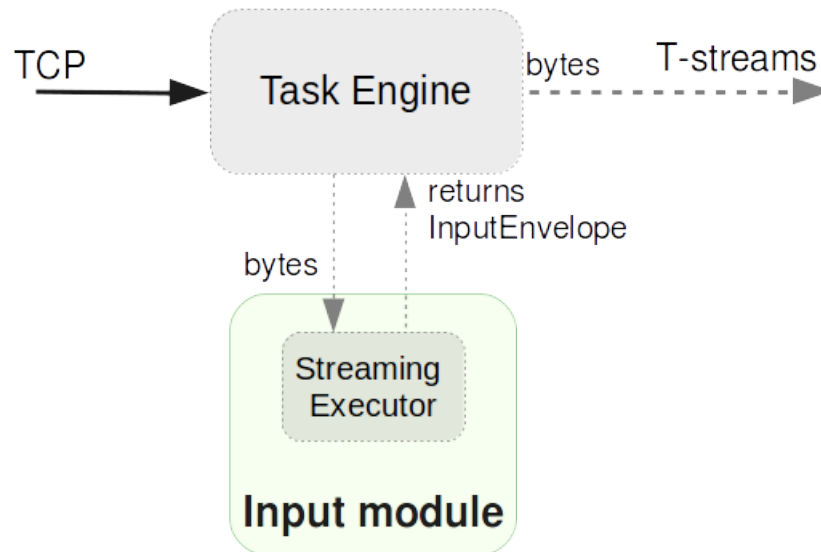


Рис. 3.40: Figure 1.5: Processing in the input module

3. `createProcessedMessageResponse`: It is invoked after each call of the “parse” method. Its purpose is to create response to the source of data - the instance of `InputStreamingResponse`.

The parameters of the method are:

- `InputEnvelope` (it can be defined or not)
- `isEmptyOrDuplicate` - a boolean flag (denoting whether an “`InputEnvelope`” is defined and isn’t a duplicate (true) or an `InputEnvelope` is a duplicate or empty (false))

Default implementation of the method:

```

def createProcessedMessageResponse(envelope: Option[InputEnvelope],
    isEmptyOrDuplicate: Boolean): InputStreamingResponse = {
  var message = ""
  var sendResponsesNow = true
  if (isEmptyOrDuplicate) {
    message = s"Input envelope with key: '${envelope.get.key}' has been sent\n"
    sendResponsesNow = false
  } else if (envelope.isDefined) {
    message = s"Input envelope with key: '${envelope.get.key}' is duplicate\n"
  } else {
    message = s"Input envelope is empty\n"
  }
  InputStreamingResponse(message, sendResponsesNow)
}

```

4. `createCheckpointResponse`: It is invoked on checkpoint’s finish. It’s purpose is to create response for data source to inform that checkpoint has been done. It returns an instance of `InputStreamingResponse`.

Default implementation of the method:

```

def createCheckpointResponse(): InputStreamingResponse = {
  InputStreamingResponse(s"Checkpoint has been done\n", isBuffered = false)
}

```

```
}

```

There is a manager inside the module which allows to:

- retrieve a list of output stream names by a set of tags (by calling `getStreamsByTags()`)
- initiate checkpoint at any time (by calling `initiateCheckpoint()`) which would be performed only at the end of processing step (check diagram at the [Input Streaming Engine](#) page)

Entities description

InputEnvelope:

- key of an envelope
- information about the destination
- “check on duplication” boolean flag (it has higher priority than `duplicateCheck` in `InputInstance`)
- message data

InputStreamingResponse:

- message - string message
- `sendResponsesNow` - a boolean flag denoting whether response should be saved in temporary storage or all responses from this storage should be send to the source right now (including this one)

To see a flow chart on how these methods intercommunicate, please, visit the [Input Streaming Engine](#) page.

Input Modules Provided By SJ-Platform

The Stream Juggler Platform offers two examples of Input Module implementation. These are ready-to-use input modules for two most general input data formats: CSV and Regex. Find a detailed description of these modules at the `Provided_Input_Modules` section.

Modules of Regular Type

The most generic modules in the system are modules of a regular-streaming type. A simplified definition of a regular module is a handler that performs data transformation and put the processed data into T-streams.

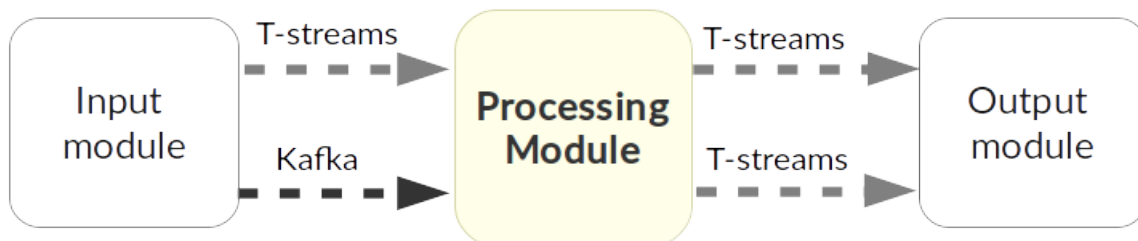


Рис. 3.41: Figure 1.6: Regular module direct interaction with pipeline components

The diagram below represents the dataflow in the regular module.

The `TaskEngine` of a regular module receives data from T-streams. It deserializes the flow of bytes to `TStreamsEnvelope[T]` (where `[T]` is a type of messages in the envelope) which is then passed to the `StreamingExecutor`.

The `StreamingExecutor` processes the received data and sends them to the `TaskEngine` as a result data.

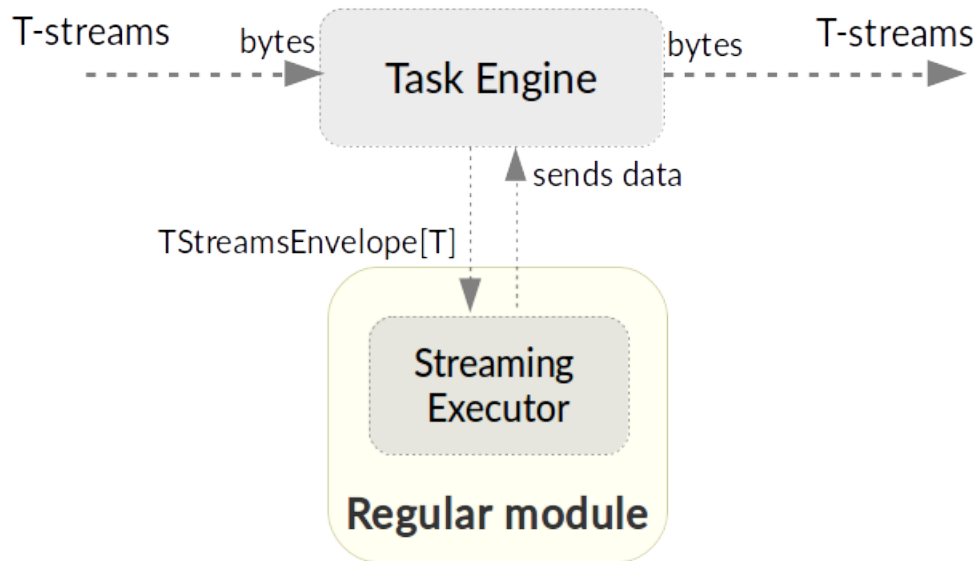


Рис. 3.42: Figure 1.7: Processing in the regular module

The TaskEngine serializes all the received data to the flow of bytes and puts it back to T-Streams to send further.

In the Regular module the executor provides the following methods that does not perform any work by default so you should define their implementation by yourself.

1. `onInit`: It is invoked only once, when a module is launched. This method can be used to initialize some auxiliary variables, or check the state variables on existence and create them if necessary . Thus, you should do preparation of the executor before usage.

Example of the checking a state variable:

```
if (!state.isExist(<variable_name>)) state.set(<variable_name>, <variable_value>)
```

<variable_name> must have the String type

<variable_value> can be any type (a user must be careful when casting a state variable value to a particular data type)

2. `onMessage`: It is invoked for every received message from one of the inputs that are defined within the instance. There are two possible data types of input sources - that's why there are two methods with appropriate signatures:

```
def onMessage(envelope: TStreamEnvelope[T]): Unit
def onMessage(envelope: KafkaEnvelope[T]): Unit
```

Each envelope has a type parameter that defines the type of data in the envelope.

Note: The data type of the envelope can be only “KafkaEnvelope” data type or “TStreamEnvelope” data type. A user may specify one of them or both, depending on which type(s) is(are) used.

3. `onBeforeCheckpoint`: It is invoked before every checkpoint.

4. `onTimer`: It is invoked every time when a set timer expires. Inside the method there is access to a parameter that defines a delay between a real response time and an invocation of this handler.
5. `onIdle`: It is invoked every time when idle timeout expires but a new message hadn't appeared. It is a moment when there is nothing to process.
6. `onBeforeStateSave`: It is invoked prior to every saving of the state. Inside the method there is a flag denoting whether the full state (true) or partial changes of state (false) will be saved.

The module may have a state. A state is a sort of a key-value storage and can be used to keep some global module variables related to processing. These variables are persisted and are recovered after a fail.

In case of a fail (when something is going wrong in one of the methods described above) a whole module will be restarted. And the work will start with the `onInit` method call.

Inside the module there is a manager allowing to get access to:

- an output that is defined within the instance (by calling `getPartitionedOutput()` or `getRoundRobinOutput()`),
- timer (by calling `setTimer()`)
- state (by calling `getState()`) if it is a stateful module
- list of output names (by calling `getStreamsByTags()`). Every output contains its own set of tags which are used to retrieve it.
- initiation of checkpoint (by calling `initiateCheckpoint()`).

To see a flow chart on how these methods intercommunicate see the [Regular Streaming Engine](#) section.

Modules of Batch Type

Modules of a batch-streaming type process events collecting them in batches. A batch is a minimum data set for a handler to collect the events in the stream. The size of a batch is defined by a user. It can be a period of time or a quantity of events or a specific type of the event after receiving which the batch is considered closed. Then, the queue of batches is sent further in the flow for the next stage of processing.

Batch Collector

In the module a Batch Collector is responsible for the logic of collecting batches. It provides the following methods, implementation of which you should specify.

1. `getBatchesToCollect`: It should return a list of stream names that are ready to be collected.
2. `afterEnvelopeReceive`: It is invoked when a new envelope is received.
3. `prepareForNextCollecting`: It is invoked when a batch is collected. If several batches are collected at the same time then the method is invoked for each batch.

Let us consider an example:

This is a batch collector defining that a batch consists of a certain number of envelopes:

```
class NumericalBatchCollector(instance: BatchInstanceDomain,
                             performanceMetrics: BatchStreamingPerformanceMetrics,
                             streamRepository: Repository[StreamDomain])
extends BatchCollector(instance, performanceMetrics, streamRepository) {

  private val logger = LoggerFactory.getLogger(this.getClass)
  private val countOfEnvelopesPerStream = mutable.Map(instance.getInputsWithoutStreamMode.map(x => (x,
  0)))
```

```

private val everyNthCount = 2 (2)

def getBatchesToCollect(): Seq[String] = {
  countOfEnvelopesPerStream.filter(x => x._2 == everyNthCount).keys.toSeq
  ↪ (3)
}

def afterEnvelopeReceive(envelope: Envelope): Unit = {
  increaseCounter(envelope) (4)
}

private def increaseCounter(envelope: Envelope) = {
  countOfEnvelopesPerStream(envelope.stream) += 1
  logger.debug(s"Increase count of envelopes of stream: ${envelope.stream} to: $
  ↪ {countOfEnvelopesPerStream(envelope.stream)}.")
}

def prepareForNextCollecting(streamName: String): Unit = {
  resetCounter(streamName) (5)
}

private def resetCounter(streamName: String) = {
  logger.debug(s"Reset a counter of envelopes to 0.")
  countOfEnvelopesPerStream(streamName) = 0
}

```

Let's take a look at the main points:

- .(1) - create a storage of incoming envelopes for each input stream.
- .(2) - set a size of batch (in envelopes).
- .(3) - check that batches contain the necessary number of envelopes.
- .(4) - when a new envelope is received then increase the number of envelopes for specific batch.
- .(5) - when a batch has been collected then reset the number of envelopes for this batch.

The module allows transformation of the data aggregated from input streams applying the sliding window.

A window size is equal to a number of batches. The window closes once it is full, i.e. the set number of batches is collected. The collected batches are transferred further for processing and the window slides forward for the set interval. This is the sliding window technique.

The diagram below is a simple illustration of how a sliding window operation looks like.

As shown in the figure, every time the window slides over an input stream, the batches of events that fall within the window are combined and operated upon to produce the transformed data of the windowed stream. It is important that any window operation needs to specify the parameters:

- batch size — The quantity of events within a batch, or a period of time during which the events are collected in one batch.
- window size - The duration of the window, i.e. how many batches should be collected before sliding.
- sliding interval - A step size at which the window slides forward.

In the example, the operation is applied over the last 3 events, and slides by 2 events. Thus, the window size is 3 and the sliding interval is 2.

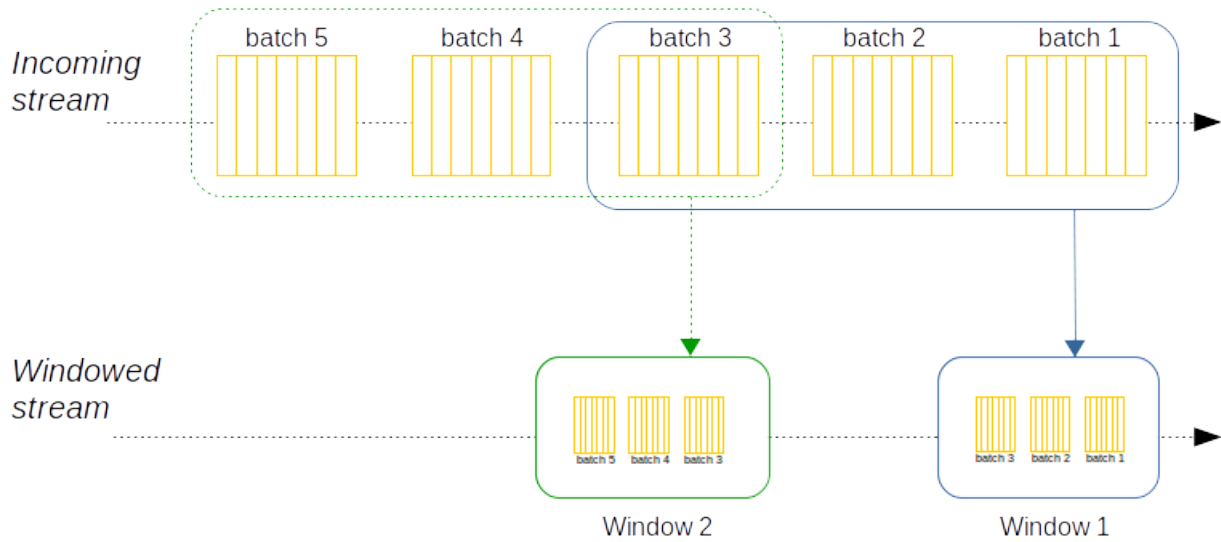


Рис. 3.43: Figure 1.8: Sliding windowing

In general, a window consists of batches, a batch consists of events (messages) that may contain data of different type depending on a data type of input. So, each event should have a type parameter that defines the type of data containing in the event unit.

The executor of the batch module provides the following methods that does not perform any work by default. So you should define their implementation by yourself.

1. `onInit`: It is invoked only once, when a module is launched. This method can be used to initialize some auxiliary variables or check the state variables on existence and if it's necessary to create them. Thus, you should do preparation of the executor before usage.

Example of the checking a state variable:

```
if (!state.isExist(<variable_name>)) state.set(<variable_name>, <variable_value>)
```

<variable_name> should be of the String type

<variable_value> can be of any type (be careful when you will cast a state variable value to a particular data type)

2. `onWindow`: It is invoked when a window for each input stream is collected (a list of input streams are defined within the instance). These collected windows are accessible via a window repository within the method. A window consists of batches, a batch consists of envelopes (messages). There are two possible data types of envelopes - that's why you should cast the envelope inside the method. Each envelope has a type parameter that defines the type of message data.

Example of a message casting to a particular data type:

```
val allWindows = windowRepository.getAll()
allWindows.flatMap(x => x._2.batches).flatMap(x =>
x.envelopes).foreach {
case kafkaEnvelope: KafkaEnvelope[Integer @unchecked] => //here there is access to certain fields
↳such as offset and data of integer type
case tstreamEnvelope: TStreamEnvelope[Integer @unchecked] => //here there is access to certain
↳fields such as txnUUID, consumerName and data (array of integers)
}
```

The data type of the envelope can be “KafkaEnvelope” data type or “TStreamEnvelope” data type. If you specify the inputs of the only one of this data types in an instance, you shouldn’t match the envelope like in the example above and cast right the envelope to a particular data type:

```
val tstreamEnvelope =  
    envelope.asInstanceOf[TStreamEnvelope[Integer]]
```

3. `onBeforeCheckpoint`: It is invoked before every checkpoint
4. `onTimer`: It is invoked every time when a set timer expires. Inside the method there is access to a parameter that defines a delay between a real response time and an invocation of this handler
5. `onIdle`: It is invoked every time when idle timeout expires but a new message hasn’t appeared. It is a moment when there is nothing to process
6. `onBeforeStateSave`: It is invoked before every saving of the state. Inside the method there is a flag denoting whether the full state (true) or partial changes of state (false) will be saved

When running a module in a parallel mode (the instance “parallelism” parameter is greater than 1), you may need to exchange data between tasks at the exact moment. You should use shared memory for it, e.g. Hazelcast or any other. In this case, the following handlers are used for synchronizing the tasks’ work:

1. `onEnter`: The system awaits every task to finish the `onWindow` method and then the `onEnter` method of all tasks is invoked.
2. `onLeaderEnter`: The system awaits every task to finish the `onEnter` method and then the `onLeaderEnter` method of a leader task is invoked.

To see a flow chart about how these methods intercommunicate see the [Batch Streaming Engine](#) section.

The Batch module can either have a state or not. A state is a sort of a key-value storage and can be used to keep some global module variables related to processing. These variables are persisted and are recovered after a fail.

A fail means that something is going wrong in one of the methods described above. In this case a whole module will be restarted. And the work will start with the `onInit` method call.

Saving of the state is performed alongside with the checkpoint. At a checkpoint the data received after processing is checked for completeness. The checkpoint is an event that provides an exactly-once processing.

There is a manager inside the module which grants access to:

- output that was defined within the instance (by calling `getPartitionedOutput()` or `getRoundRobinOutput()`),
- timer (by calling `setTimer()`)
- state (by calling `getState()`) (only if it is a module with state)
- list of output names (by calling `getStreamsByTags()`). Every output contains its own set of tags which are used to retrieve it.
- initiation of checkpoint (by calling `initiateCheckpoint()`)

Modules of Output Type

Modules of an output type are responsible for saving of output data to external data destinations (Elasticsearch, SQL database, RESTful).

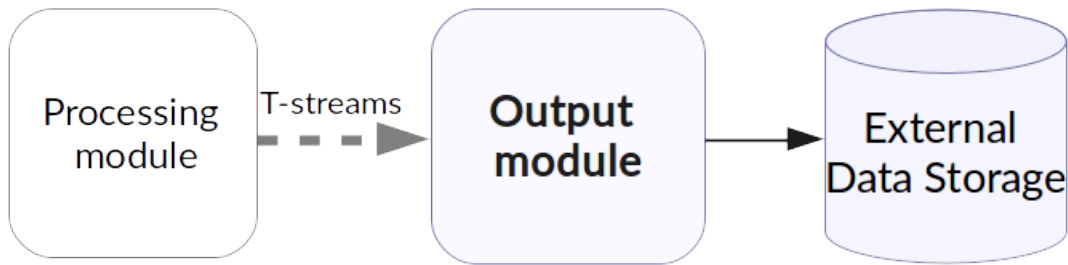


Рис. 3.44: Figure 1.9: Output module direct interaction with pipeline components

They transform the result of data processing received from T-streams and pass them to an external data storage. They allow to transform one data item from incoming streaming into one and more data output items.

The diagram below illustrates the dataflow in an output module.

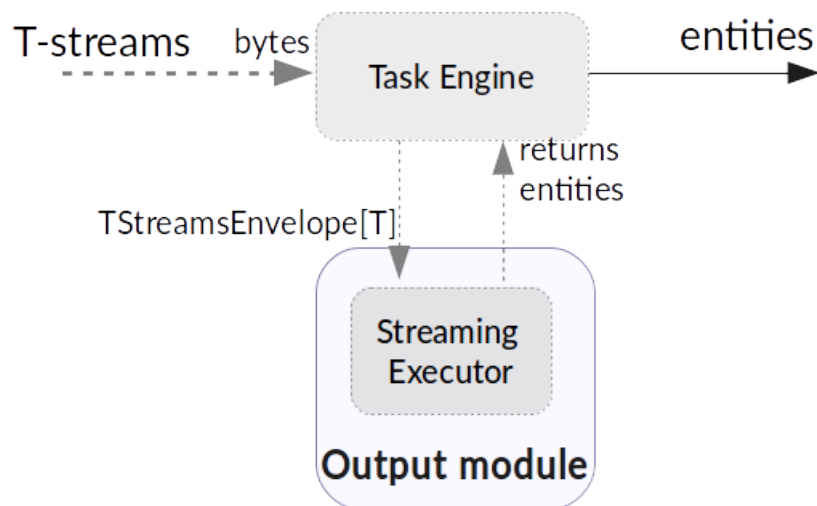


Рис. 3.45: Figure 1.10: Processing in the output module

The TaskEngine deserializes the stream of bytes from T-Streams to TStreamsEnvelope[T] (where [T] is a type of messages in the envelope) and sends it to the StreamingExecutor. The StreamingExecutor returns Entities back to the TaskEngine.

They are then put to an external datastorage.

The output executor provides the following methods that does not perform any work by default so you should define their implementation by yourself.

1. `onMessage`: It is invoked for every received message from one of the inputs that are defined within the instance. Inside the method you have access to the message that has the TStreamEnvelope type.
2. `getOutputEntity`: It is invoked once when module running. This method returns the current working entity, i.e. fields and types. This method must be overridden.

We should assign a type to an output envelope that corresponds to the type of an external storage (Elasticsearch, SQL database, RESTful).

To see a flow chart on how these methods intercommunicate, please, visit the [Output Streaming Engine](#) section.

A detailed manual on how to write a module you may find at the [Hello World Custom Module](#) page.

Modules' performance is determined by the work of an engine. Engines of different types (Input, Regular/Batch, Output) have different structure, components and the workflow corresponding to the type of a module.

Please, find more information about engines at the [Engines: types, workflow](#) page.

3.4.5 Module Instances

Each type of modules described above requires an instance of a corresponding type. An instance is a set of settings determining the collaborative work of an engine and a module. These settings are specified via the UI or REST API and determine the mode of the module operation: data stream type the module is going to work with, checkpoint concept, settings of state and parallelism, etc. In the system instances can be of the following types: input, processing (regular/batch), output.

Module's instances require the following elements to be created for their work:

- a stream
- a service
- a provider

You should create these elements before creating an instance. You need streams for instance inputs and outputs. Streams, in their turn, require specific services to be created. Each service (based on its type) requires a provider of a corresponding type. All these elements form the infrastructure for an instance.

Each instance type works with a specific type of streams, services and providers. Find below the detailed information on the types of providers, services and streams required for each instance type. Besides, we will provide you an example to explain the dependence of entity types on an instance type.

Streams

The Stream Juggler Platform supports Apache Kafka and T-stream types of streams. And while the Apache Kafka streams are a well-known type of streaming introduced by Apache Software Foundation, the T-streams is intentionally designed for the Stream Juggler Platform as a complement for Apache Kafka. The T-streams has more features than Kafka and makes exactly-once processing possible. Find more about T-streams at the [site](#) .

The following stream types can be used for output streams that export resulting data from the system to an external storage: Elasticsearch, SQL-database, a system with RESTful interface. They are determined by the type of the external data storage.

Services and Providers

To create streams of exact type in the system you need to create a service and a provider for this service. The types of a service and a provider are determined by the type of a stream you need for the module.

Example of Instance Infrastructure

In this section we will describe the process of determining of all the needed entities for the instance infrastructure.

For example, there is some issue for which you need to process data from Apache Kafka in a micro-batch mode. So we need to include a Batch module into our pipeline.

For the Batch module we need to create a batch instance. In the system an instance of any type requires Apache Zookeeper service and Apache Zookeeper provider for it (Figure 1.11). The Apache Zookeeper service should be unique for all the instances in the system.

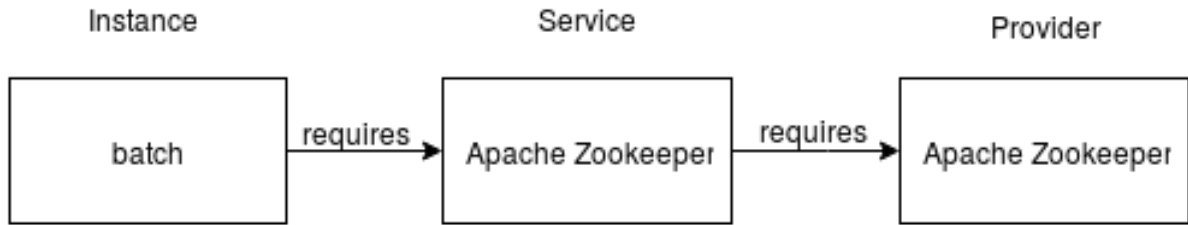


Рис. 3.46: Figure 1.11: Instance dependence on Apache Zookeeper

The batch instance will receive data from Apache Kafka streams. Apache Kafka streams require the Apache Kafka service to exist in our system. The Apache Kafka service requires two specific providers of the following types: Apache Kafka and Apache Zookeeper (the same as in the previous step) (Figure 1.12).

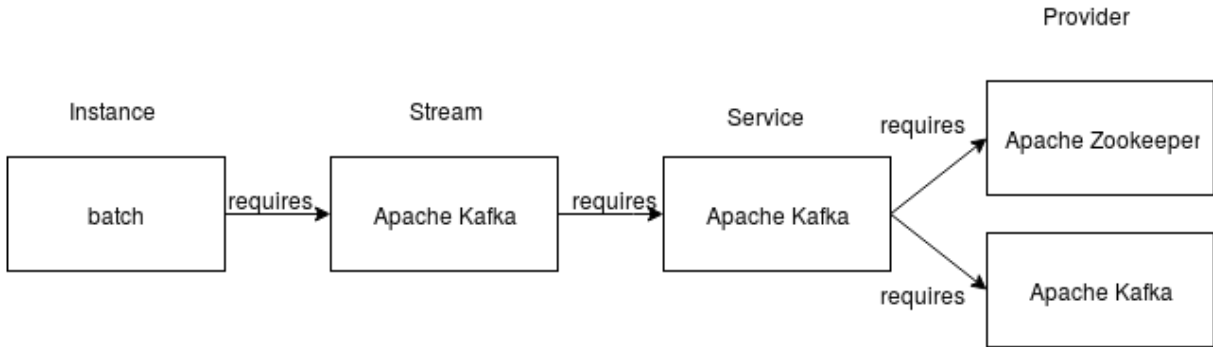


Рис. 3.47: Figure 1.12: Apache Kafka streaming infrastructure for the batch instance

So these are the instance and the streaming components types that we need for our example:

At this point we have determined the type of instance in the pipeline and the types of streaming components. So we can start building the infrastructure.

Firstly, create two providers - Apache Kafka and Apache Zookeeper. Secondly, create Apache Kafka service and Apache Zookeeper service (that will be unique for all instances in the system). Thirdly, create the stream of Apache Kafka. Finally, create the instance of a batch module.

General Instance Infrastructure

The schemas below may help you to understand the dependency of entities in the system for each module instance type.

Input module instance type works with the following entities types:

Processing module instance type (regular or batch) works with the following entities types:

Output module instance type works with the following entities types:

The table below explains what inputs and outputs can be used for a particular instance type:

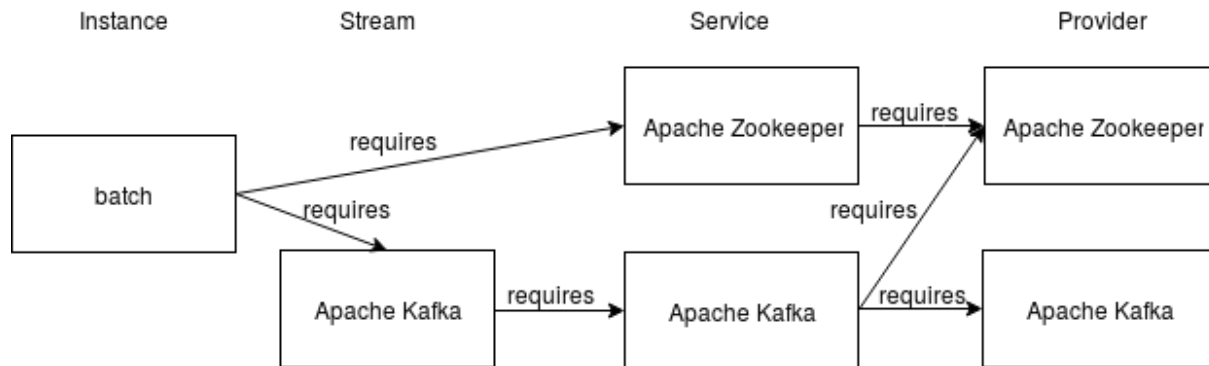


Рис. 3.48: Figure 1.13: Batch instance infrastructure example

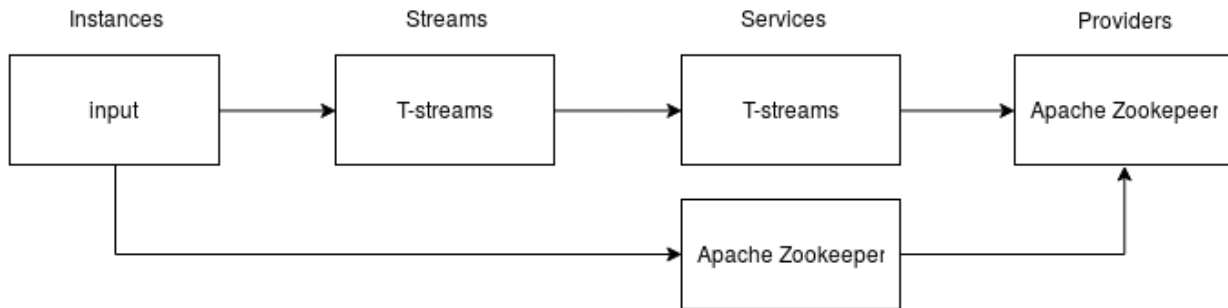


Рис. 3.49: Figure 1.14: Instance infrastructure for the input instance type

→ points to the entity type required for creation of this entity.

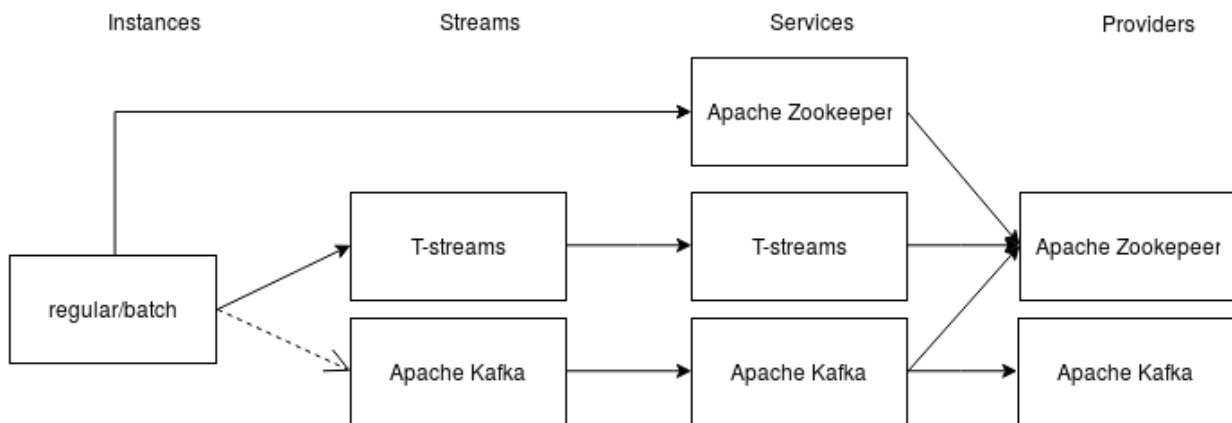


Рис. 3.50: Figure 1.15: Instance infrastructure for the processing instance type

→ points to the entity type required for creation of this entity.

-.-> points to the entity which may be needed when creating a dependent entity.

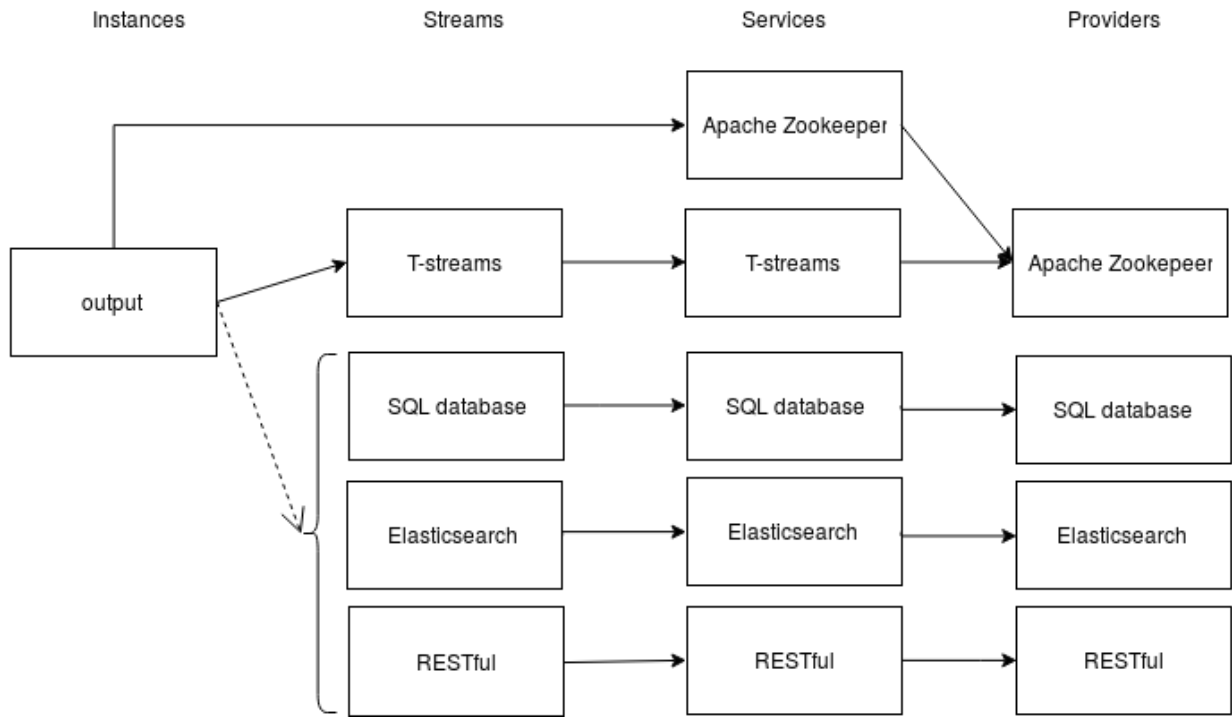


Рис. 3.51: Figure 1.16: Instance infrastructure for the output instance type

→ points to the entity type required for creation of this entity.

- - - - -> { points to the set of entities, one of which shall be created before creating the dependent entity.

| Instance type | Inputs | Outputs |
|---------------|--|---|
| Input | TCP (provided by Input Streaming Engine) | T-streams Providers: Apache Zookeeper Services: T-streams, Apache Zookeeper |
| Regular/Batch | T-streams Providers: Apache Zookeeper Services: T-streams, Apache Zookeeper Apache Kafka Providers: Apache Zookeeper, Apache Kafka Services: Apache Zookeeper, Apache Kafka | T-streams Providers: Apache Zookeeper Services: T-streams, Apache Zookeeper |
| Output | T-streams Providers: Apache Zookeeper Services: T-streams, Apache Zookeeper | Elasticsearch Providers: Elasticsearch Services: Elasticsearch, Apache Zookeeper SQL database Providers: SQL database Services: SQL database, Apache Zookeeper RESTful Providers: RESTful Services: RESTful, Apache Zookeeper |

We hope this information will help you to select the most appropriate types of entities in the system to build a pipeline for smooth data stream processing.

You can find the details on creating instances and their infrastructure in the [UI Guide](#).

3.5 Engines: types, workflow

Contents

- Engines: types, workflow
 - Engine Types

- * [Input Streaming Engine](#)
- * [Regular Streaming Engine](#)
- * [Batch Streaming Engine](#)
- * [Output Streaming Engine](#)

An engine is the base of the system. It provides basic I/O functionality. It is started via a Mesos framework which provides distributed task dispatching and then the statistics on task execution.

The engine performs data processing using an uploaded module. After its uploading, the engine receives raw data and sends them to the module executor. At this moment a module starts working. Module's executor starts data processing and returns the resulting data back to the engine where they are deserialized to be passed into the stream or a storage.

3.5.1 Engine Types

The types of the engines correspond to the types of modules in the platform (see [Modules: Types, Structure, Pipeline](#)):

1. Input Streaming Engine
2. Output Streaming Engine
3. Regular Streaming Engine
4. Batch Streaming Engine

Each engine has its unique workflow.

Input Streaming Engine

Input streaming engine is required to run an input module (currently TCP Input Stream Module); it handles external inputs, does data deduplication, transforms raw data to objects.

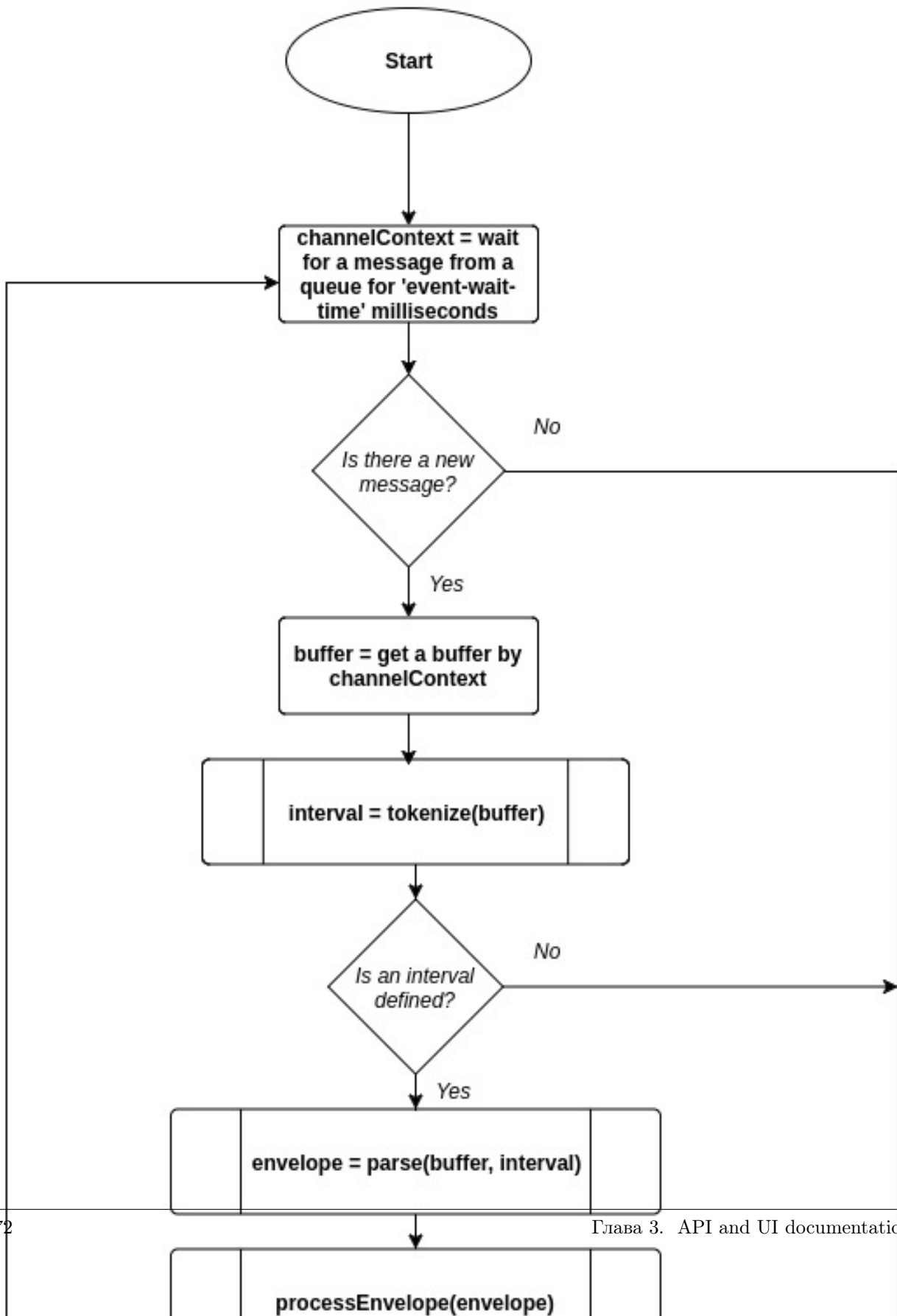
Here is the engine processing flowchart.

Input module waits for new events (messages) from the TCP channel. A wait time period ('event-wait-time') is 1000 ms by default. Once a new data portion is received, the "tokenize" method determines the beginning and the end of the Interval (i.e. significant set of bytes in the incoming flow of bytes).

Once an Interval is set the buffer of incoming data is validated to define whether the Interval contains a message or meaningless data. If the buffer of incoming data is determined as a message it is checked for duplication and a key/id is assigned to it. At this stage the destination where the data will be passed to is attributed to the message (this is T-streams for the input module). The message is processed and passed further into T-streams after a checkpoint is done.

Via T-streams the message is sent further in the pipeline to the next stage of processing (to a Regular/Batch module).

Tip: The engine utilizes methods provided by its module. The description of the methods you can find at the [Modules of Input Type](#) section.

Input Module

Regular Streaming Engine

Regular Streaming Engine receives incoming events and then passes data after transformation to the next processing step.

The processing flow for Regular Engine is presented below.

The diagram represents two types of processing:

- for a stateless mode of processing (the left schema). The module does not have a state that means it does not aggregate the incoming data via state mechanism. On each checkpoint the data are sent further in the pipeline.
- for a statefull mode of processing (the right schema). For stateful processing the state is saved at the same moment the checkpoint is performed. Regular module gets data element by element and aggregate it via state mechanism. On each checkpoint all aggregated data will be sent further in the pipeline and the state will be cleared.

Once a module is launched the ‘onInit’ method performs some preparations for the module, for example initializes some auxiliary variables, or checks the state variables on existence. Then a new message is received. The incoming data are ingested via T-streams or Apache Kafka. The messages are processed by two methods appropriate for T-streams or Apache Kafka messages.

A checkpoint is performed after a set period of time or after a set number of messages is received.

If the module has a state the data are stored at the moment of checkpoint. In case of a failure the stored data from the state will be recovered and the module will be restarted.

If there is no state the checkpoint is performed and the cycle starts again from receiving a new message.

Tip: The engine utilizes methods provided by its module. The description of the methods you can find at the [Modules of Regular Type](#) section.

Batch Streaming Engine

Batch Streaming Engine receives incoming events and organizes incoming data into batches, fulfill window operations (i.e. collect batches in a window and shift the window after checkpoint) then sends data after transformation to the next processing step.

The processing flow for Batch engine is presented below.

The diagram represents two types of processing:

- for a stateless mode of processing (the left schema): The module does not have a state that means it does not aggregate the incoming data via state mechanism. On each checkpoint the data are sent further in the pipeline.
- for a statefull mode of processing (the right schema). For stateful processing the state is saved at the same moment the checkpoint is performed. Batch module gets data from Input module batch by batch and aggregate it via state mechanism. On each checkpoint all aggregated data will be sent to Output module and the state will be cleared.

Once a module is launched the ‘onInit’ method performs some preparations for the module, for example, initializes some auxiliary variables, or checks the state variables on existence.

Then a new message is received. The incoming data are ingested via T-streams or Apache Kafka. The messages are processed by two methods appropriate for T-streams or Apache Kafka messages.

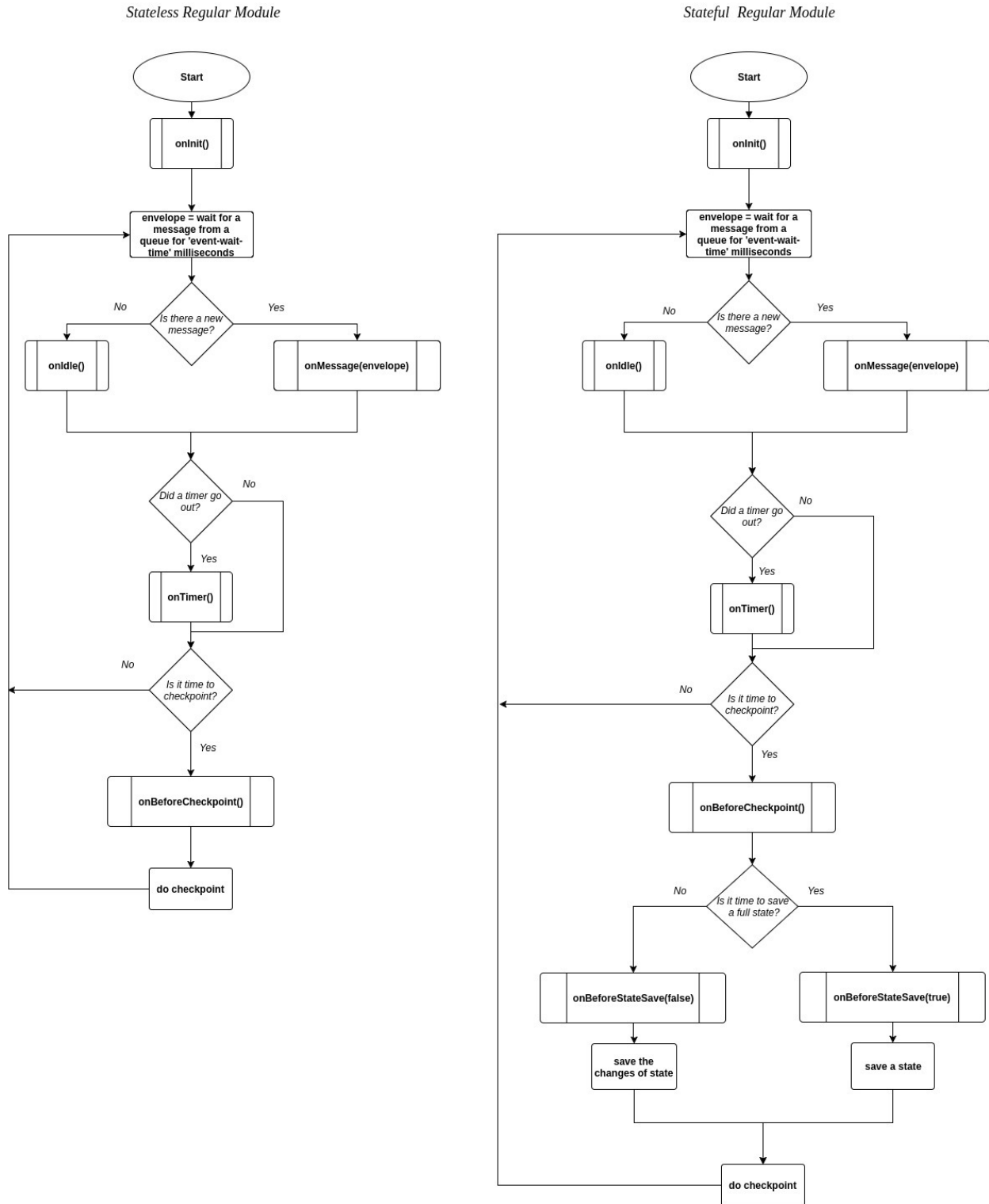
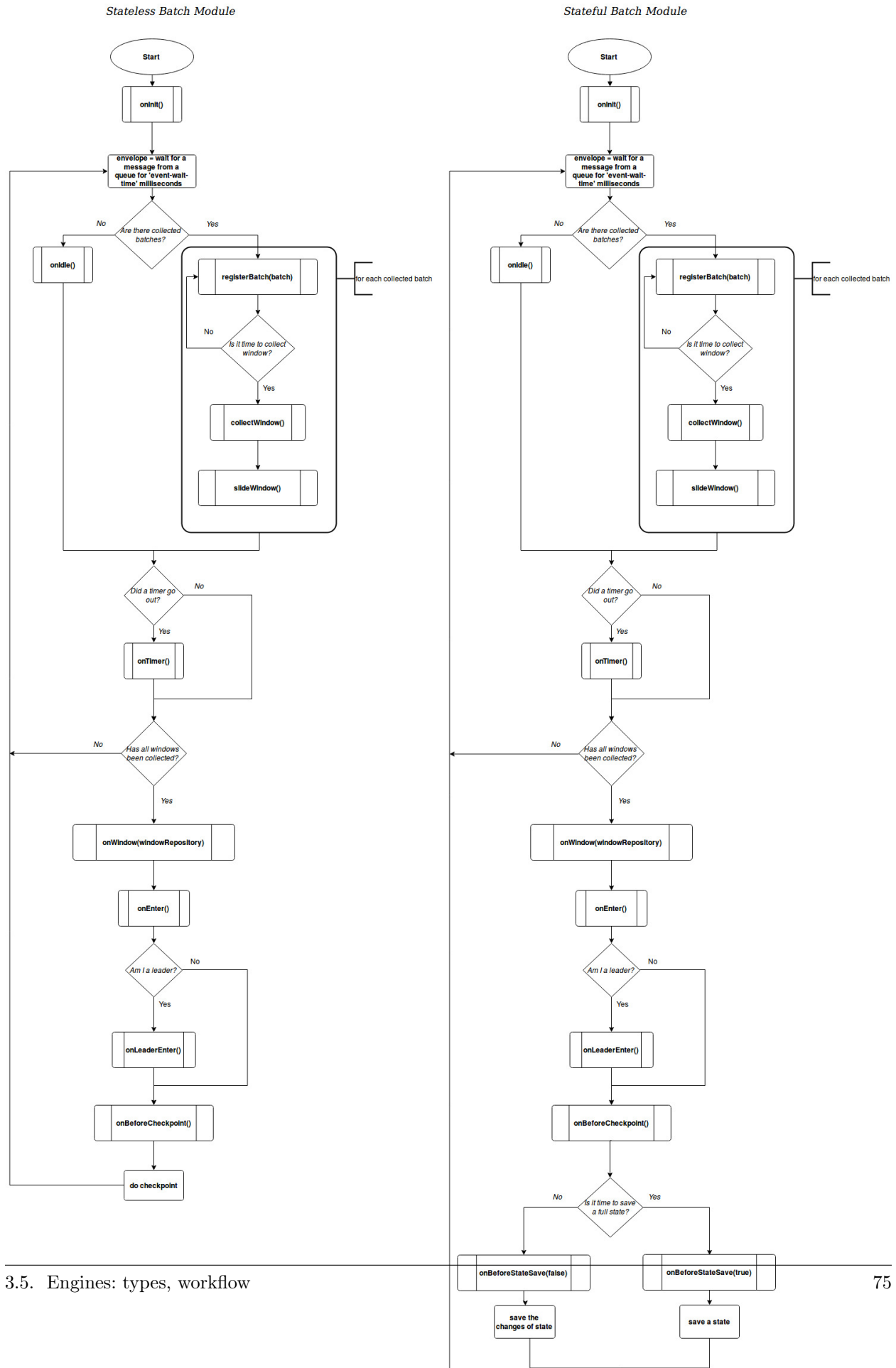


Рис. 3.53: Figure 1.2



Input data are collected in batches. A batch is a minimum data set for a handler to collect the events in the stream. In the module it is a [Batch Collector](#) that is responsible for the logic of collecting batches.

The batches are collected in a window. The number of batches in a window is set in the Instance settings. The engine collects one window per one incoming stream. And, as the module may have one and more incoming streams, then all windows are collected together for processing data in them.

The Batch module allows for intercommunication between tasks that process incoming data. The number of tasks is set in the ‘parallelism’ field of Instance settings. The following handlers are used for synchronizing the tasks’ work. It can be helpful at information aggregation using shared memory, e.g. Hazelcast or any other.

1. “onEnter”: The system awaits for every task to finish the “onWindow” method and then the “onEnter” method of all tasks is invoked.
2. “onLeaderEnter”: The system awaits for every task to finish the “onEnter” method and then the “onLeaderEnter” method of a leader task is invoked.

After the data are processed the checkpoint is performed and the result of processing is sent further into T-streams.

If the module has a state the data are stored at the moment of checkpoint. In case of a failure the stored data from the state will be recovered and the module will be restarted.

If there is no state the checkpoint is performed and the cycle starts again from collecting new messages into batches.

Tip: The engine utilizes methods provided by its module. The description of the methods you can find at the [Modules of Batch Type](#) section.

Output Streaming Engine

Output Streaming Engine handles external output from event processing pipeline to external data destinations (Elasticsearch, JDBC, etc.).

The processing flow for Output Engine is presented below.

It waits for an event (message) in T-streams outcoming from a Regular/Batch module. A wait time period (‘event-wait-time’) is 1000 ms by default. When receiving an envelope of T-streams type, it processes the data transforming it into a data type appropriate for an external datastorage.

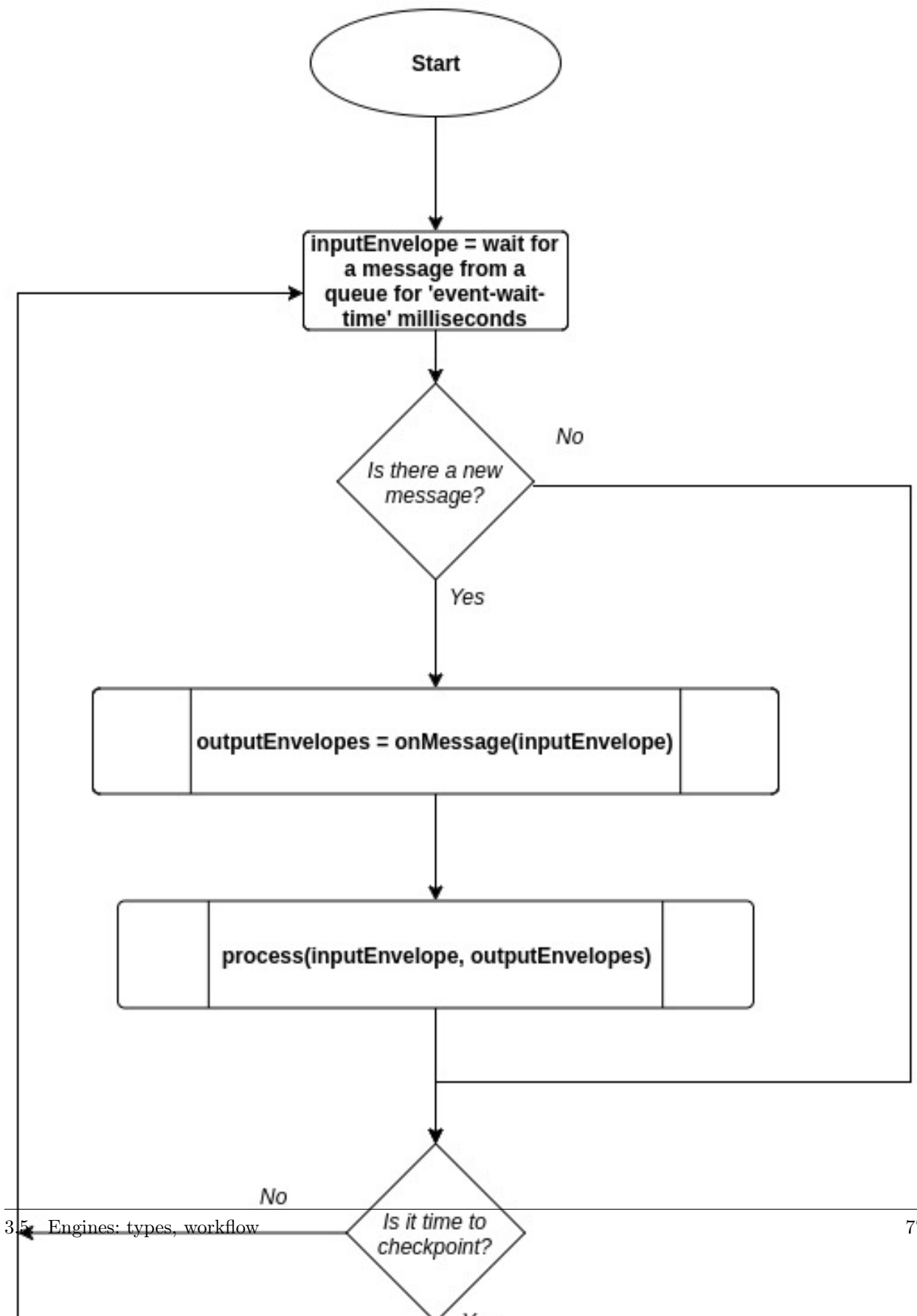
The data are passed to the external storage (Elasticsearch, SQL database, REST, etc.) right after the processing.

To avoid data duplication in the storage, in case of module failure prior to a checkpoint the engine is restarted and incoming messages are written instead of the previously received data. The messages will be written again up to the checkpoint.

After a checkpoint the cycle repeats again starting from receiving a new message.

Tip: The engine utilizes methods provided by its module. The description of the methods you can find at the [Modules of Output Type](#) section.

Output Module



3.6 Custom Module Development Guide

Contents

- Custom Module Development Guide
 - Before Starting With Modules
 - Input Streaming Custom Module
 - Regular Streaming Custom Module
 - Batch Streaming Custom Module
 - Output Streaming Custom Module
 - Hello World Custom Module
 - * Prerequisites
 - * Task description
 - * Basic classes description
 - * Processing description
 - Input module
 - Regular module
 - Validator
 - onMessage
 - onBeforeCheckpoint
 - deserialize
 - specification.json
 - Output module
 - Validator
 - onMessage
 - getOutputEntity
 - specification.json
 - [More Code](#)

In this section, we describe how to write your own module for the Stream Juggler Platform.

The Stream Juggler Platform is a system for your custom module implementation. It allows adjusting the system to your custom aims. Creation of a custom module will not become a challenge for an average developer who knows Scala language as no special tools or services are necessary.

Prior to the module development, please, take a look at the [SJ-Platform Architecture](#) and [Modules: Types, Structure, Pipeline](#) sections.

As a simple refresher, there are three types of modules in the Stream Juggler Platform:

1. Input module

Functions: transformation of raw data into objects, data deduplication.

It works with the stream types:

- input streams: TCP
- output streams: T-streams

2. Processing module

Functions: data transformation and calculation.

- Regular-streaming processing module: processes data event by event;
- Batch-streaming processing module: organizes data into batches and processes it using sliding window.

A processing module works with the stream types:

- input streams: T-streams, Apache Kafka
- output streams: T-streams

3. Output module

Functions: saving data to external data storages (Elasticsearch, SQL database, RESTful).

It works with the stream types:

- input streams: T-streams
- output streams: depends on an external data storage (currently Elasticsearch, SQL database, RESTful types of data storages are supported).

The workflow of the SJ-Platform implies the structure:

1. The processing module receives data for processing from Apache Kafka and T-streams. You also can use TCP as a source, but you will need an input module in this case. The input module handles external inputs, does data deduplication, transforms raw data into objects for T-streams.
2. A processing module performs the main transformation and calculation of data. It accepts data via T-streams and Apache Kafka. The processed data are put into T-streams only. So an output module is required in the next step as we can not get the result data right from T-streams and put them into an external storage.
3. An output module is necessary to transfer the data from T-streams into the result data of the type appropriate for the external storage.

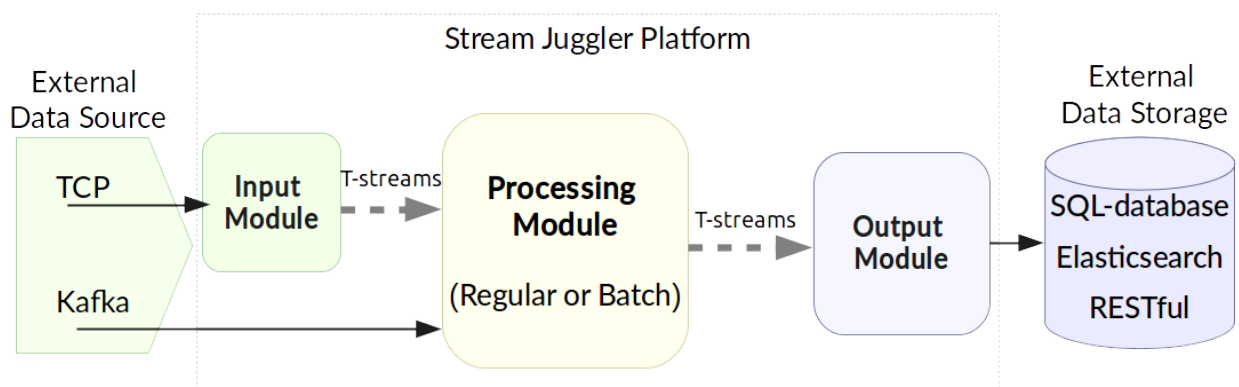


Рис. 3.56: Figure 1.1: Pipeline structure

Below you will find the instructions on custom module creation in Scala.

3.6.1 Before Starting With Modules

The instructions below are given for assembling a JAR file via sbt in Scala. Install sbt following the instructions in [the official documentation](#) .

It is meant that all necessary services are deployed for the module and you know for sure:

- what type of module you want to create;
- what type of inputs/outputs are necessary for it;
- what engine will be used (it should exist in the database);
- what type of external storage you are going to use for result data.

Below, the instructions on each module type are provided.

A [Hello World Custom Module](#) is presented as a tutorial on a module development.

3.6.2 Input Streaming Custom Module

1. Create a new sbt project depending on sj-engine-core library, i.e. use the latest version from the [Apache Maven repository](#) in your build.sbt file. Also mark this dependency as provided. This prevents it from being included in the assembly JAR. For example:

```
libraryDependencies += "com.bws" %% "sj-engine-core" % "1.0" % "provided"
```

2. Create an executor class inheriting the InputStreamingExecutor class and override the necessary methods (see [Modules of Input Type](#)).
3. Create a validator class inheriting the StreamingValidator class and override the validate method if necessary (see [Streaming Validator](#)).
4. Create specification.json in a resources folder and fill it in as shown in the example (see [Json_example_input](#)).
5. Assemble a jar of your module by calling sbt instruction from the project folder, e.g. 'sbt my-input-module/assembly'.
6. Upload the module (via UI or REST).
7. Create an instance of the module (via UI or REST).
8. Launch the instance.

Tip: You can use a module simulator for preliminary testing of executor work (see [Input Engine Simulator](#)).

3.6.3 Regular Streaming Custom Module

1. Create a new sbt project with a dependency on the sj-engine-core library, i.e. use the latest version from the [Apache Maven repository](#) in your build.sbt file. Also mark this dependency as provided. This prevents it from being included in the assembly JAR. For example:

```
libraryDependencies += "com.bws" %% "sj-engine-core" % "1.0" % "provided"
```

2. Create an executor class inheriting RegularStreamingExecutor class and override the necessary methods (see [Modules of Regular Type](#)).

3. Create a validator class inheriting `StreamingValidator` class and override the `validate` method if necessary (see [Streaming Validator](#)).
4. Create `specification.json` in a `resources` folder and fill it in as shown in the example (see `Json_example_regular`).
5. Assemble a jar of your module by calling `sbt` instruction from project folder, e.g. `'sbt my-regular-module/assembly'`.
6. Upload the module (via REST or UI).
7. Create an instance of the module (via REST or UI).
8. Launch the instance.

Tip: You can use a module simulator for preliminary testing of executor work (see [Regular Engine Simulator](#)).

3.6.4 Batch Streaming Custom Module

1. Create a new `sbt` project with a dependency on the `sj-engine-core` library, i.e. use the latest version from the [Apache Maven repository](#) in your `build.sbt` file. Also mark this dependency as provided. This prevents it from being included in the assembly JAR. For example:

```
libraryDependencies += "com.bws" %% "sj-engine-core" % "1.0" % "provided"
```

2. Create an executor class inheriting `BatchStreamingExecutor` class and override the necessary methods (see [Modules of Batch Type](#)).
3. Create a batch collector inheriting `BatchCollector` class and override the required methods (see [Batch Collector](#)).
4. Create a validator class inheriting `StreamingValidator` class and override the `validate` method if necessary (see [Streaming Validator](#)).
5. Create `specification.json` in a `resources` folder and fill it in as shown in the example (see `Json_example_batch`).
6. Assemble a jar of your module by calling `sbt` instruction from project folder, e.g. `'sbt my-batch-module/assembly'`.
7. Upload the module (via REST or UI).
8. Create an instance of the module (via REST or UI).
9. Launch the instance.

Tip: You can use a module simulator for preliminary testing of executor work (see [Batch Engine Simulator](#)).

3.6.5 Output Streaming Custom Module

1. Create a new `sbt` project with a dependency on the `sj-engine-core` library, i.e. use the latest version from the [Apache Maven repository](#) in your `build.sbt` file. Also mark this dependency as provided. This prevents it from being included in the assembly JAR. For example:

```
libraryDependencies += "com.bws" %% "sj-engine-core" % "1.0" % "provided"
```

2. Create an executor class inheriting `OutputStreamingExecutor` class and override the necessary methods (see [Modules of Output Type](#)).
3. Create a validator class inheriting `StreamingValidator` class and override the `validate` method if necessary (see [Streaming Validator](#)).
4. Create `specification.json` in a `resources` folder and fill it in as shown in the example (see `Json_example_output`).
5. Create class of entity that extends `OutputEnvelope`. Override method `getFieldsValue`.
6. Assemble a jar of your module by calling `sbt` instruction from the project folder, e.g. `'sbt my-output-module/assembly'`.
7. Create an index in Elasticsearch and the index mapping, or a table in a database, or deploy some REST service. Name of the index is provided in Elasticsearch service. SQL database stream name is a table name. Elasticsearch stream name is a document type. A full URL to entities of the REST service is `"http://<host>:<port><basePath>/<stream-name>"`.
8. Upload the module (via Rest API or UI).
9. Create an instance of the module (via Rest API or UI).
10. Launch the instance.

Tip: You can use a module simulator for preliminary testing of executor work ([Output Engine Simulator](#)).

3.6.6 Hello World Custom Module

This tutorial explains how to write a module using a simple Hello World example. Let's create a module together!

Prerequisites

First of all, you should:

- follow the deployment process described in `Minimesos_deployment` up to Point 9 inclusive
- OR follow the deployment process described in [Deployment of Required Services on Mesos](#) up to Point 7 inclusive

And remember `<ip>` of the machine where everything is deployed on and the `<port>` of deployed SJ-REST (in `Minimesos` deployment it is written in Point 7 in variable `$address`, in `Mesos` deployment it is written in Point 4 in variable `$address`).

Task description

Let's describe the task to be resolved.

In this example we are going to develop the system to aggregate information about nodes accessibility. Raw data are provided by the `fping` utility.

An example of the `fping` utility usage:

```
fping -l -g 91.221.60.0/23 2>&1 | awk '{printf "%s ", $0; system("echo $(date +%s%N | head -c -7)");}'
```

Here we are going to ping all addresses in particular subnet indefinitely. Result of fping utility execution is a stream of lines which looks like:

```
91.221.60.14 : [0], 84 bytes, 0.46 ms (0.46 avg, 0% loss)
91.221.61.133 : [0], 84 bytes, 3.76 ms (3.76 avg, 0% loss)
<...>
```

We process them via awk utility, just adding current system time to the end of the line:

```
91.221.60.77 : [0], 84 bytes, 0.84 ms (0.84 avg, 0% loss) 1499143409312
91.221.61.133 : [0], 84 bytes, 0.40 ms (0.40 avg, 0% loss) 1499143417151
<...>
```

There could be error messages as the output of fping utility which are sent to stdout, that's why all of them look as follows:

```
ICMP Unreachable (Communication with Host Prohibited) from 91.221.61.59 for ICMP Echo sent to 91.221.61.59
↪1499143409313
ICMP Unreachable (Communication with Host Prohibited) from 91.221.61.215 for ICMP Echo sent to 91.221.61.
↪215 1499143417152
<...>
```

As we can see, awk processes them too - so there is also a timestamp at the end of error lines.

So, there could be 2 types of lines:

- Normal response:

```
91.221.61.133 : [0], 84 bytes, 0.40 ms (0.40 avg, 0% loss) 1499143417151
```

And we are interested only in three values from it:

- IP (91.221.60.77),
- response time (0.40 ms),
- timestamp (1499143417151)

- Error response:

```
ICMP Unreachable (Communication with Host Prohibited) from 91.221.61.59 for ICMP Echo sent to 91.221.
↪61.59 1499143409313
```

And we are interested only in two values from it:

- IP (91.221.61.59),
- timestamp (1499143409313)

Everything we receive from 'fping + awk' pipe is going to our configured stream-juggler module, which aggregates all data for every needed amount of time, e.g. for 1 minute, and provides the output like:

```
<timestamp of last response> <ip> <average response time> <total amount of successful packets> <total
↪amount of unreachable responses> <total amount of packets sent>
```

for all IPs for which it has received data at that particular minute.

All output data are going to be sent into Elasticsearch to store them and have an ability to show on a plot (via Kibana).

Basic classes description

Let's create classes for the described input and output data of stream-juggler module.

As we can see, there are common fields - IP and timestamp - in 'ping + awk' outgoing responses. Both are for normal and error responses.

So, we can create an abstract common class:

```
abstract class PingResponse {  
  val ts: Long  
  val ip: String  
}
```

And then extend it by EchoResponse and UnreachableResponse classes:

```
case class EchoResponse(ts: Long, ip: String, time: Double) extends PingResponse  
case class UnreachableResponse(ts: Long, ip: String) extends PingResponse
```

There were two classes for input records. But we need to aggregate data inside our module, so let's create internal class - PingState:

```
case class PingState(lastTimeStamp: Long = 0, totalTime: Double = 0, totalSuccessful: Long = 0,  
  totalUnreachable: Long = 0) {  
  
  // This one method is needed to update aggregated information.  
  def += (pingResponse: PingResponse): PingState = pingResponse match {  
    case er: EchoResponse => PingState(er.ts, totalTime + er.time, totalSuccessful + 1, totalUnreachable)  
    case ur: UnreachableResponse => PingState(ur.ts, totalTime, totalSuccessful, totalUnreachable + 1)  
  }  
  
  // Returns description  
  def getSummary(ip: String): String = {  
    lastTimeStamp.toString + ',' + ip + ',' +  
    {  
      if(totalSuccessful > 0) totalTime / totalSuccessful  
      else 0  
    } + ',' +  
    totalSuccessful + ',' + totalUnreachable  
  }  
}
```

Let's then create an output class (name it PingMetrics), which contains all fields we need:

```
class PingMetrics {  
  var ts: Date = null  
  var ip: String = null  
  var avgTime: Double = 0  
  var totalOk: Long = 0  
  var totalUnreachable: Long = 0  
  var total: Long = 0  
}
```

But there is a condition: an output class should extend OutputEnvelope abstract class of the Stream-Juggler engine:

```
abstract class OutputEnvelope {  
  def getFieldsValue: Map[String, Any]  
}
```


It has one method - `getFieldsValue` - which is needed to obtain `map[fieldName: String -> fieldValue: Any]`. So, we need a set of variables with names of fields. Looks like all of them will be constants, that's why we include them into companion class:

```
object PingMetrics {
  val tsField = "ts"
  val ipField = "ip"
  val avgTimeField = "avg-time"
  val totalOkField = "total-ok"
  val totalUnreachableField = "total-unreachable"
  val totalField = "total"
}
```

And override the `getFieldsValue` method in the following way:

```
class PingMetrics extends OutputEnvelope {

  import PingMetrics._

  var ts: Date = null
  var ip: String = null
  var avgTime: Double = 0
  var totalOk: Long = 0
  var totalUnreachable: Long = 0
  var total: Long = 0

  override def getFieldsValue = {
    Map(
      tsField -> ts,
      ipField -> ip,
      avgTimeField -> avgTime,
      totalOkField -> totalOk,
      totalUnreachableField -> totalUnreachable,
      totalField -> total
    )
  }
}
```

Processing description

Architecture of our solution is going to look like at the schema below:

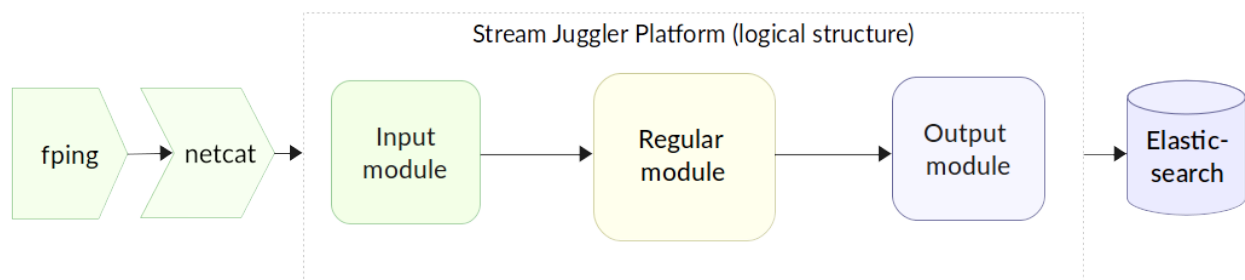


Рис. 3.57: Figure 1.2: Fping example pipeline structure

Netcat appears here because we will send our data to SJ-module via TCP connection.

That is a general description.

If we look deeper into the structure, we will see the following data flow:

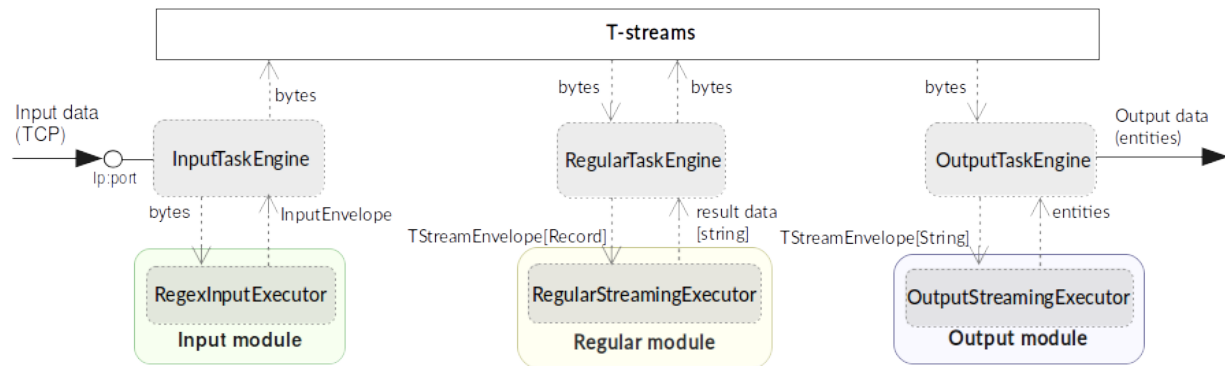


Рис. 3.58: Figure 1.3: Processing in the fping example pipeline

All input data elements are going as a flow of bytes to a particular interface provided by `InputTaskEngine`. That flow is going straight to `RegexInputExecutor` (which implements the `InputStreamingExecutor` interface) and is converted to `InputEnvelope` objects which store all data inside as `Record` (provided by the Apache Avro library).

`InputEnvelope` objects then go back to `InputTaskEngine` which serializes them to the stream of bytes and then sends to T-Streams.

`RegularTaskEngine` deserializes the flow of bytes to `TStreamEnvelope[Record]` which is then put to `RegularStreamingExecutor`.

`RegularStreamingExecutor` processes the received data and returns them as a resulting stream of strings.

`RegularTaskEngine` serializes all the received data to the flow of bytes and puts it back to T-Streams.

Then `OutputTaskEngine` deserializes the stream of bytes from T-Streams to `TStreamEnvelope[String]` and sends it to `OutputStreamingExecutor`. `OutputStreamingExecutor` returns `Entities` back to `OutputTaskEngine`. They are then put to Elasticsearch.

Input module

The Input module is `RegexInputExecutor` (it implements the `InputStreamingExecutor`) and it is provided via the Sonatype repository. Its purpose (in general) is to process an input stream of strings using regexp rules provided by a user and create `InputEnvelope` objects as a result.

The rules are described in `pingstation-input.json`. As we can see, there are rules for each type of input records and each has its own value in the `outputStream` fields: “echo-response” and “unreachable-response”.

So, the `InputEnvelope` objects are put into two corresponding streams.

Regular module

The data from both of these streams are sent to a Regular module. We choose the Regular module instead of the Batch one because we need to process each input element separately. So we define an `Executor` class which implements `RegularStreamingExecutor`:

```
class Executor(manager: ModuleEnvironmentManager) extends RegularStreamingExecutor[Record](manager)
```

A manager (of `ModuleEnvironmentManager` type) here is just a source of information and a point of access to several useful methods: get output stream, get the state (for stateful modules to store some global variables), etc. We use `Record` (Apache Avro) type here as a generic type because output elements of the input module are stored as Avro records.

The data is received from two streams, each of them will have its own name, so let's create the following object to store their names:

```
object StreamNames {
  val unreachableResponseStream = "unreachable-response"
  val echoResponseStream = "echo-response"
}
```

And just import it inside our class:

```
import StreamNames._
```

The `Regular` module gets data from the `Input` module element by element and aggregates them via the state mechanism. On each checkpoint all aggregated data are sent to the `Output` module and the state is cleared.

So we need to obtain the state in our class:

```
private val state = manager.getState
```

To describe the whole logic we need to override the following methods:

- `onMessage(envelope: TStreamEnvelope[T])` - to get and process messages;
- `onBeforeCheckpoint()` - to send everything gained further;
- `deserialize(bytes: Array[Byte])` - to deserialize flow of bytes from T-Streams into `Record` (Apache Avro) correctly.

Validator

An instance contains an options field of a `String` type. This field is used to send some configuration to the module (for example, via this field regex rules are passed to `InputModule`). This field is described in JSON-file for a particular module.

When this field is used, its validation is handled with `Validator` class. So it is necessary to describe the `Validator` class here.

The `Input` module uses an options field to pass Avro Schema to the `Regular` module. That's why we create `Validator` class in the following way (with constant field in singleton `OptionsLiterals` object):

```
object OptionsLiterals {
  val schemaField = "schema"
}
class Validator extends StreamingValidator {

  import OptionsLiterals._

  override def validate(options: String): ValidationInfo = {
    val errors = ArrayBuffer[String]()

    val jsonSerializer = new JsonSerializer
```

```
val mapOptions = jsonSerializer.deserialize[Map[String, Any]](options)
mapOptions.get(schemaField) match {
  case Some(schemaMap) =>
    val schemaJson = jsonSerializer.serialize(schemaMap)
    val parser = new Schema.Parser()
    if (Try(parser.parse(schemaJson)).isFailure)
      errors += s"'$schemaField' attribute contains incorrect avro schema"

    case None =>
      errors += s"'$schemaField' attribute is required"
}

ValidationInfo(errors.isEmpty, errors)
}
```

And then just try to parse the schema.

onMessage

The onMessage method is called every time the Executor receives an envelope.

As we remember, there are two possible types of envelopes in our example: echo-response and unreachable-response, which are stored in two different streams.

We obtain envelopes from both of them and the name of the stream is stored in the envelope.stream field:

```
val maybePingResponse = envelope.stream match {
  case `echoResponseStream` =>
    // create EchoResponse and fill its fields
  case `unreachableResponseStream` =>
    // create UnreachableResponse and fill its fields
  case stream =>
    // if we receive something we don't need
}
```

The envelope.data.head field contains all data we need and its type is Record (Apache Avro).

So the next step is obvious - we will use Try scala type to cope with possibility of a wrong or a corrupted envelope:

```
val maybePingResponse = envelope.stream match {
  case `echoResponseStream` =>
    Try {
      envelope.data.dequeueAll(_ => true).map { data =>
        EchoResponse(data.get(FieldNames.timestamp).asInstanceOf[Long],
          data.get(FieldNames.ip).asInstanceOf[Utf8].toString,
          data.get(FieldNames.latency).asInstanceOf[Double])
      }
    }

  case `unreachableResponseStream` =>
    Try {
      envelope.data.dequeueAll(_ => true).map { data =>
        UnreachableResponse(data.get(FieldNames.timestamp).asInstanceOf[Long],
          data.get(FieldNames.ip).asInstanceOf[Utf8].toString)
      }
    }
}
```

```

    }

    case stream =>
      logger.debug("Received envelope has incorrect stream field: " + stream)
      Failure(throw new Exception)
  }

```

And then just process maybePingResponse variable to obtain actual pingResponse or to finish execution in case of an error:

```
val pingResponses = maybePingResponse.get
```

After unfolding an envelope we need to store it (and to aggregate information about each host). As mentioned, we will use state mechanism for this purpose.

The following code does what we need:

```

if (state.isExist(pingResponse.ip)) {
  // If IP already exists, we need to get its data, append new data and put everything back (rewrite)
  val pingEchoState = state.get(pingResponse.ip).asInstanceOf[PingState]
  state.set(pingResponse.ip, pingEchoState + pingResponse)
} else {
  // Otherwise - just save new one pair (IP - PingState)
  state.set(pingResponse.ip, PingState() + pingResponse)
}

```

So, here is the whole code that we need to process a new message in our Executor class:

```

class Executor(manager: ModuleEnvironmentManager) extends RegularStreamingExecutor[Record](manager) {
  private val state = manager.getState
  override def onMessage(envelope: TStreamEnvelope[Record]): Unit = {
    val maybePingResponse = envelope.stream match {
      case `echoResponseStream` =>
        Try {
          envelope.data.dequeueAll(_ => true).map { data =>
            EchoResponse(data.get(FieldNames.timestamp).asInstanceOf[Long],
              data.get(FieldNames.ip).asInstanceOf[Utf8].toString,
              data.get(FieldNames.latency).asInstanceOf[Double])
          }
        }

      case `unreachableResponseStream` =>
        Try {
          envelope.data.dequeueAll(_ => true).map { data =>
            UnreachableResponse(data.get(FieldNames.timestamp).asInstanceOf[Long],
              data.get(FieldNames.ip).asInstanceOf[Utf8].toString)
          }
        }

      case stream =>
        logger.debug("Received envelope has incorrect stream field: " + stream)
        Failure(throw new Exception)
    }

    val pingResponses = maybePingResponse.get

    pingResponses.foreach { pingResponse =>
      if (state.isExist(pingResponse.ip)) {

```

```
    val pingEchoState = state.get(pingResponse.ip).asInstanceOf[PingState]
    state.set(pingResponse.ip, pingEchoState + pingResponse)
  } else {
    state.set(pingResponse.ip, PingState() + pingResponse)
  }
}
}
```

onBeforeCheckpoint

A onBeforeCheckpoint method calling condition is described in ‘pingstation-input.json’ configuration file:

```
"checkpointMode" : "every-nth",
"checkpointInterval" : 10
```

So we can see it will be called after each 10 responses received in the onMessage method.

First of all we need to obtain an output object to send all data into. In this example we will use RoundRobinOutput because it is not important for us in this example how data would be spread out among partitions:

```
val outputName: String = manager.outputs.head.name
val output: RoundRobinOutput = manager.getRoundRobinOutput(outputName)
```

In manager.outputs all output streams are returned. In this project there would be only one output stream, so we just get its name. And then we obtain RoundRobinOutput object for this stream via getRoundRobinOutput.

Then we use the state.getAll() method to obtain all data we’ve collected.

It returns Map[String, Any]. We use the following code to process all elements:

```
// Second one element here is converted to PingState type and is put to output object via getSummary_
↪ conversion to string description.
case (ip, pingState: PingState) =>
  output.put(pingState.getSummary(ip))

case _ =>
  throw new IllegalStateException

Full code of onBeforeCheckpoint method:
override def onBeforeCheckpoint(): Unit = {
  val outputName = manager.outputs.head.name
  val output = manager.getRoundRobinOutput(outputName)

  state.getAll.foreach {
    case (ip, pingState: PingState) =>
      output.put(pingState.getSummary(ip))

    case _ =>
      throw new IllegalStateException
  }

  state.clear
}
```

deserialize

This method is called when we need to correctly deserialize the flow of bytes from T-Streams into Record (Apache Avro).

There is an AvroSerializer class which shall be used for this purpose. But due to the features of Avro format we need a schema to do that properly.

Avro schema is stored into manager.options field.

So, the following code listing shows the way of creating AvroSerialiser and obtaining an avro scheme:

```
private val jsonSerializer: JsonSerializer = new JsonSerializer
private val mapOptions: Map[String, Any] = jsonSerializer.deserialize[Map[String, Any]](manager.options)
private val schemaJson: String = jsonSerializer.serialize(mapOptions(schemaField))
private val parser: Parser = new Schema.Parser()
private val schema: Schema = parser.parse(schemaJson)
private val avroSerializer: AvroSerializer = new AvroSerializer
override def deserialize(bytes: Array[Byte]): GenericRecord = avroSerializer.deserialize(bytes, schema)
```

specification.json

This file describes the module. Examples of the description can be found at the Json_schema section.

Output module

We define Executor class (in another package), which extends OutputStreamingExecutor:

```
class Executor(manager: OutputEnvironmentManager) extends OutputStreamingExecutor[String](manager)
```

Manager here (of OutputEnvironmentManager type) is also a point of access to some information but in this example we will not use it.

Type of data sent by Regular module is String that's why this type is used as a template type.

We need to override two methods:

- onMessage(envelope: TStreamEnvelope[String]) - to get and process messages
- getOutputEntity() - to return format of output records

Validator

The Validator class here is empty due to the absence of extra information on how we need to process data from the Regular module.

onMessage

The full code of this method is listed below:

```
override def onMessage(envelope: TStreamEnvelope[String]): mutable.Queue[PingMetrics] = {
  val list = envelope.data.map { s =>
    val data = new PingMetrics()
    val rawData = s.split(",")
```

```
        data.ts = new Date(rawData(0).toLong)
        data.ip = rawData(1)
        data.avgTime = rawData(2).toDouble
        data.totalOk = rawData(3).toLong
        data.totalUnreachable = rawData(4).toLong
        data.total = data.totalOk + data.totalUnreachable
        data
    }

    list
}
```

All data are in the ‘envelope’ data field.

So, for each record in this field we create a new PingMetrics instance and fill in all corresponding fields. Then just return a sequence of these objects.

getOutputEntity

Signature of the method looks like:

```
override def getOutputEntity: Entity[String]
```

It returns instances of Entity[String] - that class contains metadata on OutputEnvelope structure: map (field name -> field type) (Map[String, NamedType[T]]).

In the ‘es-echo-response-1m.json’ file we use the “elasticsearch-output” string as a value of the type field. It means that we use Elasticsearch as output for our SJ-module. Other possible variants are RESTful and SQL databases.

So, for Elasticsearch destination type we shall use an appropriate builder in ‘getOutputEntity’ (there are three of them - one for each type) and just describe all fields we have:

```
override def getOutputEntity: Entity[String] = {
    val entityBuilder = new ElasticsearchEntityBuilder()
    val entity: Entity[String] = entityBuilder
        .field(new DateField(tsField))
        .field(new JavaStringField(ipField))
        .field(new DoubleField(avgTimeField))
        .field(new LongField(totalOkField))
        .field(new LongField(totalUnreachableField))
        .field(new LongField(totalField))
        .build()
    entity
}
```

specification.json

This file describes the module. Examples of description can be found at the [Json_schema](#) section.

3.6.7 More Code

More module examples you can find at the [fping example project](#) and [sFlow example project](#) GitHub repositories.

3.7 Testing Modules on Simulators

Stream Juggler Platform provides a user with a range of simulators for module testing purposes. A simulator is a ready-to-use environment that allows you to rapidly test modules in the development process.

Four types of simulators are provided - one per each module type. Choose which you want to use in accordance with the type of a module you want to test. And follow the instructions below.

3.7.1 Input Engine Simulator

It is a class for testing the implementation of an `input module` (Executor).

Simulator imitates the behavior of the `Input Streaming Engine`: it sends byte buffer to Executor, gets input envelopes from it, checks envelopes for duplicates (if it is necessary), and builds `Output Data`.

To use the simulator you need to add the dependency to the build.sbt:

```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
libraryDependencies += "com.bwsw" %% "sj-engine-simulators" % "1.0-SNAPSHOT" % "test"
```

Constructor arguments

| Argument | Type | Description |
|-----------------|-----------------------------|--|
| executor* | InputStreamingExecutor | Implementation of the <code>input module</code> under testing |
| evictionPolicy* | InputInstanceEvictionPolicy | A field of an instance (see Create an instance of a module) |
| separator | String | Delimiter between data records (empty string by default) |
| charset | Charset | Encoding of incoming data (UTF-8 by default) |

Note: * - a required field

Important: T - is a type of data created by Executor

The data record is a string that will be parsed by `executor.parse()` to some entity.

The simulator provides the following methods:

- `prepare(record: String)` - writes one data record to a byte buffer.
- `prepare(records: Seq[String])` - writes a collection of data records to a byte buffer.
- `process(duplicateCheck: Boolean, clearBuffer: Boolean = true): Seq[OutputData[T]]` - sends byte buffer to the executor as long as it can tokenize the buffer. The `duplicateCheck` argument indicates that every envelope has to be checked on duplication. The `clearBuffer` argument indicates that byte buffer with data records has to be cleared after processing ('true' by default). The method returns list of `Output Data`.
- `clear()` - clears a byte buffer with data records.

Output Data

This simulator provides information on the processing of incoming data by the `input` module.

| Field | Format | Description |
|-----------------------------|---------------------------------------|---|
| <code>inputEnvelope</code> | <code>Option[InputEnvelope[T]]</code> | Result of the <code>executor.parse()</code> method |
| <code>isNotDuplicate</code> | <code>Option[Boolean]</code> | Indicates that <code>inputEnvelope</code> is not a duplicate if <code>inputEnvelope</code> is defined; otherwise it is 'None' |
| <code>response</code> | <code>InputStreamingResponse</code> | Response that will be sent to a client after an <code>inputEnvelope</code> has been processed |

Important: `T` - is a type of data created by `Executor`

Usage example

E.g. you have your own `Executor` that splits byte buffer by a comma and tries to parse it into 'Integer':

```
class SomeExecutor(manager: InputEnvironmentManager) extends InputStreamingExecutor[Integer](manager) {
  override def tokenize(buffer: ByteBuf): Option[Interval] = { ... }

  override def parse(buffer: ByteBuf, interval: Interval): Option[InputEnvelope[Integer]] = { ... }
}
```

If you want to see what is returned by `Executor` after processing, `Input Engine Simulator` can be used in the following way:

```
val manager: InputEnvironmentManager
val executor = new SomeExecutor(manager)

val hazelcastConfig = HazelcastConfig(600, 1, 1, EngineLiterals.lruDefaultEvictionPolicy, 100)
val hazelcast = new HazelcastMock(hazelcastConfig)
val evictionPolicy = InputInstanceEvictionPolicy(EngineLiterals.fixTimeEvictionPolicy, hazelcast)

val simulator = new InputEngineSimulator(executor, evictionPolicy, ",")
simulator.prepare(Seq("1", "2", "a", "3", "b")) // byte buffer in simulator will contain "1,2,a,3,b,"

val outputDataList = simulator.process(duplicateCheck = true)
println(outputDataList)
```

For more examples, please, visit: [sj-csv-input-test](#), [sj-regex-input-test](#).

3.7.2 Regular Engine Simulator

It is a class for testing the implementation of a `regular` module (`Executor`).

The simulator imitates the behavior of the `Regular Streaming Engine` (stateful mode): it sends envelopes to `Executor`, allows invoking checkpoint's handlers, gets data from output streams and state.

To use the simulator you need to add the dependency to the `build.sbt`:

```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
libraryDependencies += "com.bwsw" %% "sj-engine-simulators" % "1.0-SNAPSHOT" % "test"
```

Constructor arguments

| Argument | Type | Description |
|----------|---|--|
| executor | RegularStreamingExecutor | Implementation of the regular module under testing |
| manager | ModuleEnvironmentManagerMock or StatefulModuleEnvironmentManager (see Module Environment Manager Mock) | |

Important: T - the type of data received by Executor.

Provided methods

- `prepareState(state: Map[String, Any])` - loads state into a state storage.
- `state` - key/value map.
- `prepareTstream(entities: Seq[T], stream: String, consumerName: String = "default-consumer-name"): Long` - creates one t-stream envelope (`TStreamEnvelope[T]` type) and saves it in a local buffer. Returns an ID of the envelope.
- `entities` - list of incoming data elements.
- `stream` - name of a stream with incoming data.
- `consumerName` - name of a consumer ('default-consumer-name' by default).
- `prepareKafka(entity: T, stream: String): Long` - creates one kafka envelope (`KafkaEnvelope[T]` type) and saves it in a local buffer. Returns an ID of that envelope.
- `entity` - an incoming data element.
- `stream` - name of a stream with incoming data.
- `prepareKafka(entities: Seq[T], stream: String): Seq[Long]` - creates a list of kafka envelopes (`KafkaEnvelope[T]` type) - one envelope for each element from entities, and saves it in a local buffer. Returns a list of envelopes' IDs.
- `entities` - list of incoming data elements.
- `stream` - name of a stream with incoming data.
- `process(envelopesNumberBeforeIdle: Int = 0, clearBuffer: Boolean = true): SimulationResult` - sends all envelopes from local buffer and returns output streams and state (see [Simulation Result](#)).
- `envelopesNumberBeforeIdle` - number of envelopes after which `executor.onIdle()` will be invoked ('0' by default). '0' means that `executor.onIdle()` will never be called.
- `clearBuffer` - indicates that all envelopes will be removed from a local buffer after processing.
- `beforeCheckpoint(isFullState: Boolean): SimulationResult` - imitates the behavior of the [Regular Streaming Engine](#) before checkpoint: invokes `executor.onBeforeCheckpoint()`, then invokes `executor.onBeforeStateSave(isFullState)` and returns output streams and state (see [Simulation Result](#)).
- `isFullState` - the flag denotes that either the full state ('true') or a partial change of state ('false') is going to be saved.
- `timer(jitter: Long): SimulationResult` - imitates that a timer went out (invokes `executor.onTimer(jitter)`).
- `jitter` - a delay between a real response time and an invocation of this handler.

- `clear()` - removes all envelopes from a local buffer.

Usage Example

E.g. you have your own `Executor` that takes strings and calculates their length:

```
class SomeExecutor(manager: ModuleEnvironmentManager) extends RegularStreamingExecutor[String](manager) {
  private val state = manager.getState
  private val output = manager.getRoundRobinOutput("output")

  override def onIdle(): Unit = {
    val idleCalls = state.get("idleCalls").asInstanceOf[Int]
    state.set("idleCalls", idleCalls + 1)
    val symbols: Integer = state.get("symbols").asInstanceOf[Int]
    output.put(symbols)
  }

  override def onMessage(envelope: KafkaEnvelope[String]): Unit = {
    val symbols = state.get("symbols").asInstanceOf[Int]
    val length = envelope.data.length
    state.set("symbols", symbols + length)
  }

  override def onMessage(envelope: TStreamEnvelope[String]): Unit = {
    val symbols = state.get("symbols").asInstanceOf[Int]
    val length = envelope.data.toList.mkString.length
    state.set("symbols", symbols + length)
  }
}
```

If you want to see what the executor puts into an output stream and to the state after processing, `Regular Engine Simulator` can be used in the following way:

```
val stateSaver = mock(classOf[StateSaverInterface])
val stateLoader = new StateLoaderMock
val stateService = new RAMStateService(stateSaver, stateLoader)
val stateStorage = new StateStorage(stateService)
val options = ""
val output = new TStreamStreamDomain("out", mock(classOf[TStreamServiceDomain]), 3, tags = Array("output"
↪))
val tStreamsSenderThreadMock = new TStreamsSenderThreadMock(Set(output.name))
val manager = new ModuleEnvironmentManagerMock(stateStorage, options, Array(output), ↪
↪tStreamsSenderThreadMock)
val executor: RegularStreamingExecutor[String] = new SomeExecutor(manager)
val tstreamInput = "t-stream-input"
val kafkaInput = "kafka-input"

val simulator = new RegularEngineSimulator(executor, manager)
simulator.prepareState(Map("idleCalls" -> 0, "symbols" -> 0))
simulator.prepareTstream(Seq("ab", "c", "de"), tstreamInput)
simulator.prepareKafka(Seq("fgh", "g"), kafkaInput)
simulator.prepareTstream(Seq("ijk", "lm"), tstreamInput)

val envelopesNumberBeforeIdle = 2
val results = simulator.process(envelopesNumberBeforeIdle)
println(results)
```

`println(results)` will print:

```
SimulationResult(ArrayBuffer(StreamData(out,List(PartitionData(0,List(8)), PartitionData(1,List(14))))),
↪Map(symbols -> 14, idleCalls -> 2))
```

The mock method is from the `org.mockito.Mockito.mock` library.

To see more examples, please, visit [fping example project repository](#).

3.7.3 Batch Engine Simulator

It is a class for testing the implementation of a [batch module](#) (Executor).

The simulator imitates the behavior of the [Batch Streaming Engine](#) (stateful mode): it collects data envelopes in batches, then collects batches in a window, sends data in a window to the Executor, allows invoking checkpoint's handlers, gets data from output streams and state.

To use simulator you need to add this dependency to the build.sbt:

```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
libraryDependencies += "com.bwsw" %% "sj-engine-simulators" % "1.0-SNAPSHOT" % "test"
```

Constructor arguments

| Argument | Type | Description |
|----------------|--------------------------|--|
| executor | BatchStreamingExecutor | Implementation of the batch module under testing |
| manager | ModuleEnvironmentManager | Mock for StatefulModuleEnvironmentManager (see Module Environment Manager Mock) |
| batchCollector | BatchCollector | Implementation of Batch Collector |

Important: T - the type of data received by Executor

Provided methods

- `prepareState(state: Map[String, Any])` - loads state into a state storage.
- `state` - key/value map.
- `prepareTstream(entities: Seq[T], stream: String, consumerName: String = "default-consumer-name")`: Long - creates one t-stream envelope (TStreamEnvelope[T] type) and saves it in a local buffer. Returns an ID of the envelope.
- `entities` - the list of incoming data elements.
- `stream` - the name of a stream with incoming data.
- `consumerName` - the name of a consumer ('default-consumer-name' by default).
- `prepareKafka(entity: T, stream: String)`: Long - creates one kafka envelope ('KafkaEnvelope[T]' type) and saves it in a local buffer. Returns an ID of that envelope.
- `entity` - an incoming data element.
- `stream` - the name of a stream with incoming data.

- `prepareKafka(entities: Seq[T], stream: String): Seq[Long]` - creates a list of kafka envelopes ('KafkaEnvelope[T]' type) - one envelope for each element from entities, and saves it in a local buffer. Returns a list of envelopes' IDs.
- `entities` - the list of incoming data elements.
- `stream` - the name of a stream of incoming data.
- `process(batchesNumberBeforeIdle: Int = 0, window: Int, slidingInterval: Int, saveFullState: Boolean = false, removeProcessedEnvelopes: Boolean = true): BatchSimulationResult` - sends all envelopes from local buffer and returns output streams, state and envelopes that haven't been processed (see [Batch Simulation Result](#)). This method retrieves batches using `batchCollector`. Then it creates a window repository and invokes the `Executor` methods for every stage of the processing cycle. The methods are invoked in the following order: `onWindow`, `onEnter`, `onLeaderEnter`, `onBeforeCheckpoint`, `onBeforeStateSave`. At the end of this method all envelopes will be removed from `batchCollector`.
- `batchesNumberBeforeIdle` - the number of retrieved batches between invocations of `executor.onIdle()` ('0' by default). '0' means that `executor.onIdle()` will never be called.
- `window` - count of batches that will be contained into a window (see [Batch-streaming instance fields](#)).
- `slidingInterval` - the interval at which a window will be shifted (count of processed batches that will be removed from the window) (see [Batch-streaming instance fields](#)).
- `saveFullState` - the flag denotes that either the full state ('true') or a partial change of the state ('false') is going to be saved after every checkpoint.
- `removeProcessedEnvelopes` - indicates that all processed envelopes will be removed from a local buffer after processing.
- `beforeCheckpoint(isFullState: Boolean): SimulationResult` - imitates the behavior of the [Batch Streaming Engine](#) before checkpoint: invokes `executor.onBeforeCheckpoint()`, then invokes `executor.onBeforeStateSave(isFullState)` and returns output streams and state (see [Simulation Result](#)).
- `isFullState` - the flag denotes that either the full state ('true') or partial changes of state ('false') are going to be saved.
- `timer(jitter: Long): SimulationResult` - imitates that a timer went out (invokes `executor.onTimer(jitter)`).
- `jitter` - the delay between a real response time and an invocation of this handler.
- `clear()` - removes all envelopes from a local buffer.

Batch Simulation Result

After invocation of the `process` method some envelopes could remain not processed by `Executor` when there are not enough batches for completing a window.

`case class BatchSimulationResult(simulationResult: SimulationResult, remainingEnvelopes: Seq[Envelope])` - contains output streams, state (see [Simulation Result](#)) (`simulationResult`) and envelopes that haven't been processed (`remainingEnvelopes`).

Usage Example

E.g. you have your own `Executor` that takes strings and calculates their length:

```
class SomeExecutor(manager: ModuleEnvironmentManager) extends BatchStreamingExecutor[String](manager) {  
  private val state = manager.getState  
  private val output = manager.getRoundRobinOutput("out")  
}
```

```

override def onIdle(): Unit = {
  val idleCalls = state.get("idleCalls").asInstanceOf[Int]
  state.set("idleCalls", idleCalls + 1)
}

override def onWindow(windowRepository: WindowRepository): Unit = {
  val symbols = state.get("symbols").asInstanceOf[Int]

  val batches = {
    if (symbols == 0)
      windowRepository.getAll().values.flatMap(_._batches)
    else
      windowRepository.getAll().values.flatMap(_._batches.takeRight(windowRepository.slidingInterval))
  }

  val length = batches.flatMap(_._envelopes).map {
    case t: TStreamEnvelope[String] =>
      t.data.dequeueAll(_ => true).mkString
    case k: KafkaEnvelope[String] =>
      k.data
  }.mkString.length
  state.set("symbols", symbols + length)
}

override def onBeforeCheckpoint(): Unit = {
  val symbols: Integer = state.get("symbols").asInstanceOf[Int]
  output.put(symbols)
}
}

```

If you want to see what the Executor puts into an output stream and into the state after processing, Batch Engine Simulator can be used in the following way:

```

val stateSaver = mock(classOf[StateSaverInterface])
val stateLoader = new StateLoaderMock
val stateService = new RAMStateService(stateSaver, stateLoader)
val stateStorage = new StateStorage(stateService)
val options = ""
val output = new TStreamStreamDomain("out", mock(classOf[TStreamServiceDomain]), 3, tags = Array("output"
↪))
val tStreamsSenderThreadMock = new TStreamsSenderThreadMock(Set(output.name))
val manager = new ModuleEnvironmentManagerMock(stateStorage, options, Array(output), ↪
↪tStreamsSenderThreadMock)
val executor: BatchStreamingExecutor[String] = new SomeExecutor(manager)
val tstreamInput = new TStreamStreamDomain("t-stream-input", mock(classOf[TStreamServiceDomain]), 1)
val kafkaInput = new KafkaStreamDomain("kafka-input", mock(classOf[KafkaServiceDomain]), 1, 1)
val inputs = Array(tstreamInput, kafkaInput)

val batchInstanceDomain = mock(classOf[BatchInstanceDomain])
when(batchInstanceDomain.getInputsWithoutStreamMode).thenReturn(inputs.map(_._name))

val batchCollector = new SomeBatchCollector(batchInstanceDomain, ↪
↪mock(classOf[BatchStreamingPerformanceMetrics]), inputs)

val simulator = new BatchEngineSimulator(executor, manager, batchCollector)
simulator.prepareState(Map("idleCalls" -> 0, "symbols" -> 0))
simulator.prepareTstream(Seq("a", "b"), tstreamInput.name)

```

```
simulator.prepareTstream(Seq("c", "de"), tstreamInput.name)
simulator.prepareKafka(Seq("fgh", "g"), kafkaInput.name)
simulator.prepareTstream(Seq("ijk", "lm"), tstreamInput.name)
simulator.prepareTstream(Seq("n"), tstreamInput.name)
simulator.prepareKafka(Seq("p", "r", "s"), kafkaInput.name)

val batchesNumberBeforeIdle = 2
val window = 4
val slidingInterval = 2
val results = simulator.process(batchesNumberBeforeIdle, window, slidingInterval)

println(results)
```

println(results) will print:

```
BatchSimulationResult(SimulationResult(List(StreamData(out,List(PartitionData(0,List(17))))),Map(symbols -> ↵
↵17, idleCalls -> 4)),ArrayBuffer(<last envelope>))
```

<last-envelope> is a `KafkaEnvelope[String]` that contains string “s”.

The mock method is from the `org.mockito.Mockito.mock` library.

`SomeBatchCollector` is an example of the `BatchCollector` implementation. The `getBatchesToCollect` method returns all nonempty batches, `afterEnvelopeReceive` counts envelopes in batches, `prepareForNextCollecting` resets counters.

Accumulation of batches is implemented in `BatchCollector`:

```
class SomeBatchCollector(instance: BatchInstanceDomain,
    performanceMetrics: BatchStreamingPerformanceMetrics,
    inputs: Array[StreamDomain])
    extends BatchCollector(instance, performanceMetrics, inputs) {
  private val countOfEnvelopesPerStream = mutable.Map(instance.getInputsWithoutStreamMode.map(x => (x, ↵
↵0)): _*)

  def getBatchesToCollect(): Seq[String] =
    countOfEnvelopesPerStream.filter(x => x._2 > 0).keys.toSeq

  def afterEnvelopeReceive(envelope: Envelope): Unit =
    countOfEnvelopesPerStream(envelope.stream) += 1

  def prepareForNextCollecting(streamName: String): Unit =
    countOfEnvelopesPerStream(streamName) = 0
}
```

For more examples, please, visit [sFlow example project repository](#).

3.7.4 Output Engine Simulator

It is a class for testing the implementation of the `output module` (Executor).

Simulator imitates the behavior of the `Output Streaming Engine`: it sends transactions to the Executor, gets output envelopes from it and builds requests for loading data to an output service. Simulator uses `Output Request Builder` to build requests.

To use the simulator you need to add the dependency to the `build.sbt`:


```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
libraryDependencies += "com.bwsj" %% "sj-engine-simulators" % "1.0-SNAPSHOT" % "test"
```

Constructor arguments

| Argument | Type | Description |
|----------------------|---|---|
| executor | OutputStreamingExecutor | Implementation of the output module under testing |
| outputRequestBuilder | OutputRequestBuilder[OT] (see Output Request Builder) | Builder of requests for output service |
| manager | OutputEnvironmentManager | Instance of the OutputEnvironmentManager used by Executor |

Important:

- IT - the type of data received by Executor
- OT - the type of requests that outputRequestBuilder creates. The type depends on the type of output service (see “Request format” column of the table in the [Output Request Builder](#) section).

The simulator provides the following methods:

- prepare(entities: Seq[IT], stream: String = "default-input-stream consumerName: String = "default-consumer-name"): Long - takes a collection of data (entities argument), creates one transaction (TStreamEnvelope[IT] type) with the stream name that equals to the value of the stream parameter, saves them in a local buffer and returns an ID of the transaction. The consumerName argument has a default value (“default-consumer-name”). You should define it only if the executor uses consumerName from TStreamEnvelope.
- process(clearBuffer: Boolean = true): Seq[OT] - sends all transactions from local buffer to Executor by calling the onMessage method for each transaction, gets output envelopes and builds requests for output services. The clearBuffer argument indicates that local buffer with transactions have to be cleared after processing. That argument has a default value “true”.
- clear() - clears local buffer that contains transactions.

Simulator has the beforeFirstCheckpoint flag that indicates that the first checkpoint operation has not been performed. Before the first checkpoint operation the Simulator builds a delete request for each incoming transaction (in the process method). beforeFirstCheckpoint can be changed automatically by calling manager.initiateCheckpoint() inside the Executor, or manually.

Output Request Builder

It provides the following methods to build requests based on an output envelope for a specific output service:

- buildInsert - builds a request to insert data
- buildDelete - builds a request to delete data

There are three implementations of the OutputRequestBuilder for each type of output storage:

| Classname | Request format | Output storage type |
|--------------------|--------------------------------------|---------------------|
| EsRequestBuilder | String | Elasticsearch" |
| JdbcRequestBuilder | PreparedStatementMock | SQL database |
| RestRequestBuilder | org.eclipse.jetty.client.api.Request | RESTful service |

Note: Constructors of the `EsRequestBuilder` and the `JdbcRequestBuilder` takes the `outputEntity` argument. It should be created using the `executor.getOutputEntity` method.

Usage example

E.g. you have your own `Executor`, that takes pairs `(Integer, String)` and puts them in Elasticsearch:

```
class SomeExecutor(manager: OutputEnvironmentManager)
  extends OutputStreamingExecutor[(Integer, String)](manager) {
  override def onMessage(envelope: TStreamEnvelope[(Integer, String)]): Seq[OutputEnvelope] = { ... }
  override def getOutputEntity: Entity[String] = { ... }
}
```

If you want to see what `Executor` returns after processing and what requests are used to save processed data, `Output Engine Simulator` can be used in the following way:

```
val manager: OutputEnvironmentManager
val executor = new SomeExecutor(manager)

val requestBuilder = new EsRequestBuilder(executor.getOutputEntity)
val simulator = new OutputEngineSimulator(executor, requestBuilder, manager)
simulator.prepare(Seq((1, "a"), (2, "b")))
simulator.prepare(Seq((3, "c")))
val requestsBeforeFirstCheckpoint = simulator.process()
println(requestsBeforeFirstCheckpoint)

// "perform" the first checkpoint
simulator.beforeFirstCheckpoint = false
simulator.prepare(Seq((4, "d"), (5, "e")))
val requestsAfterFirstCheckpoint = simulator.process()
println(requestsAfterFirstCheckpoint)
```

`requestsBeforeFirstCheckpoint` will contain delete and insert requests, `requestsAfterFirstCheckpoint` will contain insertion requests only.

To see more examples, please, examine the following sections: [fping example project repository](#), [sj example project repository](#)

3.7.5 Objects For Simulators With States

Under this section the class of objects used for Simulators with states is described. These Simulators are [Regular Engine Simulator](#) and [Batch Engine Simulator](#).

Simulation Result

case class `SimulationResult`(streamDataList: Seq[StreamData], state: Map[String, Any]) - contains data elements for each output stream and the state at a certain point in time.

case class StreamData(stream: String, partitionDataList: Seq[PartitionData]) - contains data items that have been sent to an output stream.

case class PartitionData(partition: Int, dataList: Seq[AnyRef]) - contains data elements that have been sent to an output stream partition.

Module Environment Manager Mock

It is a mock to manage the environment of a module (regular or batch) that has got a state.

It creates [Partitioned Output Mock](#) and [Round Robin Output Mock](#) to save the information on where the data would be transferred.

Constructor arguments

| Argument | Type | Description |
|--------------|----------------------------|---|
| stateStorage | StateStorage | A storage of the state |
| options | String | User-defined instance options |
| outputs | Array[TStreamStreamDomain] | The list of instance output streams |
| senderThread | TStreamsSenderThreadMock | The mock for thread for sending data to the T-Streams service (described below) |
| fileStorage | FileStorage | A file storage (mocked by default) |

T-streams Sender Thread Mock

Contains a collection of output elements (see [Simulation Result](#)).

Constructor arguments

| Argument | Type | Description |
|----------|-------------|----------------------|
| streams | Set[String] | Storage of the state |

Provided methods:

- def getStreamDataList: Seq[StreamData] - returns streams with data.
- def prepareToCheckpoint(): Unit - removes data from streams.

Module Output Mocks

They puts data into TStreamsSenderThreadMock.

Partitioned Output Mock

The mock of an output stream that puts data into a specific partition.

Provided methods:

- put(data: AnyRef, partition: Int) - puts data into a specific partition.

Round Robin Output Mock

The mock of an output stream that puts data using the round-robin policy.

Provided methods:

- `put(data: AnyRef)` - puts data into a next partition.

3.8 Streams in SJ-Platform

SJ-Platform enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

3.8.1 Stream Conception in SJ-Platform

The streaming component is essential in SJ-Platform. The data are fed to the system, transferred between modules and exported to an external storage via streams.

The system receives data via TCP or Apache Kafka. Within SJ-Platform modules exchange data via T-streams. The results are exported to an external storage via output streams which type is determined by the type of the storage. For now, Elasticsearch, SQL-database and RESTful output stream types are supported.

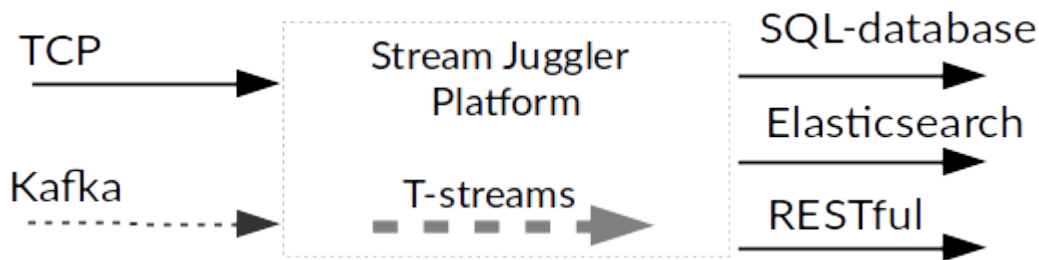


Рис. 3.59: Figure 1.1: I/O Streams in SJ-Platform

The table below shows what types of streams are required as input/output streams for a particular module type.

| Module type | Input stream | Output stream |
|---------------|---------------------------|--|
| Input | TCP | T-streams |
| Regular/Batch | T-streams Apache Kafka | T-streams |
| Output | T-streams | Elasticsearch SQL database RESTful |

Input Streams

The data can be received by the system from different sources. Currently, the platform supports obtaining data from Apache Kafka and via TCP connections.

SJ-Platform supports [Apache Kafka](#) as a standard providing a common interface for the integration of many applications.

In case of using TCP as a source, there arises a need in an input module to transform the input data into a stream and transfer them for further processing. At the project [repository](#) two ready-to-use input modules are available for users - a CSV input module and a Regex input module - that transform text data into a message format acceptable for T-streams.

Internal Streams

Within the platform, the data are transferred to and from modules via transactional streams or T-streams. It is a message broker which is native to SJ-Platform and designed primarily for exactly-once processing. More information on T-streams can be found at the [project site](#). Some general information on T-streams you can find below.

The easiest way to try T-streams and dive into basic operations with T-streams is to download [T-streams-hello](#). It demonstrates basic operations which help to understand how to use T-streams in general.

The image below illustrates the data design format in T-streams. The stream consists of partitions. Each partition holds a number of transactions with data elements inside.

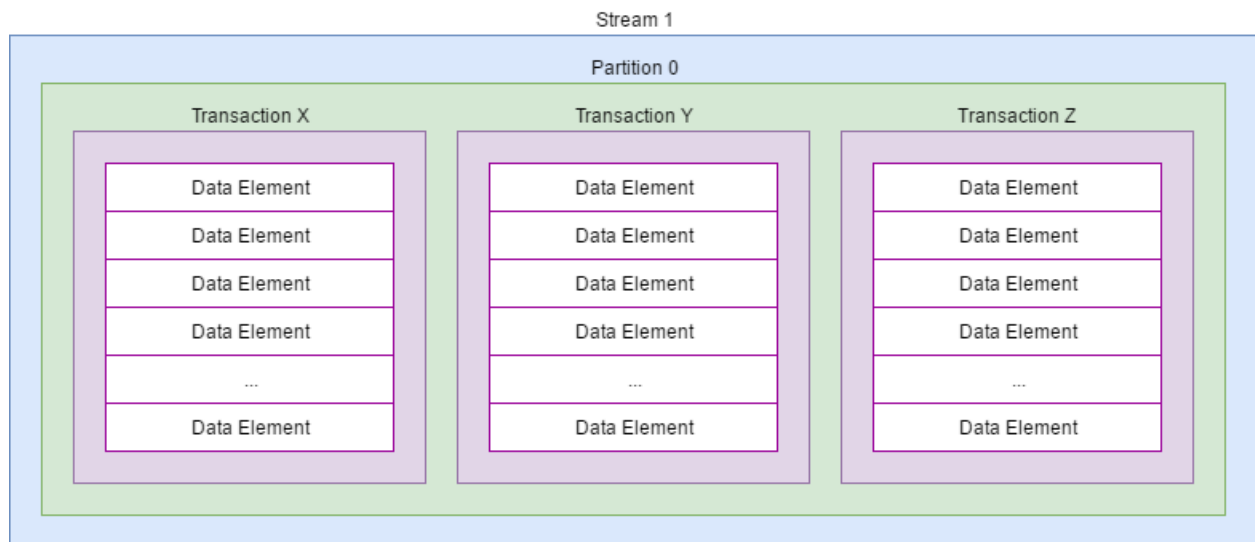


Рис. 3.60: Figure 1.2: T-streams stream structure

Data elements are time-sorted within a transaction.

Transactions are used to write or read data from T-streams. The transaction is also a basic recovery element. This means, that in case of a crash, the processing can recover from a transaction.

Transactions are iterated from the earliest to the latest and data are read from every transaction. After a transaction (or transaction set) is handled properly, the checkpoint is performed which means that even in case of a crash or for another reason the processing will continue with the transaction which is the next to the processed one.

T-streams allows exactly-once processing due to a checkpoint group. The CheckpointGroup is a special entity which allows a developer to do atomic checkpoint for a group of producers and consumers. Several producers and consumers can be bunched up into a group, which can do a checkpoint atomically. This means all producers and consumers in that group fix the current state. This is the key component of exactly-once data processing in SJ-Platform. Find more information in the [official documentation](#).

Output streams

Output streams are streams which are a destination for results.

The processed data are put into T-streams.

The data are retrieved from T-streams using an output module. The output module puts the data from T-streams to the streams of the type which is suitable for the type of the external data storage.

For now, SJ-Platform supports the following types of output streams:

- Elasticsearch, to save data to Elasticsearch;
- SQL database, to save data to JDBC-compatible databases;
- RESTful, to save data to a RESTful storage.

3.8.2 Streaming Infrastructure

Streams need infrastructure: providers and services. They make streams flexible. Streaming flexibility lies in the one-to-many connection between providers and services, streams and modules. One provider works with many services (of various types). One stream type can be used by different module instances. Different types of streams and instances can be created on the base of a common infrastructure. There is no need to duplicate the settings for each individual entity.

A provider is an entity which contains general information to access a physical service.

A service is an entity which contains specific information to access a physical service.

They can be of different types. The types of modules and streams in the pipeline determine the type of providers and services that are necessary in a particular case.

The diagram for interconnections of the platform entities can be useful in selecting the types of providers and services. Please, visit the [Module Instances](#) section for more information.

Firstly, decide what types of modules will perform data transformation and processing in the pipeline. The determined module types will help to clarify which streams are required for them.

Secondly, find in the diagram at [Module Instances](#) what services are necessary for these types of streams.

Finally, when services are determined, it is easy to see what types of providers should be created.

Start creating the infrastructure from providers, then proceed with services and then streams.

Detailed instructions on stream creation can be found in the [Tutorial](#) (for creating infrastructure via REST API) or in the [UI Guide section](#) for creating streams through the Web UI.

3.9 Platform Deployment

In this section you will find the information on how to start working with the Stream Juggler Platform. We offer two options here:

1. to deploy the platform on Mesos cluster, or
2. to run the pre-built VirtualBox® image with the [Fping Example Task](#) solution implemented on the SJ-Platform base.

Please, read the requirements for each option and decide which is more suitable for your aims.

3.9.1 SJ-Platform Deployment on Mesos Cluster

The first option is to deploy SJ-Platform on a cluster. Currently, the deployment on [Apache Mesos](#) as a universal distributed computational engine is supported.

The detailed step-by-step instructions on Stream Juggler Platform deployment on Mesos cluster is provided. You will find here a complete list of requirements and the deployment procedure description. We will deploy and start all the necessary services. The platform will contain no entities. It means you can structure the pipeline corresponding to your aims from scratch. The entities can be added to the platform via [REST API](#) or [the UI](#).

Minimum system requirements are as follows:

- Linux host with 4-8 GB of RAM and 4 CPU cores;
- Docker v17.03 installed (see the official [installation guide](#));
- Java installed (see the official [installation guide](#)).

Overall Deployment Infrastructure

The Stream Juggler Platform works on the base of the following services:

- Resource management is fulfilled via [Apache Mesos](#).
- To start applicable services on Mesos we use [Docker](#).
- [Marathon](#) allows running long-life tasks on Mesos.
- To perform leader election in case the currently leading Marathon instance fails [Apache Zookeeper](#) is used. Zookeeper is also responsible for instance task synchronization for a Batch module.
- Data sources for the platform are [Netty](#) and [T-streams](#) libraries and [Apache Kafka](#).
- We use [MongoDB](#) as a document database that provides high performance and availability. All created platform entities (Providers, Services, Streams, Instances, etc.), as well as configurations, are stored here.
- Elasticsearch 5.5.2, SQL database or a system which provides the RESTful interface are external storages the output data is stored to.

SJ-Platform's backend is written in Scala. The UI is based on Angular 4.0. REST API is written in Akka HTTP.

The section covers the steps on deployment of necessary services and configurations. The system is set up with no entities. You can create them depending on your aims and tasks. The steps on creating platform entities are provided in the UI-Guide.

Deployment of Required Services on Mesos

In the first step, deploy Mesos and other required services.

1. Deploy Mesos, Marathon, Zookeeper. You can follow the instructions in the official [instalation guide](#).

If you are planning to launch an instance with greater value of the parallelizm parameter, i.e. to run tasks on more than 1 nodes, you need to increase the `executor_registration_timeout` parameter for Mesos-slave.

The requirements to Mesos-slave:

- 2 CPUs;

- 4096 memory.

Mesos-slave should support Docker containerizer.

Now make sure you have access to Mesos interface, Marathon interface. Apache Zookeeper should be active.

2. Create a configuration file (config.properties) and JSON files for the physical services - MongoDB, SJ-rest, tts. Please, name them as it is specified here.

mongo.json

Replace <mongo_port> with the port of MongoDB. It should be one of the available Mesos-slave ports.

```
{
  "id":"mongo",
  "container":{"
    "type":"DOCKER",
    "docker":{"
      "image":"mongo:3.4.7",
      "network":"BRIDGE",
      "portMappings":[
        {
          "containerPort":27017,
          "hostPort":"<mongo_port>",
          "protocol":"tcp"
        }
      ],
      "parameters":[
        {
          "key":"restart",
          "value":"always"
        }
      ]
    }
  ],
  "instances":1,
  "cpus":0.1,
  "mem":512
}
```

sj-rest.json

Please, replace:

- <slave_advertise_ip> with a valid Mesos-slave IP;
- <zk_ip> and <zk_port> with the Zookeeper address;
- <rest_port> with the port for the SJ-rest service. It should be one of the available Mesos-slave ports.
- <mongo_port> with the port of MongoDB. Use the one you specified in mongo.json.

```
{
  "id":"sj-rest",
  "container":{"
    "type":"DOCKER",
    "docker":{"
      "image":"bwsj/sj-rest:dev",
      "network":"BRIDGE",
      "portMappings":[
        {
```



```

        "containerPort":8080,
        "hostPort":"<rest_port>",
        "protocol":"tcp"
    }
  ],
  "parameters":[
    {
      "key":"restart",
      "value":"always"
    }
  ]
},
"instances":1,
"cpus":0.1,
"mem":1024,
"env":{
  "MONGO_HOSTS":"<slave_advertise_ip>:<mongo_port>",
  "ZOOKEEPER_HOST":"<zk_ip>",
  "ZOOKEEPER_PORT":"<zk_port>"
}
}

```

For sj-rest.json it is better to upload the docker image separately:

```
sudo docker pull bwsj/sj-rest:dev
```

config.properties

This is a file with configurations for the tts service (used for T-streams).

Please, replace:

- <zk_ip> according to the Zookeeper address;
- <token> and <prefix-name> with valid token and prefix (description is provided in the [T-streams](#)). These token and prefix should be specified then in the T-streams service JSON (see below).

```

key=<token>
active.tokens.number=100
token.ttl=120

host=0.0.0.0
port=8080
thread.pool=4

path=/tmp
data.directory=transaction_data
metadata.directory=transaction_metadata
commit.log.directory=commit_log
commit.log.rocks.directory=commit_log_rocks

berkeley.read.thread.pool = 2

counter.path.file.id.gen=/server_counter/file_id_gen

auth.key=dummy
endpoints=127.0.0.1:31071
name=server

```

```
group=group

write.thread.pool=4
read.thread.pool=2
ttl.add-ms=50
create.if.missing=true
max.background.compactions=1
allow.os.buffer=true
compression=LZ4_COMPRESSION
use.fsync=true

zk.endpoints=<zk_ip>
zk.prefix=<prefix_name>
zk.session.timeout-ms=10000
zk.retry.delay-ms=500
zk.connection.timeout-ms=10000

max.metadata.package.size=100000000
max.data.package.size=100000000
transaction.cache.size=300

commit.log.write.sync.value = 1
commit.log.write.sync.policy = every-nth
incomplete.commit.log.read.policy = skip-log
commit.log.close.delay-ms = 200
commit.log.file.ttl-sec = 86400
stream.zookeeper.directory=/tts/tstreams

ordered.execution.pool.size=2
transaction-database.transaction-keep-time-min=70000
subscribers.update.period-ms=500
```

Specify the same token and prefix in the T-streams service JSON:

```
{
  "name": "tstream-ps-service",
  "description": "Example of T-streams service",
  "type": "service.t-streams",
  "provider": "zookeeper-ps-provider",
  "prefix": "<prefix-name>",
  "token": "<token>"
}
```

tts.json

This is a JSON file for T-streams. Please, replace:

- <path_to_conf_directory> with an appropriate path to the configuration file directory on your computer;
- <slave_advertise_ip> with the Mesos-slave IP;
- <tts_port> with the port for the tts service. It should be one of the available Mesos-slave ports.

```
{
  "id": "tts",
  "container": {
    "type": "DOCKER",
    "volumes": [
```

```

    {
      "containerPath": "/etc/conf/config.properties",
      "hostPath": "<path_to_conf_directory>",
      "mode": "RO"
    }
  ],
  "docker": {
    "image": "bws/tstreams-transaction-server",
    "network": "BRIDGE",
    "portMappings": [
      {
        "containerPort": 8080,
        "hostPort": "<tts_port>",
        "protocol": "tcp"
      }
    ],
    "parameters": [
      {
        "key": "restart",
        "value": "always"
      }
    ]
  }
},
"instances": 1,
"cpus": 0.1,
"mem": 512,
"env": {
  "HOST": "<slave_advertise_ip>",
  "PORT0": "<tts_port>"
}
}

```

3. Run the services on Marathon.

We will run the services via REST API. Send the provided requests.

Replace <marathon_address> with a valid Marathon address.

Mongo:

```
curl -X POST http://<marathon_address>/v2/apps -H "Content-type: application/json" -d @mongo.json
```

SJ-rest:

```
curl -X POST http://<marathon_address>/v2/apps -H "Content-type: application/json" -d @sj-rest.json
```

tts:

```
curl -X POST http://<marathon_address>/v2/apps -H "Content-type: application/json" -d @tts.json
```

Via the Marathon interface make sure the services are deployed.

Now look and make sure you have access to the Web UI. You will see the platform but it is not completed with any entities yet.

In the next section we will show you how to upload engines for your modules, configurations for engines and module validators.

Engine Uploading

Before uploading modules, upload the engine jars for them.

1. You should download the engine jars for each module type (input-streaming, regular-streaming, batch-streaming, output-streaming) and a Mesos framework:

```
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-input-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-regular-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-batch-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-output-streaming-engine.jar
wget http://c1-ftp1.netpoint-dc.com/sj/1.0-SNAPSHOT/sj-mesos-framework.jar
```

Now upload the engine jars into the platform. Please, replace `<slave_advertise_ip>` with the Mesos-slave IP and `<rest-port>` with the SJ-rest service port:

```
cd sj-platform
address=<slave_advertise_ip>:<rest-port>

curl --form jar=@sj-mesos-framework.jar http://$address/v1/custom/jars
curl --form jar=@sj-input-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-regular-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-batch-streaming-engine.jar http://$address/v1/custom/jars
curl --form jar=@sj-output-streaming-engine.jar http://$address/v1/custom/jars
```

When creating a module you should use correct name and version of the engine:

| Module type | Engine name | Engine version |
|-------------------|------------------------------------|----------------|
| Input-streaming | com.bws.w.input.streaming.engine | 1.0 |
| Regular-streaming | com.bws.w.regular.streaming.engine | 1.0 |
| Batch-streaming | com.bws.w.batch.streaming.engine | 1.0 |
| Output-streaming | com.bws.w.output.streaming.engine | 1.0 |

Specify them in the module specification JSON for engine-name and engine-version fields, for example:

```
{...
  "module-type": "regular-streaming",
  "engine-name": "com.bws.w.regular.streaming.engine",
  "engine-version": "1.0",
  ...}
```

2. Setup configurations for engines.

The range of configurations includes required and optional ones.

The list of all configurations can be viewed at the [Configuration](#) page.

To set up required configurations for the engines, run the following commands. Please, replace:

- `<slave_advertise_ip>` with the Mesos-slave IP;
- `<marathon_address>` with the address of Marathon;
- `<rest-port>` with the SJ-rest service port.

```
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
  ↪ "name": "session-timeout", "value": "7000", "domain": "configuration.apache-zookeeper"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
  ↪ "name": "current-framework", "value": "com.bws.w.fw-1.0", "domain": "configuration.system"}"
↪ "
```

```

curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"crud-rest-host\", \"value\": \"<slave_advertise_ip>\", \"domain\": \"configuration.system\
↪ \"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"crud-rest-port\", \"value\": \"<rest-port>\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"marathon-connect\", \"value\": \"http://<marathon_address>\", \"domain\": \"\
↪ configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"marathon-connect-timeout\", \"value\": \"60000\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"kafka-subscriber-timeout\", \"value\": \"100\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"low-watermark\", \"value\": \"100\", \"domain\": \"configuration.system\"}"

```

3. Send the next POST requests to upload configurations for module validators:

```

curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"input-streaming-validator-class\", \"value\": \"com.bwsj.crud.rest.instance.validator.\
↪ InputInstanceValidator\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"regular-streaming-validator-class\", \"value\": \"com.bwsj.crud.rest.instance.validator.\
↪ RegularInstanceValidator\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"batch-streaming-validator-class\", \"value\": \"com.bwsj.crud.rest.instance.validator.\
↪ BatchInstanceValidator\", \"domain\": \"configuration.system\"}"
curl --request POST "http://$address/v1/config/settings" -H 'Content-Type: application/json' --data "{\
↪ "name\": \"output-streaming-validator-class\", \"value\": \"com.bwsj.crud.rest.instance.validator.\
↪ OutputInstanceValidator\", \"domain\": \"configuration.system\"}"

```

4. You can add optional configurations if necessary. They have default values in the system but can be overridden. Find the full list of optional configurations at the [Optional configurations table](#).

Creating Platform Entities

Under this section you will find the information on platform entities creation.

We will not provide you with specific instructions as this part is custom and the set of platform entities you need for your tasks may differ. Step-by-step instructions on creating entities for example issue solutions are provided in the [Fping Example Task](#) and [Sflow Example Task](#) sections of Tutorial.

The following entities should be uploaded or created in the system:

1. Modules;
2. Providers;
3. Services;
4. Streams;
5. Instances.

Modules

You should create your own modules. Please, use instructions on module creation at [Custom Module Development Guide](#).

Then upload modules following the instruction in [Step 3. Module Uploading](#) of the Tutorial. Use REST API requests to upload each module (see [CRUD REST API for Modules](#)). Replace `<module_jar_name>` with the name of the module JAR file:

```
curl --form jar=@<module_jar_name>.jar http://$address/v1/modules
```

Or module uploading can be performed via the UI (see [Modules](#)).

Providers

Providers are a part of the streaming infrastructure. They can be created using REST API (replace `<provider_name>` with the name of the provider JSON file):

```
curl --request POST "http://$address/v1/providers" -H 'Content-Type: application/json' --data "@api-json/  
↪providers/<provider_name>.json"
```

For more details see [CRUD REST API for Providers](#).

Or providers can be created via the UI (see [Providers](#)).

Services

Services are a part of the streaming infrastructure. They can be created using REST API (replace `<service_name>` with the name of the service JSON file):

```
curl --request POST "http://$address/v1/services" -H 'Content-Type: application/json' --data "@api-json/  
↪services/<service_name>.json"
```

For more details see [CRUD REST API for Services](#).

Or services can be created via the UI (see [Services](#)).

Streams

Streams provide data exchange between modules. They can be created using REST API (replace `<stream_name>` with the name of the stream JSON file):

```
curl --request POST "http://$address/v1/streams" -H 'Content-Type: application/json' --data "@api-json/  
↪streams/<stream_name>.json"
```

For more details see [CRUD REST API for Streams](#).

Or streams can be created via the UI (see [Streams](#)).

Instances

Instances are used with engines to determine their collaborative work with modules. Each module needs an individual instance for it. Its type corresponds to the module type (input-streaming, regular-streaming or batch-streaming, output-streaming). Several instances with different settings can be created for one module to enable different processing scenarios.

Instances can be created using REST API (replace `<instance_name>` with the name of the instance JSON file):

```
curl --request POST "http://$address/v1/modules/input-streaming/pingstation-input/1.0/instance" -H 'Content-Type: application/json' --data "@api-json/instances/<instance_name>.json"
```

For more details see [CRUD REST API for Instances](#).

Or instances can be created via the UI (see [Instances](#)).

To start processing you should launch instances one by one. Use REST API (see [Start an instance](#)) or the Web UI (see [Instances](#)) to start processing and monitor the task execution.

3.9.2 Running Pre-built VirtualBox® Image

Another option to start working with SJ-Platform is to run a pre-built VirtualBox® image.

We suggest deploying the platform using Vagrant with VirtualBox® as a provider. This is the most rapid way to run the platform and assess its performance. It takes up to 30 minutes. The platform is deployed with all entities necessary to demonstrate the solution for the example task described in the [Fping Example Task](#) section.

Minimum system requirements are as follows:

- At least 8 GB of free RAM;
- VT-x enabled in BIOS;
- [Vagrant 1.9.1](#) installed;
- [VirtualBox 5.0.40](#) installed.

These requirements are provided for deployment on Ubuntu 16.04 OS.

To determine if CPU VT extensions are enabled in BIOS, do the following:

1. Install CPU-checker:

```
$ sudo apt-get update
$ sudo apt-get install cpu-checker
```

2. Then check:

```
$ kvm-ok
```

If the CPU is enabled, you will see:

```
INFO: /dev/kvm exists
KVM acceleration can be used
```

Otherwise, the response will look as presented below:

```
INFO: /dev/kvm does not exist
HINT: sudo modprobe kvm_intel
INFO: Your CPU supports KVM extensions
INFO: KVM (vmx) is disabled by your BIOS
HINT: Enter your BIOS setup and enable Virtualization Technology (VT),
and then hard poweroff/poweron your system
KVM acceleration can NOT be used
```

Prerequisites

1. At the first step install Vagrant and VirtualBox.

You can do it following the instructions in the official documentation:

- for [Vagrant](#)
- for [VirtualBox](#)

Please, make sure to install the service of the versions specified below:

- Vagrant 1.9.1
- VirtualBox 5.0.40
- Ubuntu 16.04

2. Then, clone the project repository from GitHub:

```
$ git clone https://github.com/bwsw/sj-demo-vagrant.git
$ cd sj-demo-vagrant
```

Launching Virtual Machine

To launch Vagrant use the following command:

```
$ vagrant up
```

It will take up to 30 minutes, 8GB memory and 7 CPUs.

Note: Please, make sure the ports are available! In Linux you can use netstat for monitoring network connections and statistics. Run the following command to list all open ports or currently running ports including TCP and UDP:

```
netstat -lntu
```

At the end of deploying you can see URLs of all services.

The detailed VM_Description is provided for you to understand the process of virtual machines' creation.

The platform is deployed with the entities: configurations, engines, providers, services, streams. Modules and instances are created as for the [Fping Example Task](#) described in Tutorial. To launch the data processing follow the instructions provided in the fping-Launch-Instances step of the example task.

Destroying Virtual Machine

To destroy the virtual machine(s) use:

```
$ vagrant destroy
```

Virtual machine(s) will be terminated.

In case, any problems occur during the deployment, please, open an issue in the project [GitHub repository](#) and let the project team solve it.

3.10 UI Guide

Contents

- UI Guide
 - Get Started
 - Configuration
 - Providers
 - Services
 - Streams
 - Modules
 - Custom Files
 - * Custom Jars
 - * Custom Files
 - Instances

3.10.1 Get Started

The Stream Juggler Platform has a user-friendly UI to create a processing sequence of arbitrary complexity, monitor it in action and manage it using pre-created modules with flexible functionality.

If you are a developer and you want to use the platform you need to know about some prerequisites for the platform. Make sure the following services are preliminarily deployed:

- Mesos
- Marathon
- Apache Zookeeper
- MongoDB
- Apache Kafka as an input source (optional)
- an external storage: Elasticsearch, SQL database or RESTful (optional).

Tip: Find more about SJ-Platform architecture at [SJ-Platform Architecture](#). Follow the [Platform Deployment](#) instructions to set up the services.

Once everything is ready, you can proceed to the Stream Juggler Platform.

SJ-Platform allows you to work with your own module, created in accordance with your requirements in order to solve your problem. How to create a module is described in detail in the [Custom Module Development Guide](#) section.

A module utilizes an instance/instances, i.e. a full range of settings for collaborative work of an engine and a module. See more information on a module structure at [Modules: Types, Structure, Pipeline](#).

The SJ-Platform allows you to upload your custom module for data stream processing with prerequisite engines and configuration settings. They are required to launch a module and define the processing in it.

They are provided by SJ-Platform and can be uploaded from the [Maven repository](#). Find more information about uploading configurations in the [Configuration](#) and the [Custom Files](#) sections of this document.

For correct module interaction, a stream/streams are required. The [Streams in SJ-Platform](#) section describes the streaming component of the system in detail.

Once you know what types of modules and instances you need to include into the pipeline, you can choose types of streams, providers and services which you need to create for them. Look at the diagram in the [Module Instances](#) section. It may help you to understand the dependency of entity types in the platform.

For example, if you want to create a regular module that will process Apache Kafka input data streams you have to create Apache Kafka service with Apache Kafka and Apache Zookeeper providers for it.

Below you will find the information on uploading your module via the UI and launching data processing.

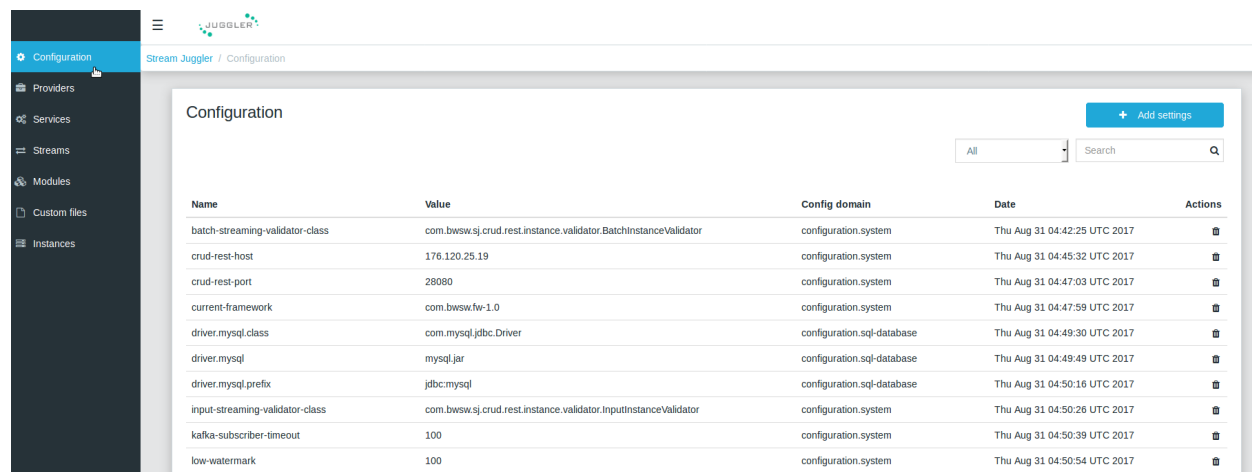
3.10.2 Configuration

So, the first step is to upload necessary configurations to the platform.

These are the basic settings required for the platform. In addition, here we will explain how to add configurations to the system.

Configurations are the settings necessary for the system work.

The configurations can be added under the Configuration tab of the main navigation bar.



The screenshot shows the 'Configuration' tab in the Stream Juggler UI. It features a table with columns: Name, Value, Config domain, Date, and Actions. There are 11 rows of configuration data. In the top right corner of the table area, there is a '+ Add settings' button and a search bar with a dropdown menu set to 'All'.

| Name | Value | Config domain | Date | Actions |
|---------------------------------|--|----------------------------|------------------------------|---------|
| batch-streaming-validator-class | com.bwsj.crud.rest.instance.validator.BatchInstanceValidator | configuration.system | Thu Aug 31 04:42:25 UTC 2017 | |
| crud-rest-host | 176.120.25.19 | configuration.system | Thu Aug 31 04:45:32 UTC 2017 | |
| crud-rest-port | 28080 | configuration.system | Thu Aug 31 04:47:03 UTC 2017 | |
| current-framework | com.bwsj.fw-1.0 | configuration.system | Thu Aug 31 04:47:59 UTC 2017 | |
| driver.mysql.class | com.mysql.jdbc.Driver | configuration.sql-database | Thu Aug 31 04:49:30 UTC 2017 | |
| driver.mysql | mysql.jar | configuration.sql-database | Thu Aug 31 04:49:49 UTC 2017 | |
| driver.mysql.prefix | jdbc:mysql | configuration.sql-database | Thu Aug 31 04:50:16 UTC 2017 | |
| input-streaming-validator-class | com.bwsj.crud.rest.instance.validator.InputInstanceValidator | configuration.system | Thu Aug 31 04:50:26 UTC 2017 | |
| kafka-subscriber-timeout | 100 | configuration.system | Thu Aug 31 04:50:39 UTC 2017 | |
| low-watermark | 100 | configuration.system | Thu Aug 31 04:50:54 UTC 2017 | |

Рис. 3.61: Figure 1.1: Configurations list

Please, click “Add Settings” in the upper-right corner above the list and fill in the form (the information on the required settings can be found in the [table](#) below):

1. Name * Enter a setting name here.
2. Value * Enter a setting value here.
3. Domain * Select a domain from the drop-down list.

Note: Required fields are marked with an asterisk (*).

Once the fields are correctly filled in, click the “Create” button and see the parameter appeared in the list of settings.

Click “Cancel” to drop all the specified settings. The configuration will not be created then.

The list of configurations created in the system can be viewed under the Configuration section of the main navigation bar.

It can be filtered by its type and/or a name using the search tool above the list.

Please, find the required configurations in the table below and make sure they are added to your platform so that your modules could work.

Таблица 3.1: Required configurations

| Config Domain | Name | Description | Example |
|---------------|--------------------------|---|---|
| system | crud-rest-host | REST interface host | localhost |
| system | crud-rest-port | REST interface port | 8080 |
| system | marathon-connect | Marathon address. Use to launch a framework which is responsible for running engine tasks and provides the information about applications that run on Mesos. Must begin with ‘http://’. | http://stream-juggler.z1.netpoint-dc.com:8080 |
| system | marathon-connect-timeout | Use when trying to connect by ‘marathon-connect’ (ms). | 60000 |
| system | current-framework | Indicates which file is used to run a framework. By this value, you can get a setting that contains a file name of framework jar. | com.bwsw.fw-0.1 |
| system | low-watermark | A number of preloaded messages for batch engine processing. | 1000 |
| kafka | subscriber-timeout | The period of time (ms) spent waiting in poll if data are not available. Must not be negative | 100 |
| zk | session.timeout | Use when connecting to Apache Zookeeper (ms). Usually when we are dealing with T-streams consumers/producers and Apache Kafka streams. | 3000 |

The range of optional settings is presented below. They have default values in the system but can be overridden by a user.

Таблица 3.2: Optional configurations

| Config Domain | Name | Description | Default value |
|---------------|------------------------------------|---|---------------|
| system | framework-principal | Framework principal for mesos authentication | — |
| system | framework-secret | Framework secret for mesos authentication | — |
| system | framework-backoff-seconds | Seconds for the first delay after crash | 7 |
| system | framework-backoff-factor | Factor for backoffSeconds parameter of following delays | 7.0 |
| system | framework-max-launch-delay-seconds | Max seconds for delay | 600 |
| system | output-processor-parallelism | A number of threads used to write data to an external data storage (Elasticsearch or RESTful) | 8 |

Note: In general ‘framework-backoff-seconds’, ‘framework-backoff-factor’ and ‘framework-max-launch-delay-

seconds’ configure exponential backoff behavior when launching potentially sick apps. This prevents sandboxes associated with consecutively failing tasks from filling up the hard disk on Mesos slaves. The backoff period is multiplied by the factor for each consecutive failure until it reaches `maxLaunchDelaySeconds`. This applies also to tasks that are killed due to failing too many health checks.

Configuration domain named ‘Apache Kafka’ contains properties used to create an Apache Kafka consumer (see [the official documentation](#)).

Note: You must not define properties such as ‘bootstrap.servers’, ‘enable.auto.commit’, ‘key.deserializer’ and ‘value.deserializer’ in order to avoid a system crash.

Configuration domain named ‘T-streams’ contains properties used for a T-streams consumer/producer.

Note: You must not define properties such as ‘producer.bind-host’, ‘producer.bind-port’, ‘consumer.subscriber.bind-host’ and ‘consumer.subscriber.bind-port’ to avoid a system crash.

To see the properties list check the following links: for a [producer](#) and for a [consumer](#) (you should use the textual constants to create a configuration).

For each uploaded custom jar a new configuration is added in the following format:

| |
|---|
| <code>key = {custom-jar-name}-{version}, value = {file-name}</code> |
|---|

3.10.3 Providers

Once all necessary configurations are added, a provider can be created.

A provider is a part of streaming infrastructure. This is an entity which contains general information to access a physical service (Apache Kafka, Apache Zookeeper, T-streams, Elasticsearch, SQL-database, RESTful).

Under the “Providers” section of the main navigation bar, you can see the list of providers, manage them, view the details.

Press “Create provider” and fill in the form where general fields and specific fields should be completed:

General fields:

- Type *

Select a type of the provider you are going to create from the drop-down list. The following options are available:

- Elasticsearch;
- Apache Zookeeper;
- Apache Kafka;
- RESTful;
- SQL database.

The type of the provider is determined by the type of the stream and the instance you want to create.

- Name * Enter a name of the provider here. It should be unique, must consist of digits, lowercase letters or hyphens and start with a letter.
- Description Enter a description for the provider here.

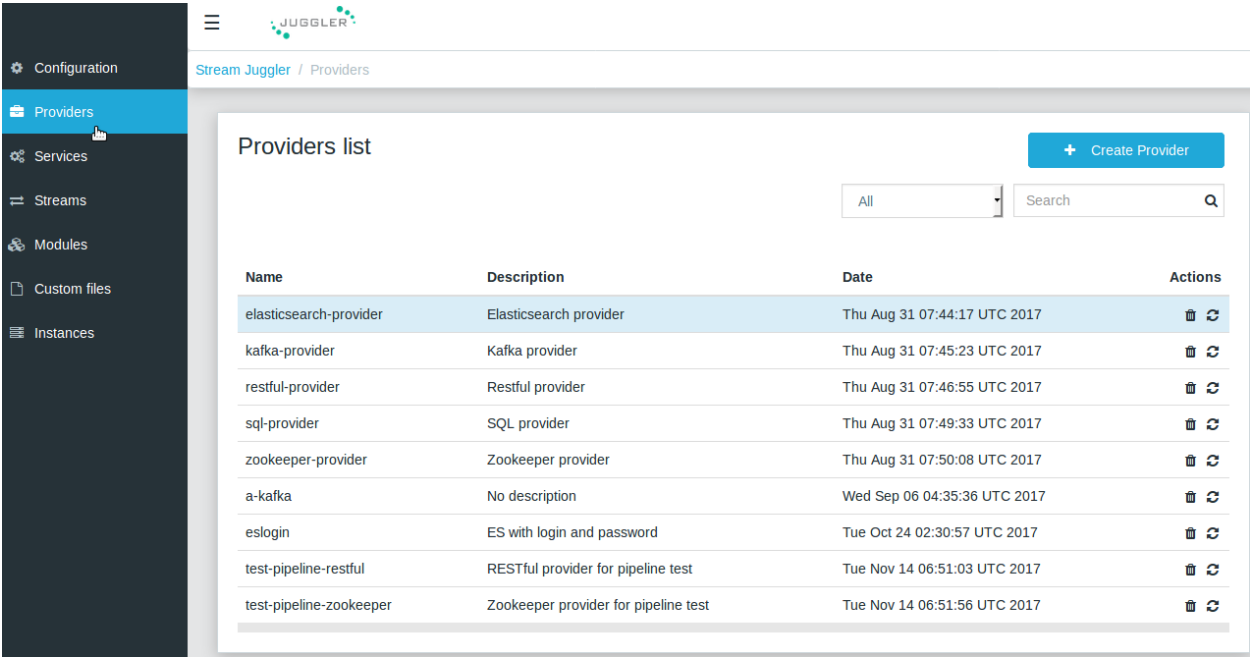


Рис. 3.62: Figure 1.2: Providers list

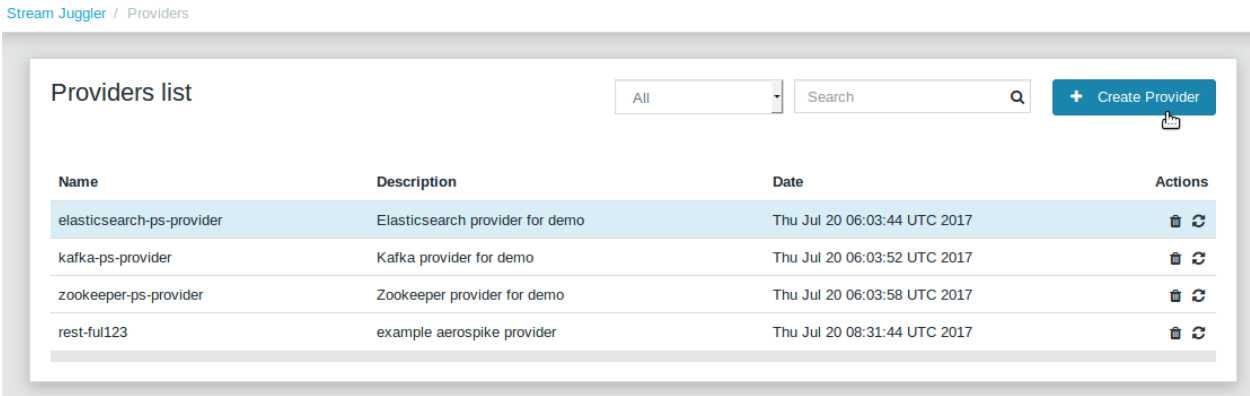


Рис. 3.63: Figure 1.3: “Create provider” button

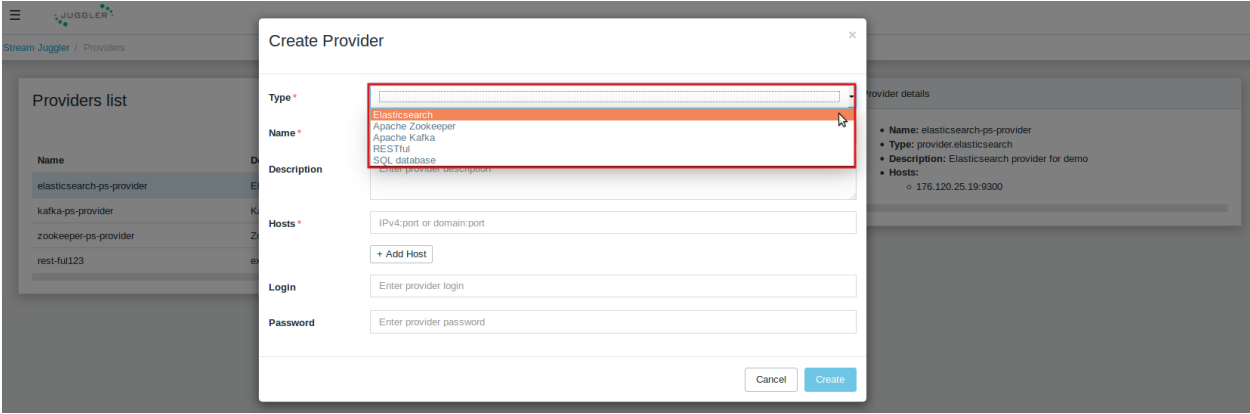


Рис. 3.64: Figure 1.4: Provider type field

- **Hosts *** Enter a provider host that is an endpoint of a physical service. Add more hosts by clicking the “Add Host” button and enter hostnames in the lines that appear.

Specific fields:

SQL database Provider Type

- **Login** Enter a provider login here if necessary
- **Password** Enter a password for the provider if necessary.
- **Driver *** Enter a provider driver name for the SQL-database provider type.

Elasticsearch Provider Type

- **Login** Enter a provider login if necessary.
- **Password** Enter a password for the provider if necessary.

Note: Required fields are marked with an asterisk (*)

Click “Create” below and see the provider appeared in the provider list. Provider details are displayed to the right when clicking a provider in the list.

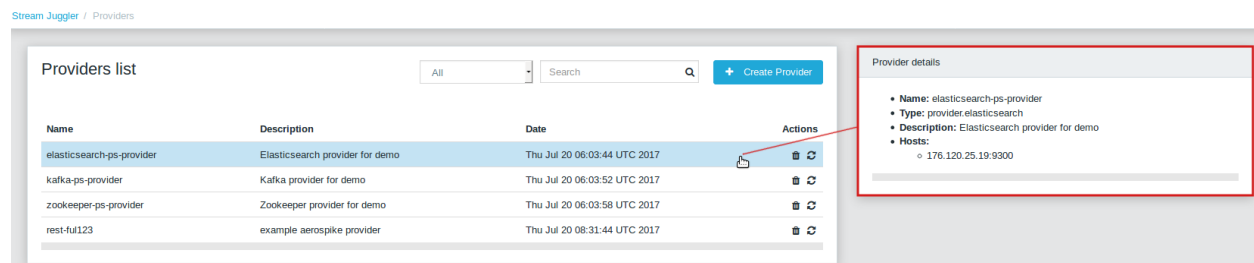


Рис. 3.65: Figure 1.5: Provider details

Click “Cancel” to drop provider creation.

You can perform the following actions on each provider in the list:

1. View provider’s name, date of creation, description.
2. Delete a provider clicking on the corresponding icon in the Action column for the provider you want to delete.

Note: A provider that is connected to a service cannot be deleted.

3. Test Connection to a provider.

The list of providers can be filtered by its type and/or a name using the search tool above the list.

3.10.4 Services

The next step is to create services. Services are a part of streaming infrastructure. This is an entity which contains specific information to access a physical service (Apache Kafka, Apache Zookeeper, T-streams, Elasticsearch, SQL-database, RESTful).

Under the Services section of the main navigation bar, you will find the list of services.

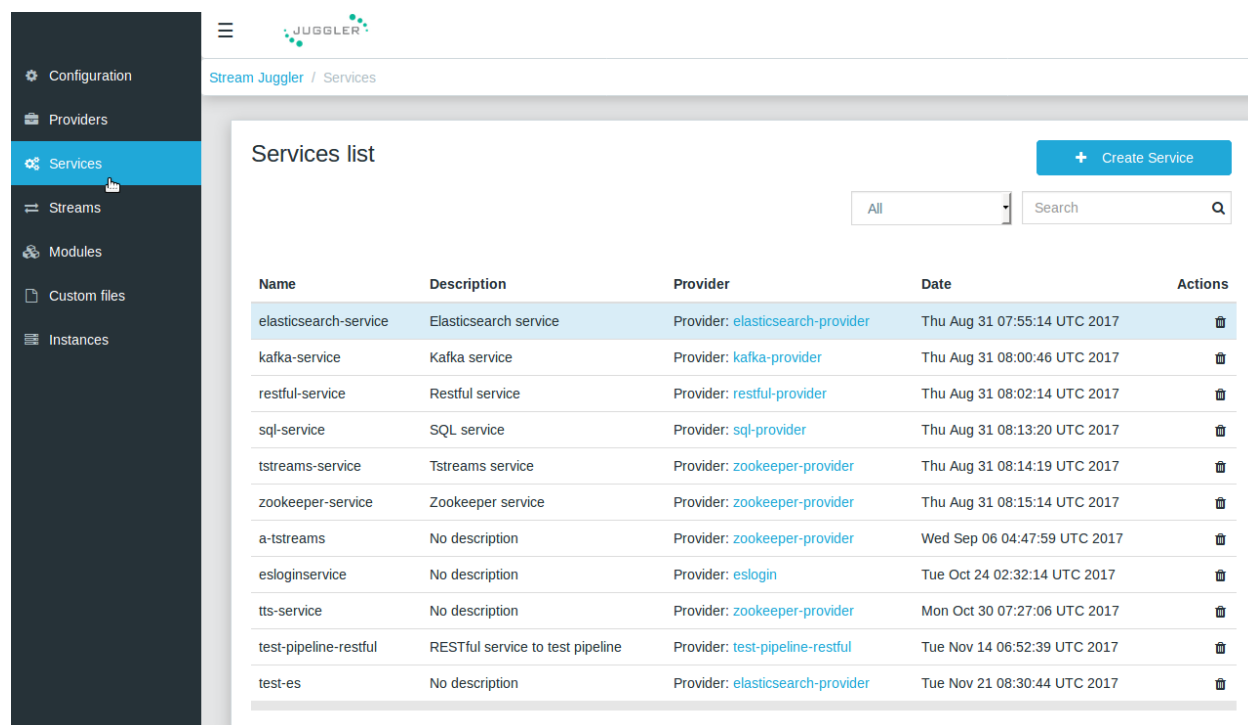


Рис. 3.66: Figure 1.6: Services list

Please, press “Create Service” and fill out the form with general and specific fields:

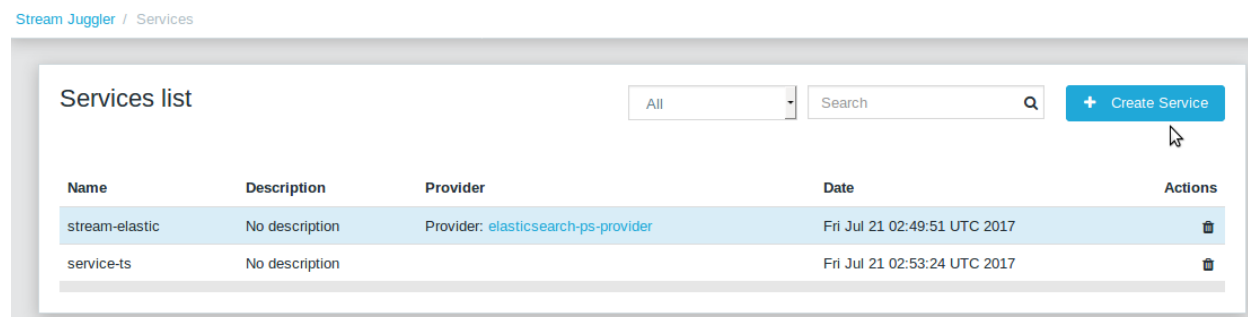


Рис. 3.67: Figure 1.7: “Create Service” button

General fields:

- Type * Select from the dropdown a type of the service:
- Elasticsearch
- SQL database
- T-streams
- Apache Kafka
- Apache Zookeeper
- RESTful

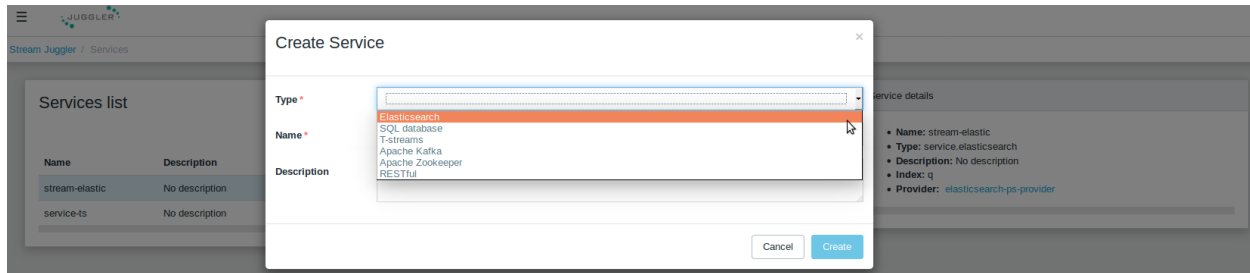


Рис. 3.68: Figure 1.8: Service type field

- **Name *** Enter a name of the services. It must consist of digits, lowercase letters or hyphens and start with a letter.
- **Description** Provide a description of the service here if necessary.
- **Provider *** This field appears once the service type is chosen.

Select a provider for the service here.

Providers available in the drop-down list are determined by the chosen service type.

Specific fields:

Apache Zookeeper Service Type

- **Namespace *** Please, specify a namespace here. It must consist of digits, lowercase letters or underscore and start with a letter.

Apache Kafka Service Type

- **ZK provider *** Please, select a zookeeper provider for the service here.

T-streams Service Type

- **Prefix *** Here a ZooKeeper path where metadata of transactions, streams are located should be specified.

Please, enter a prefix for the service here.

- **Token *** A token is a unique key for getting access to the service. It must contain no more than 32 symbols.

Please, enter a token for the service here.

Elasticsearch Service Type

- **Index *** Please, specify an index of the service here. It must consist of digits, lowercase letters or underscore and start with a letter.

SQL database Service Type

- **Database name *** Please, enter a database name for the service here.

RESTful Service Type

- **Http scheme *** Select the scheme of HTTP protocol from the drop-down list ("http" is set by default).
- **Http version** Select a http protocol version from the drop-down list ("1.1" is set by default).
- **Base path** Enter a path to the storage.
- **Headers** Enter extra HTTP headers. The values in the JSON format must be of a String type only.

Note: Required fields are marked with an asterisk (*)

Click “Create” below and you will see that the service appeared in the services list. Details of a service are displayed to the right when clicking the service in the list.

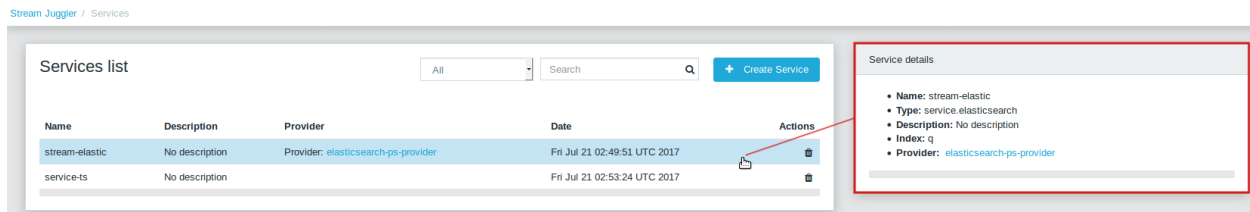


Рис. 3.69: Figure 1.9: Service details

Click “Cancel” to drop all the specified settings. The service will not be created then.

You can perform the following actions on each service in the list:

1. View service's name and description, the date of creation.
2. View a provider for the service and get the provider's information in a pop-up window by clicking on the active provider's name in the “Provider” column.

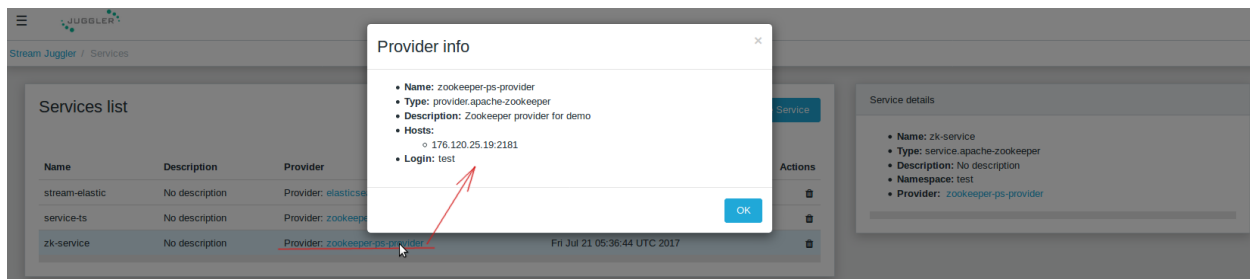


Рис. 3.70: Figure 1.10: Provider information

3. Delete a service clicking the corresponding button  in the Action column for the service you want to delete.

Note: A service used by one of the streams cannot be deleted.

The list of services can be filtered by its type and/or a name using the search tool above the list.

3.10.5 Streams

The next step is to create a data stream. A stream is a sequence of events that occur randomly at irregular intervals.

There are three kinds of streams in the SJ-Platform:

1. Input streams: These are streams which provide new events to the system. There are two input stream types in the SJ-Platform: TCP or Apache Kafka.

2. Internal streams: These are streams using which modules exchange data within the system. The only type of streams used for it is T-streams.
3. Output streams: These are streams which are a destination point for results. Three types of output streams are available for sending the processed data into different external storages: RESTful, SQL database and Elasticsearch.

Under the Streams section of the main navigation bar, you will find the list of streams.

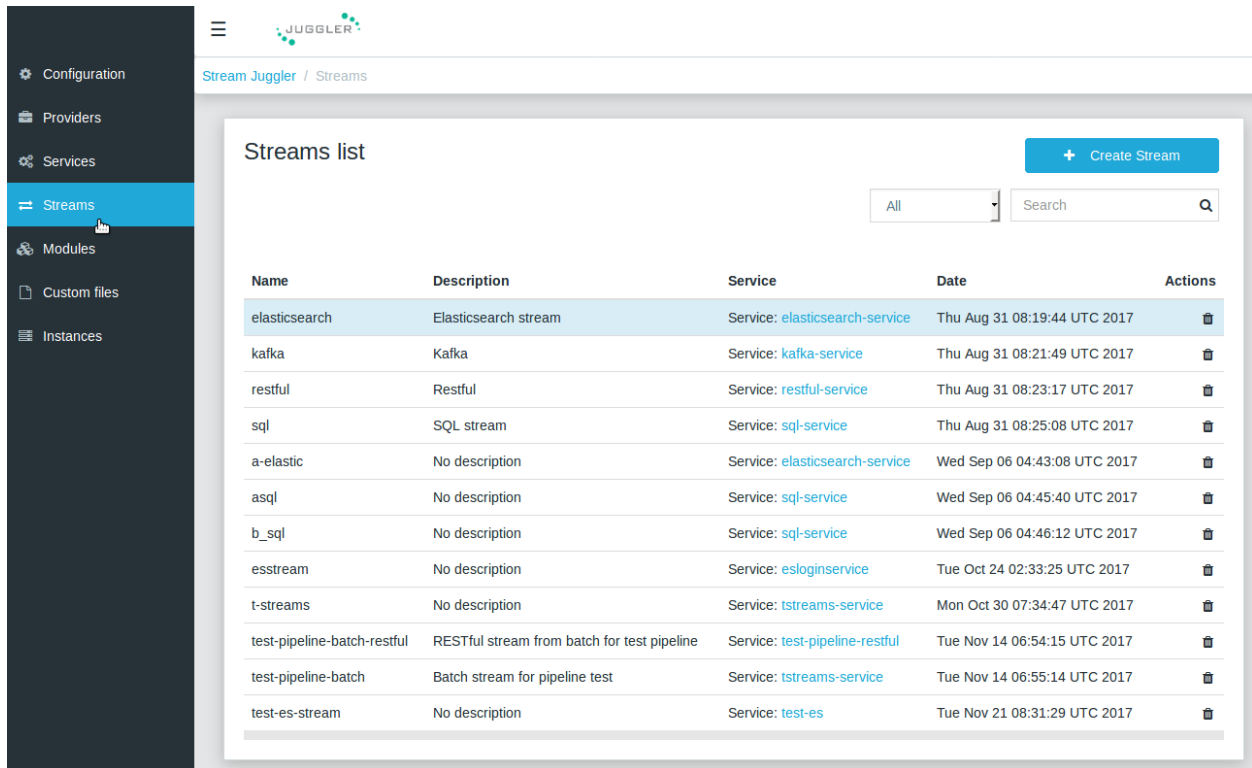


Рис. 3.71: Figure 1.11: Streams list

Please, press “Create Stream” and fill in the form where general and specific fields should be completed:

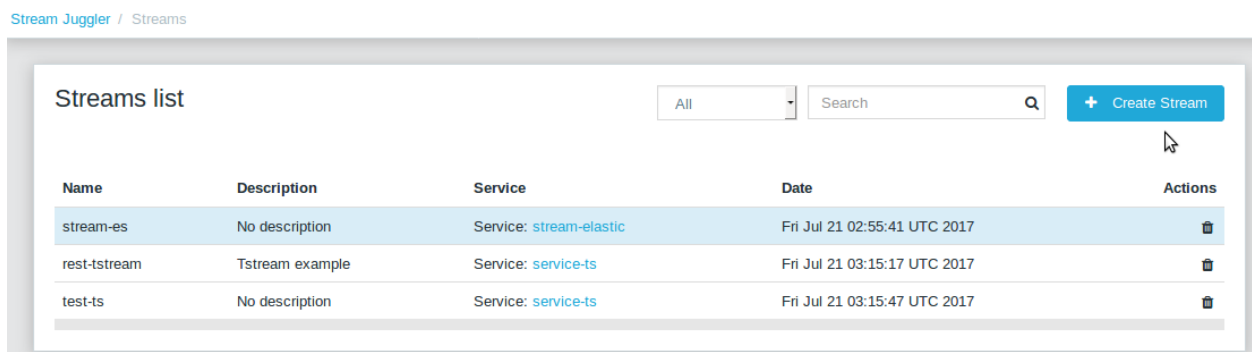


Рис. 3.72: Figure 1.12: “Create Stream” button

General fields:

- Type *

Select from a type of a stream the drop-down list:

- T-streams - It is an input stream of the T-stream type;
- Apache Kafka - It is an input stream of the Kafka type;
- SQL database - It is an output stream of the SQL database type;
- Elasticsearch - It is an output stream of the Elasticsearch type;
- RESTful - It is an output stream of the REST type.

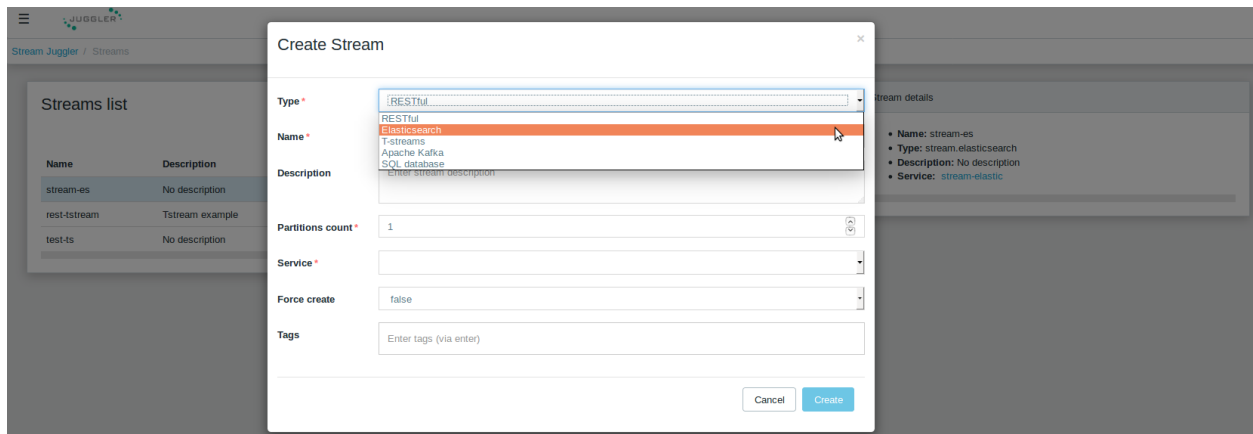


Рис. 3.73: Figure 1.13: Stream type field

- Name * Enter a stream name here. It must consist of lowercase letters, digits or hyphens only.
For the 'SQL database' stream type a name must consist of lowercase letters, digits or underscores, and start with a letter.
- Description Provide a description for the stream here if necessary.
- Service * Select a service from the drop-down list.

The range of available services is determined by a selected stream type.

Specific fields:

T-streams Stream Type

- Partitions count * A partition is a part of a data stream allocated for convenience in stream processing. A partition is a special conception which handles regular queues in multi-queues, e.g. a stream with one partition is a queue, but a stream with two partitions is like two different queues. Using streams with many partitions allows parallelizing the processing.

Enter a number of partitions. It must be a positive integer.

- Force create This field indicates if a stream should be removed and re-created by force (if it physically exists). Set it "True" or "False". The default value is "False".
- Tags Enter a tag/tags for the stream here.

Apache Kafka Stream Type

- Partitions count * A partition is a part of a data stream allocated for convenience in stream processing. A partition is a special conception which handles regular queues in multi-queues, e.g. a stream with one partition is a queue, but a stream with two partitions is

like two different queues. Using streams with many partitions allows handling parallelism properly as engine instances divide existing partitions fairly.

Enter a number of partitions. It must be a positive integer.

- Force create This field indicates if a stream should be removed and re-created by force (if it physically exists). Set it “True” or “False”. The default value is “False”.
- Tags Enter a tag/tags for the stream here.
- Replication Factor * **Replication factor** is the number of Zookeeper nodes to use.

Enter a replication factor here. It must be an integer.

SQL database Stream Type

- Partitions count * A partition is a part of a data stream allocated for convenience in stream processing. A partition is a special conception which handles regular queues in multi-queues, e.g. a stream with one partition is a queue, but a stream with two partitions is like two different queues. Using streams with many partitions allows handling parallelism properly as engine instances divide existing partitions fairly.

Enter a number of partitions. It must be a positive integer.

- Force create This field indicates if a stream should be removed and re-created by force (if it physically exists). Set it “True” or “False”. The default value is “False”.
- Tags Enter a tag/tags for the stream here.
- Primary Name of primary key field in the SQL database.

RESTful Stream Type

- Partitions count * A partition is a part of a data stream allocated for convenience in stream processing. A partition is a special conception which handles regular queues in multi-queues, e.g. a stream with one partition is a queue, but a stream with two partitions is like two different queues. Using streams with many partitions allows handling parallelism properly as engine instances divide existing partitions fairly.

Enter a number of partitions. It must be a positive integer.

- Force create This field indicates if a stream should be removed and re-created by force (if it physically exists). Set it “True” or “False”. The default value is “False”.
- Tags Enter a tag/tags for the stream here.

Elasticsearch Stream Type

- Force create This field indicates if a stream should be removed and re-created by force (if it physically exists). Set it “True” or “False”. The default value is “False”.
- Tags Enter a tag/tags for the stream here.

Note: Required fields are marked with an asterisk (*)

Click “Create” at the bottom and see the stream is in the list of streams now. Details of a stream are displayed to the right when clicking the stream in the list.

Click “Cancel” to drop all the specified settings. The stream will not be created then.

In the list of streams the following actions can be performed:

1. View stream’s name, description, date of creation.

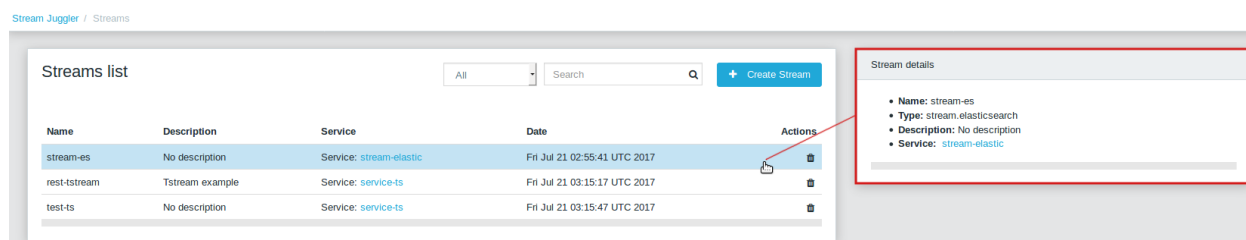


Рис. 3.74: Figure 1.14: Stream details

2. View a service for the stream and get the service's information in a pop-up window by clicking on the active service's name in the "Service" column.

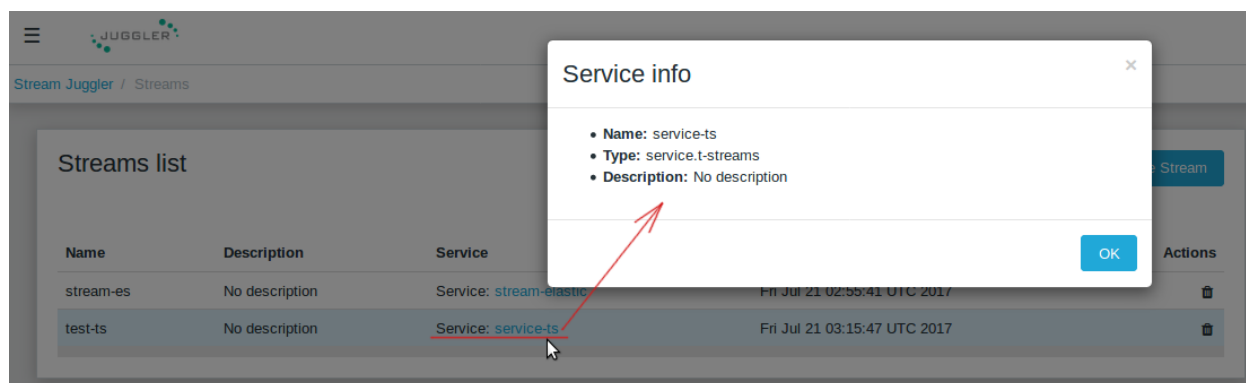



Рис. 3.75: Figure 1.15: Service information

3. Delete a stream by clicking the corresponding button  in the Actions column for the stream you want to delete.

Note: A stream used by any instance cannot be deleted.

The list of streams can be filtered by its type and/or a name using the search tool above the list.

3.10.6 Modules

In the next section — Modules — you can upload and manage your own module(s).

How to create a module is described in detail in the [Custom Module Development Guide](#) section.

The platform supports 4 types of modules:

1. Input-streaming
2. Regular-streaming (base type)
3. Batch-streaming
4. Output-streaming

Each of these types requires specific fields in its JSON file. Look at the [Json_schema](#) page to find the specification field description and examples of JSON for 4 module types.

Before uploading a module make sure an engine of the corresponding type is uploaded.

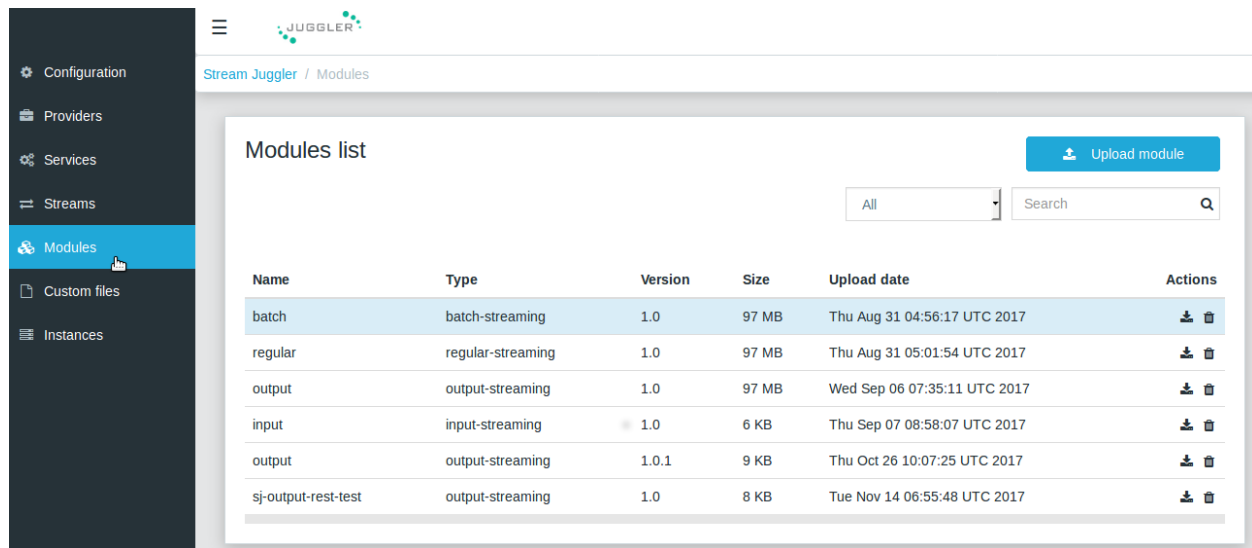


Рис. 3.76: Figure 1.16: Modules list

An engine is a basic platform component providing basic I/O functionality. It runs an application code and handles data from an input stream providing results to an output stream.

Currently, the following engine types are supported in the SJ-Platform:

1. TCP Input Engine It gets packages of data via TCP, handles them and produces series of events to T-streams. It can be used to program arbitrary TCP recognition.
2. Regular Processing Engine It gets events from Apache Kafka or T-streams and produces results to T-streams.
3. Batch Processing Engine It gets events from T-stream input streams, organizes them in batches and produces the results to T-stream output streams.
4. Output Engine
 - ElasticSearch Output Engine - allows creating output endpoint and save processed results to Elasticsearch.
 - SQL-database Output Engine - allows creating output endpoint and save processed results to MySQL, PostgreSQL, Oracle.

Engines should be uploaded as a JAR file under the [Custom Files](#) section in the “Custom Jars” tab.

After an engine is uploaded and the corresponding configurations appear under the “Configuration” tab, a module can be uploaded.

Note: Read more about necessary configurations in the [Configuration](#) section.

Click the “Upload Module” button and select a JAR file in the window to upload. Press “Open” and wait for a few seconds till the module is uploaded.

If the module is uploaded correctly a success message appears and the uploaded module is in the list of modules.

Otherwise, an error message will inform you that the module is not uploaded.

Module details are displayed to the right when clicking a module in the list.

Modules list

AllSearch

Upload module

Jar file 'ModuleRegular_5.1.1.Upload_module.jar' of module has been uploaded.

| Name | Type | Version | Size | Upload date | Actions |
|---------------------|-------------------|---------|------|------------------------------|---------|
| pingstation-process | regular-streaming | 1.0 | 5 MB | Thu Jul 20 01:24:53 UTC 2017 | |
| pingstation-output | output-streaming | 1.0 | 5 MB | Thu Jul 20 01:24:57 UTC 2017 | |

Рис. 3.77: Figure 1.17: Module is uploaded successfully

Modules list

AllSearch

Upload module



| Name | Type | Version | Size | Upload date | Actions |
|---------------------|-------------------|---------|-------|------------------------------|---------|
| batch | batch-streaming | 1.0 | 97 MB | Thu Aug 31 04:56:17 UTC 2017 | |
| regular | regular-streaming | 1.0 | 97 MB | Thu Aug 31 05:01:54 UTC 2017 | |
| output | output-streaming | 1.0 | 97 MB | Wed Sep 06 07:35:11 UTC 2017 | |
| input | input-streaming | 1.0 | 6 KB | Thu Sep 07 08:58:07 UTC 2017 | |
| output | output-streaming | 1.0.1 | 9 KB | Thu Oct 26 10:07:25 UTC 2017 | |
| sj-output-rest-test | output-streaming | 1.0 | 8 KB | Tue Nov 14 06:55:48 UTC 2017 | |

Module details

- Name: batch
- Description: Stub module by BW
- Version: 1.0
- Author:
- Licence: Apache 2.0
- Inputs:
 - Cardinality:
 - 1
 - 10
 - Types:
 - stream.apache-kafka
 - stream.t-streams
- Outputs:
 - Cardinality:
 - 1
 - 10
 - Types:
 - stream.t-streams

Рис. 3.78: Figure 1.18: Module details

In the list of modules the following actions can be performed:

1. View a module name, type, version and size, the date of uploading.
2. Download a module to your computer by clicking the download button  in the Actions column in the line for the module you want to download. You need only to specify a folder where to store the module to and click the “Save” button.
3. Delete a module by clicking the corresponding button  in the Actions column in the line for the module you want to delete.

Note: A module used by any instance cannot be deleted.

The list of modules can be filtered by its type and/or a name using the search tool above the list.

3.10.7 Custom Files

A Custom Files section is a section where a user can upload custom JAR files and other files that may be required for the module to function correctly.

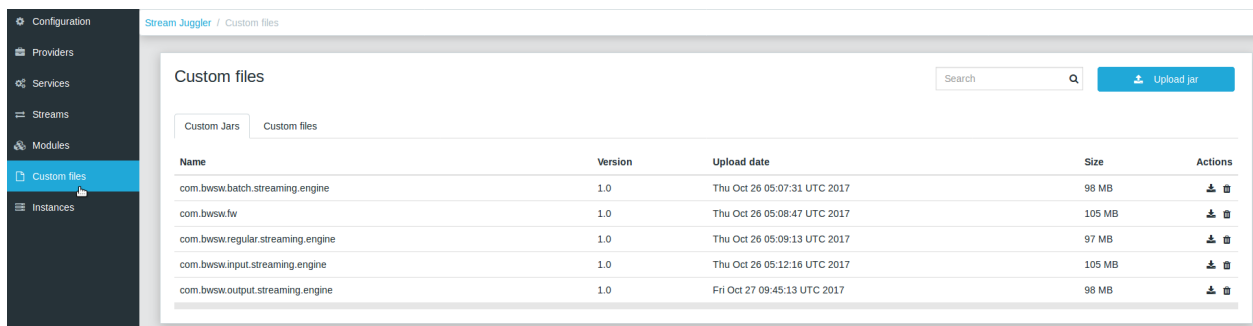




Рис. 3.79: Figure 1.19: Custom files list

Here you can find two tabs: Custom Jars and Custom files. Below you will find more information for each of these tabs.

Custom Jars

Under the “Custom Jars” tab the engine JAR files can be uploaded that are required for the module to function correctly. Click the “Upload Jar” button and select the JAR file to upload from your computer. Click “Open” in the modal window and wait for a few seconds before the JAR is uploaded. If it is uploaded successfully a success message appears above the file list and the uploaded JAR is added to the list of jars.

The following actions can be performed with the files in the list:



1. View a jar name, version and size, the date of uploading.
2. Download a jar file to your computer by clicking the download button  in the Actions column for the JAR file you want to download. You need only to specify a folder where to store the JAR and click the “Save” button.
3. Delete a jar by clicking the corresponding button  in the Actions column for the JAR file you want to delete.

The list of jars can be filtered by its name using the search tool above the list.

Custom Files

Under the “Custom files” tab any other files that are necessary for module work can be uploaded. Click the “Upload file” button and select the file to upload from your computer. Click “Open” in the modal window and wait for a few seconds before the file is uploaded. If it is uploaded successfully a success message appears above the file list and the uploaded file is added to the list of files.

The following actions can be performed with the files in the list:

1. View a file name, description, upload date and size
2. Download a file to your computer by clicking on the download icon  in the Actions column for the file you want to download. You need only to specify a folder where to store the file to and click the “Save” button.
3. Delete a file by clicking on the corresponding icon  in the Actions column for the file you want to delete.

The list of files can be filtered by its name using the search tool above the list.

3.10.8 Instances

Module’s engine uses a specific instance as a full set of settings that determine the collaborative work of an engine and a module.

Before creating an instance make sure all necessary configuration settings are added to the system.

Note: Read more about necessary configuration settings in the [Configuration](#) section.

Under the Instances section of the main navigation bar, there is a list of instances.

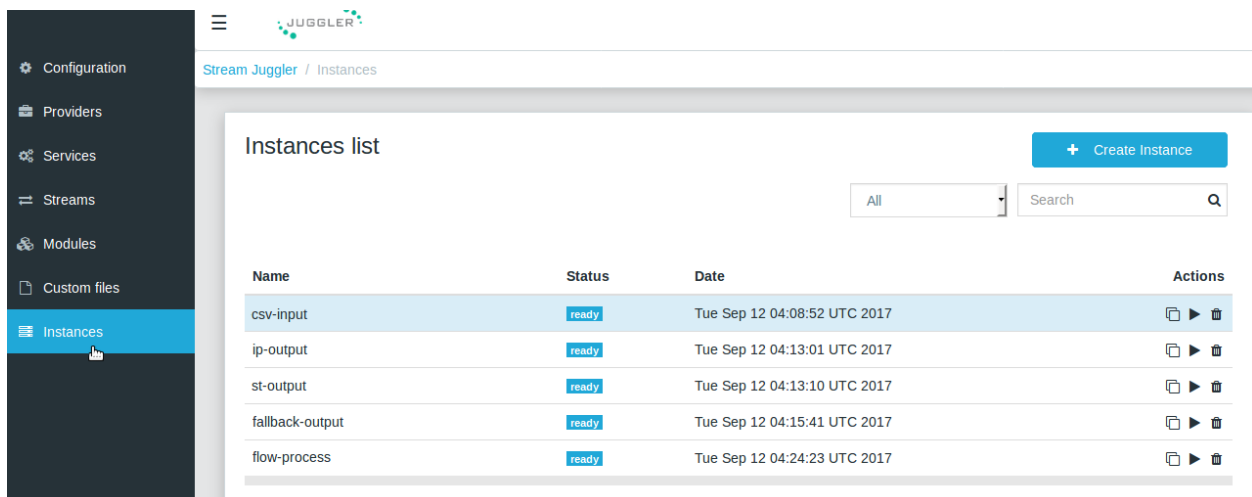


Рис. 3.80: Figure 1.20: Instance list

In the upper-right corner click “Create Instance” and choose the module from the drop-down list. An instance will be created for the selected module.

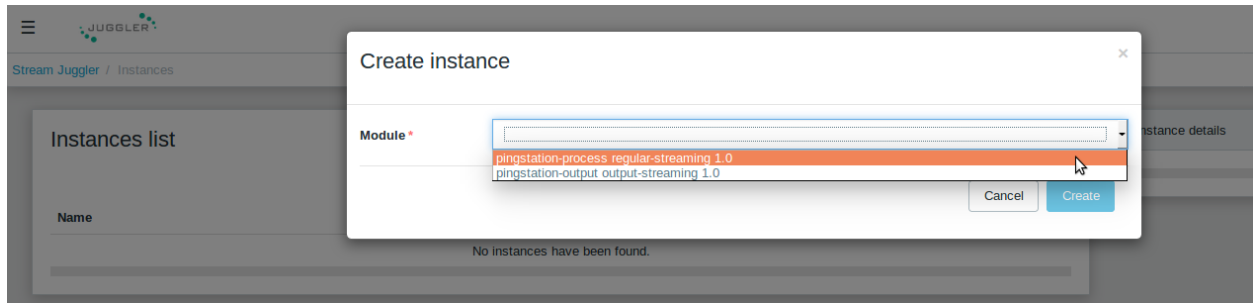


Рис. 3.81: Figure 1.21: “Create Instance” button

The type of module determines the type of instance that will be created: input-streaming, regular-streaming, batch-streaming or output-streaming.

Each type of instance requires specific settings to be filled in alongside with general settings common for all instances. These specific fields are to be determined in the instance parameters depending on each individual module type.

Please, review the lists with general and specific fields described below.

General fields

- **Name *** A unique name of an instance. Must consist of only letters, digits or hyphens, and start with a letter.
- **Description** Description of an instance.
- **Parallelism** This field determines the number of tasks that will process the data stream. To reduce the load and increase performance, the Parallelism parameter value should be larger than 1. Value may be integer or “max” string. If “max”, then parallelism equals to the minimum count of stream partitions (1 by default). For an input streaming instance, it should not exceed the total number of back-ups (Backup count + Async-backup-count)
- **Options** JSON with options for the module. It is validated by the implementation of the Streaming Validator method in the module. This field must contain valid values according to the [Streaming Validator](#).
- **Per-Task-Cores** The quantity of cores for a task (1 by default).
- **Per-Task-Ram** The amount of RAM for a task (1024 by default).
- **JVM Options** Json with jvm-options. It is important to emphasize that Mesos deletes a task if it uses more memory than it is specified in the perTaskRam parameter. There are no default options. In the example tasks in the [Tutorial](#) we use the following options for launching modules:

```
"jvmOptions" : { "-Xmx": "32m",
  "-XX:MaxDirectMemorySize=": "4m",
  "-XX:MaxMetaspaceSize=": "96m"
},
```

The values in the example fit the "perTaskRam": "192"

In general, the sum of the following parameters: Xmx, XX:MaxDirectMemorySize and XX:MaxMetaspaceSize should be less than Per-Task-Ram; XX:MaxMetaspaceSize must be greater than Xmx by 32m.

- **Node Attributes** JSON-map with [attributes](#) for the framework. While Mesos Master determines how many resources are offered to each framework, the frameworks' schedulers select which of the

offered resources to use. You can assign “+” or “-” to an attribute of Mesos resources. Resources with “+” attributes are added to the list of resources used for task launching. Resources with “-” attributes are not included into this list. In case the list of attributes is empty, the range of offered resources is determined by Mesos. Find more about Mesos resources in [the official documentation](#).

- **Coordination Service*** Name of the Apache Zookeeper service required for instance launching synchronization to avoid conflicts at Mesos resources usage.

Note: Select one and the same Apache Zookeeper service for all instances in your pipeline to avoid conflicts at Mesos resources usage.

- **Environment Variables** Variables used in the framework.
- **Performance Reporting Interval** Interval for creating a report of module performance metrics in ms (60000 by default).

Specific Fields

Input-streaming instance fields

- **Checkpoint Mode*** The value must be ‘time-interval’ for checkpointing after a set period of time or ‘every-nth’ for performing a checkpoint after a set number of events.
- **Checkpoint Interval*** The interval for performing the checkpoint. If Checkpoint Mode is ‘time-interval’ the value is set in ms. If Checkpoint Mode is ‘every-nth’ the value is the number of events after which the checkpoint is done.
- **Outputs*** Names of output streams (must be of the ‘T-streams’ type only). You can add several output streams by clicking “Add Output” and selecting an output stream name from the drop-down list.
- **Duplicate Check** The flag determines if an envelope (defined by an envelope key) has to be checked for duplication or not. False by default.
- **Lookup History*** How long a unique key of an envelope can stay in a queue for checking envelopes for duplication (in seconds). If it does not equal to 0, entries that are older than this time and not updated for this time are evicted automatically accordingly to an eviction-policy. Valid values are integers between 0 and Integer.MAX VALUE. Default value is 0, which means infinite.
- **Queue Max Size*** Maximum size of the queue that contains the unique keys of envelopes. When maximum size is reached, the queue is evicted on the basis of the policy defined at the default-eviction-policy.
- **Default Eviction Policy** If set, no items will be evicted and the “Queue Max Size” property will be ignored. You still can combine it with “Lookup History”. Can be ‘LRU’ (Least Recently Used) or ‘LFU’ (Least Frequently Used) or ‘NONE’ (NONE by default).
- **Eviction Policy** An eviction policy of input envelope duplicates. Can be ‘fix-time’ for storing an envelope key for the period specified in Lookup History, or ‘expanded-time’ meaning that if a duplicate envelope appears the time of the presence of the key will be updated (‘fix-time’ by default).
- **Backup Count** The number of backup copies you want to have (0 by default, maximum 6). Sync backup operations have a blocking cost which may lead to latency issues. You can skip this field if you do not want your entries to be backed up, e.g. if performance is more important than backing up.
- **Async-Backup-Count** The flag determines if an envelope (defined by an envelope key) has to be checked for duplication or not (0 by default). The backup operations are performed at some point in time (non-blocking operation).

Note: Backups increase memory usage since they are also kept in memory.

Regular-streaming instance fields

- Checkpoint Mode* Value must be ‘time-interval’ for checkpointing after a set period of time or ‘every-nth’ for performing a checkpoint after a set number of events.
- Checkpoint Interval* Interval for performing the checkpoint. If Checkpoint Mode is ‘time-interval’ the value is set in ms. If Checkpoint Mode is ‘every-nth’ the value is the number of events after which the checkpoint is done.
- Inputs* Names of input streams. Requires an input mode to be one of the following: ‘full’ (if you want each task to process all partitions of the stream) or ‘split’ (if you want to divide stream’s partitions among the tasks; it is a default value). The stream should exist in the system (it should be of ‘T-streams’ or ‘Apache Kafka’ type). You can add several input streams by clicking “Add Input” and selecting an input stream name from the drop-down list.

The screenshot shows the 'Create instance' dialog box. It has several sections:

- Module ***: A dropdown menu showing 'regular regular-streaming 1.0'.
- Name ***: A text input field containing 'regular-instance'.
- Description**: A text input field containing 'Instance for the regular-streaming module'.
- Inputs ***: A section with a dropdown menu showing 't-streams (tstreams-service, zookeeper-provider)' and a button 'full'. Below this is a red box highlighting an empty dropdown menu and a button 'x'.
- Outputs ***: A section with an empty dropdown menu and a button '+ Add Output'.
- Checkpoint mode ***: A dropdown menu.

 A red box highlights the 'Add Input' button and the input stream selection dropdown.

Рис. 3.82: Figure 1.22: Adding inputs when creating an instance

- Outputs* Names of output streams (should be of the ‘T-stream’ type only). You can add several output streams by clicking “Add Output” and selecting an output stream name from the drop-down list.
- Start From Value must be ‘newest’ (the system does not read the historical data, waits for new events), ‘oldest’ (the system reads all input stream events) or datetime (that requires specifying a timestamp and means the system reads events from the stream starting from the specified moment). If input streams of the instance are of Apache Kafka type, then ‘Start from’ must be ‘oldest’ or ‘newest’ (‘newest’ is default).
- State Management Allows managing stateful processing. Available values: ‘ram’ or ‘none’ (‘none’ is default). If ‘none’, no state is available. Selecting ‘ram’, you will save the state to the system memory.

- **State Full Checkpoint** The number of checkpoints after which the full checkpoint of state is performed (100 by default).
- **Event-Wait-Idle Time** Idle timeout, when no new messages appear (1000 ms is default).

Batch-streaming instance fields

- **Inputs*** Names of input streams. Requires an input mode to be one of the following: ‘full’ (if you want each task to process all partitions of the stream) or ‘split’ (if you want to divide stream’s partitions among the tasks; it is a default value). The stream should exist in the system (it should be of ‘T-streams’ or ‘Apache Kafka’ type). You can add several input streams by clicking “Add Input” and selecting an input stream name from the drop-down list.

The screenshot shows a 'Create instance' dialog box with the following fields and controls:

- Module ***: A dropdown menu showing 'batch batch-streaming 1.0'.
- Name ***: A text input field containing 'batch-instance'.
- Description**: A text input field containing 'Instance for the batch module'.
- Inputs ***: A dropdown menu showing 't-streams (tstreams-service, zookeeper-provider)' and a mode dropdown showing 'full'. Below this is a red-bordered box containing an empty dropdown menu and a mode dropdown with a close button (X).
- + Add Input**: A button with a hand cursor pointing to it.
- Outputs ***: A dropdown menu showing 't-streams (tstreams-service, zookeeper-provider)' and a '+ Add Output' button.
- Window**: A text input field containing '1'.

Рис. 3.83: Figure 1.23: Adding inputs when creating an instance

- **Outputs*** Names of output streams (must be of the ‘T-streams’ type only). You can add several input streams by clicking “Add Input” and selecting an input stream name from the drop-down list.
- **Window** Number of batches that will be contained in a window (1 by default). Must be greater than zero.
- **Sliding Interval** The interval at which a window will be shifted (count of batches that will be removed from the window after its processing). Must be greater than zero and less than or equal to the window (1 by default)
- **State Management** Allows managing stateful processing. Available values: ‘ram’ or ‘none’ (‘none’ is default). If ‘none’, no state is available. Selecting ‘ram’, you will save the state to the system memory.
- **State Full Checkpoint** The number of checkpoints after which the full checkpoint of state is performed (100 is default).
- **Start From Value** must be ‘newest’ (the system does not read the historical data, waits for new events), ‘oldest’ (the system reads all input stream events) or datetime (that requires specifying a timestamp and means the system reads events from the stream starting from the specified

moment). If input streams of the instance are of Apache Kafka type, then ‘Start from’ must be ‘oldest’ or ‘newest’ (‘newest’ is default).

- Event-Wait-Time Idle timeout, when there are no messages (1000 ms by default).

Output-streaming instance fields

- Checkpoint Mode* Value must be ‘time-interval’ for checkpointing after a set period of time or ‘every-nth’ for performing a checkpoint after a set number of events. For output streams ‘every-nth’ is only available.
- Checkpoint Interval* Interval for performing the checkpoint. If Checkpoint Mode is ‘time-interval’ the value is set in ms. If Checkpoint Mode is ‘every-nth’ the value is the number of events after which the checkpoint is done.
- Input* Name of an input stream. Must be of the ‘T-streams’ type only. Stream for this type of module has the ‘split’ mode only. The stream must exist in the system.
- Output* Name of an output stream (must be of ‘SQL-database’, ‘Elasticsearch’ or ‘RESTful’ type).
- Start From Value must be ‘newest’ (the system does not read the historical data, waits for new events), ‘oldest’ (the system reads all input stream events) or datetime (that requires specifying a timestamp and means the system reads events from the stream starting from the specified moment).

Note: Required fields are marked with an asterisk (*).

Click “Create” at the bottom and see the instance is in the list of instances now.

Click “Cancel” to drop all the specified settings. The instance will not be created then.

Instance Details

Details of an instance are displayed to the right when clicking the instance in the list.

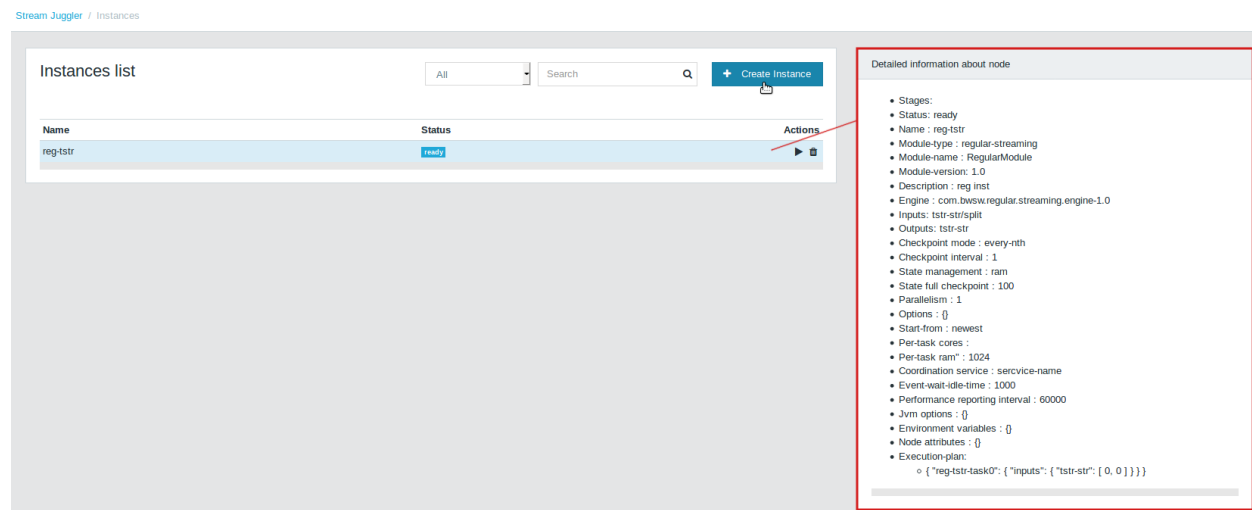


Рис. 3.84: Figure 1.24: Instance details


The system additionally shows the following fields in the Instance details panel:

- Stages Stages display information about the current status of the framework that starts an instance. It allows you to follow starting or stopping procedures of the instance.

The stages include:

- state - an instance status:
 - * ready
 - * starting
 - * started
 - * stopping
 - * stopped
 - * deleting
 - * failed
 - * error
- datetime - the last time the state has been changed.
- duration - how long a stage is in the current state. This field makes sense if a state field is ‘starting’, ‘stopping’ or ‘deleting’.
- Execution plan Execution plan consists of tasks. The number of tasks equals to a ‘Parallelism’ parameter. Each task has a unique name within the execution plan. Also, the task has a set of Input stream names and their intervals of partitions. In general, it provides the information of the sources from which the data will be consumed.
- Tasks For a started instance the task name and address (host and port) are specified in the Instance details panel.

In the list of instances the following actions can be performed:

1. Start an instance by clicking the “Start” button in the Actions column. The instance status will first change to “Starting” and in a few seconds to “Started”. That means the instance is launched and is working now.
2. Stop the instance that has been started i.e. has the “Started” status. Click the “Stop” button and wait for a while until the status changes to “Stopping” and then to “Stopped”.
3. Clone an instance. This function enables instance creation by copying the settings of an existing instance. Just click “Clone instance”  in the Actions column for the instance you want to clone.

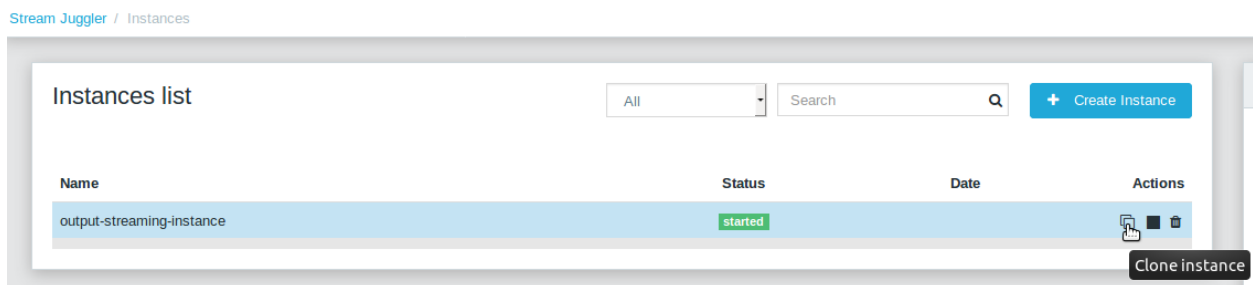




Рис. 3.85: Figure 1.25: “Clone instance” action

The form will show the settings of the selected instance. They can be edited and saved by clicking the “Create” button. A new instance will appear in the list of instances.

4. Delete an instance by clicking the corresponding button  in the Actions column for the instance you want to delete.

Note: An instance with statuses “Starting”, “Started”, “Stopping”, “Deleting” cannot be deleted.

5. View instance’s name and status. An instance may have the following statuses:
 - ready - a newly created instance and yet not started;
 - starting - a recently launched instance but yet not started (right after the “Start” button is pushed);
 - started - the launched instance that has started its work;
 - stopping - an instance that is being stopped;
 - stopped - an instance that has been stopped;
 - deleting - an instance in the process of deleting (right after the “Delete” button  is pressed);
 - failed - an instance that has been launched but not started due to some errors;
 - error - an error is detected when stopping the instance.

If an instance repeatedly gets ‘failed’ or ‘error’ statuses after pressing the “Start” button, you should follow the instructions:

1. Check that all of the following settings exist (see the [table](#) for more information on Configuration):
 - crud-rest-host (domain: system)
 - crud-rest-port (domain: system)
 - marathon-connect (domain: system)
 - current-framework (domain: system)
2. Check that the rest address specified in the ‘crud-rest-host’ and ‘crud-rest-port’ is available;
3. Check that the ‘marathon-connect’ is specified and the marathon address is available;
4. Check that there is a setting with name specified in the ‘current-framework’ and also a file with name and version (divide ‘current-framework’ by ‘-’) is uploaded.

If all described above is correct, but the “failed” or the “error” status still takes place, please contact the support team.

The information on the task execution are also available from the list of instances.

Click the “Information” button  next to the Instance name you want to get the information for.

A window will pop-up to show the information.

It displays the list of tasks with the following information for each task in the list:

- Task name.
- State - Task status.
- Directories - Directories of tasks of the instance. They are live references to the task change logs on Mesos.
- State change - The date of the last status change.
- Reason - The reason of the status change.

Stream Juggler / Instances

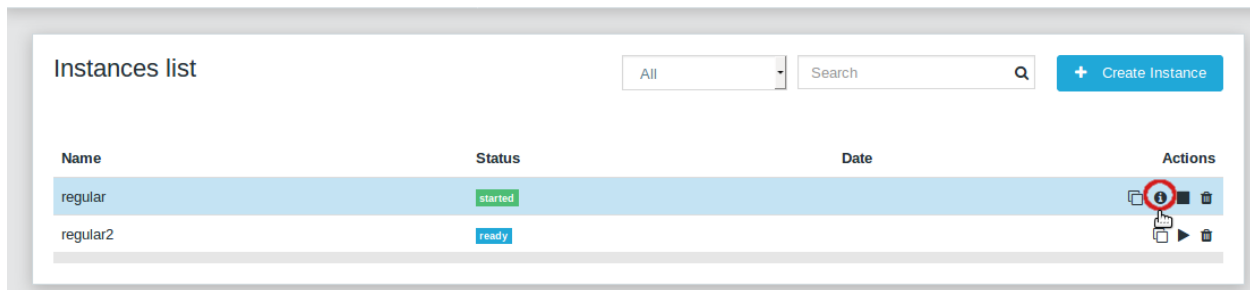


Рис. 3.86: Figure 1.26: Instance task information

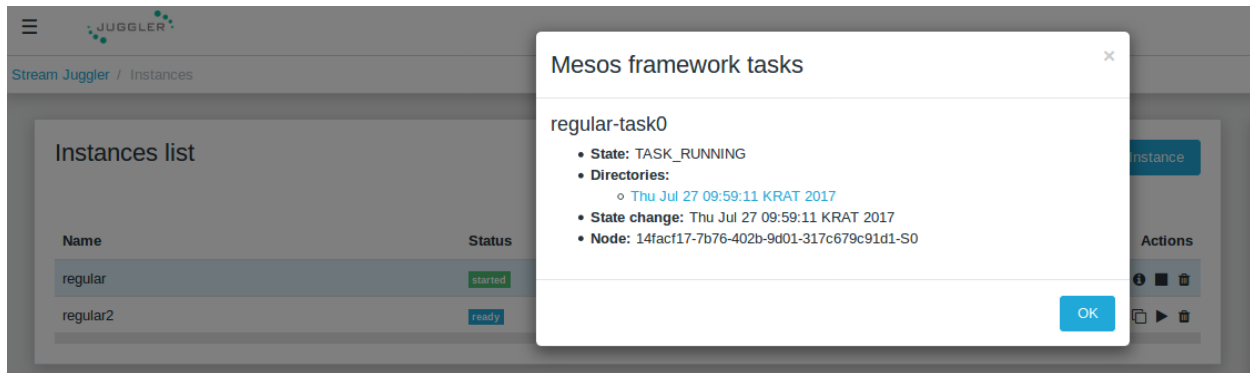


Рис. 3.87: Figure 1.27: Task information window

- Last node - Name of node that was used by a task before the status change (task failure).
- Node - Name of node used by the task.

These data are the information from the Mesos framework that starts a module. The information is displayed for started instances.

The list of instances can be filtered by its type and/or a name using the search tool above the list.

3.11 REST API Guide

Contents

- REST API Guide
 - Introduction
 - * HTTP Methods
 - * HTTP Status codes
 - * Requirements for requests and responses
 - CRUD REST API for Providers
 - * Create a new provider

- * Get provider by name
- * Get list of all providers
- * Get list of provider types
- * Delete provider by name
- * Test connection to provider
- * Get services related to a provider (by provider name)
- CRUD REST API for Services
 - * Service fields
 - Elasticsearch
 - Apache Kafka
 - T-streams
 - Apache Zookeeper
 - SQL database
 - RESTful
 - * Create a new service
 - * Get service by name
 - * Get list of all services
 - * Get list of service types
 - * Delete service by name
 - * Get streams and instances related to a service (by service name)
- CRUD REST API for Streams
 - * Stream fields
 - * Create a new stream
 - * Get list of all streams
 - * Get list of streams types
 - * Get stream by name
 - * Delete stream by name
 - * Get instances related to a stream (by stream name)
- CRUD REST API for Configurations
 - * Configuration fields
 - * Create a new configuration
 - * Get a configuration by name
 - * Get all configurations for specific config domain
 - * Delete a configuration by name
 - * Get all configurations

- * Get list of domains
- CRUD REST API for Custom Files
 - * Custom jars
 - Upload custom jar
 - Download a custom jar by file name
 - Download a custom jar by name and version
 - Delete a custom jar by file name
 - Delete a custom jar by name and version (from specification)
 - Get list of all uploaded custom jars
 - * Custom files
 - Upload a custom file
 - Download a custom file by file name
 - Delete a custom file
 - Get list of all uploaded custom files
- CRUD REST API for Modules
 - * Upload a module
 - * Download jar of uploaded module
 - * Delete uploaded module
 - * Get list of all uploaded modules
 - * Get list of types of modules
 - * Get list of all uploaded modules for such type
 - * Get specification for uploaded module
- CRUD REST API for Instances
 - * Create an instance of a module
 - Instance fields
 - General instance fields
 - Input-streaming instance fields
 - Regular-streaming instance fields
 - Batch-streaming instance fields
 - Output-streaming instance fields
 - Instance statuses
 - Execution plan
 - Stage
 - * Get all instances
 - * Get instances related to a specific module

- * Get all instances of a specific module
- * Get an instance of a specific module
- * Start an instance
- * Get the information about instance tasks
- * Stop an instance
- * Delete an instance of a specific module
- * Mesos Framework REST API

3.11.1 Introduction

The Stream Juggler platform provides developers with the opportunity of retrieving data by programming means. The range of REST API functions described on this page is intended for this purposes.

Each method request will return specific data. Choose the method you need from the list, and generate a request according to the method description below.

REST API provides access to resources (data entities) via URI paths. To use the REST API, the application will make an HTTP request and parse the response. The response format is JSON. The methods are standard HTTP methods: GET, POST and DELETE.

HTTP Methods

| Method | Description |
|--------|--|
| GET | Used for retrieving resources/resource instance. |
| POST | Used for creating resources and performing resource actions. |
| DELETE | Used for deleting resource instance. |

HTTP Status codes

Stream Jugler REST API uses HTTP status codes to indicate success or failure of an API call. In general, status codes in the 2xx range mean success, 4xx range mean there was an error in the provided information, and those in the 5xx range indicate server side errors.

Commonly used HTTP status codes are listed below.

| Status code | Description |
|-------------|--|
| 200 | OK |
| 201 | Created |
| 400 | Bad request |
| 404 | URL Not Found |
| 405 | Method Not Allowed (Method you have called is not supported for the invoked API) |
| 500 | Internal Error |

Requirements for requests and responses

Expected URI scheme for requests should include the version number of the REST API, for example:

```
http://{domain}/{version}/
```

All text data must be encoded in UTF-8.

The data format in the body of the response is JSON.

3.11.2 CRUD REST API for Providers

The range of REST API methods described below allows to create or delete a provider, get the information on the provider, get the list of providers in the system, test connection to a provider.

Таблица 3.3: Provider fields

| Field | Format | Description | Requirements |
|-------------|---------------|-------------------------|--|
| name* | String | Provider name. | Name must be unique and contain only letters, digits or hyphens. |
| description | String | Provider description. | |
| hosts* | Array[String] | List of provider hosts. | |
| login | String | Provider login. | For ‘provider.sql-database’, ‘provider.elasticsearch’ types. |
| password | String | Provider password. | For ‘provider.sql-database’, ‘provider.elasticsearch’ types. |
| type* | String | Provider type. | One of the following values are possible: ‘provider.elasticsearch’, ‘provider.apache-kafka’, ‘provider.apache-zookeeper’, ‘provider.sql-database’, ‘provider.restful’. |
| driver* | String | Driver name. | For ‘provider.sql-database’ provider type only. |

Important:

- Configurations must contain the following settings (<driver> is a value of the “driver” field) of sql database domain:
 - driver.<driver> - name of file with JDBC driver (must exists in files) (e.g. “mysql-connector-java-5.1.6.jar”)
 - driver.<driver>.class - name of class of this driver (e.g. “com.mysql.jdbc.Driver”)
 - driver.<driver>.prefix - prefix of server url: (prefix):(host:port)/(database), one of [jdbc:mysql, jdbc:postgresql, jdbc:oracle:thin]
-

Note: * - a required field.

Create a new provider

Request method: POST

Request format:

```
/v1/providers
```

Таблица 3.4: Response

| Status code | Description |
|-------------|---|
| 201 | Provider <provider name> has been created. |
| 400 | Cannot create provider. Errors: <list-of-errors>. |
| 500 | Internal server error. |

Request json example:

```
{
  "name": "kafka-provider",
  "description": "example of kafka provider",
  "type": "kafka",
  "hosts": [
    "192.168.1.133:9092",
    "192.168.1.135:9092"
  ]
}
```

Success response example:

```
{
  "status-code": 201,
  "entity": {
    "message": "Provider 'kafka-provider' has been created."
  }
}
```

Error response example:

```
{
  "status-code": 400,
  "entity": {
    "message": "Cannot create provider. Errors: <creation_errors_string>."
  }
}
```

Get provider by name

Request method: GET

Request format:

```
/v1/providers/{name}
```

Таблица 3.5: Response

| Status code | Description |
|-------------|--|
| 200 | Provider. |
| 404 | Provider <provider name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "provider": {
      "name": "kafka-example",
      "description": "example kafka provider",
      "type": "provider.apache-kafka",
      "hosts": [
        "192.168.1.133:9092",
        "192.168.1.135:9092"
      ],
      "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
    }
  }
}
```

Error response example:

```
{
  "status-code": 404,
  "entity": {
    "message": "Provider 'kafka-provider' has not been found."
  }
}
```

Get list of all providers

Request method: GET

Request format:

```
/v1/providers
```

Таблица 3.6: Response

| Status code | Description |
|-------------|------------------------|
| 200 | List of providers. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "providers": [
      {
        "name": "kafka-provider",
        "description": "example kafka provider",
        "type": "provider.apache-kafka",
        "hosts": [
          "192.168.1.133:9092",
          "192.168.1.135:9092"
        ],
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
      }
    ]
  }
}
```

```
{
  "name": "es-provider",
  "description": "elasticsearch provider example",
  "login": "my_login",
  "password": "my_pass",
  "type": "provider.elasticsearch",
  "hosts": [
    "192.168.1.133"
  ],
  "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
}
]
```

Get list of provider types

Request method: GET

Request format:

```
/v1/providers/_types
```

Таблица 3.7: Response

| Status code | Description |
|-------------|------------------------|
| 200 | List of types. |
| 500 | Internal server error. |

Success response example:

```
{
  entity: {
    types: [
      {
        id: "provider.elasticsearch",
        name: "Elasticsearch"
      },
      {
        id: "provider.apache-zookeeper",
        name: "Apache Zookeeper"
      },
      {
        id: "provider.apache-kafka",
        name: "Apache Kafka"
      },
      {
        id: "provider.restful",
        name: "RESTful"
      },
      {
        id: "provider.sql-database",
        name: "SQL database"
      }
    ]
  }
},
```



```
status-code: 200
}
```

Delete provider by name

Request method: DELETE

Request format:

```
/v1/providers/{name}
```

Таблица 3.8: Response

| Status code | Description |
|-------------|---|
| 200 | Provider |
| 404 | Provider <provider name> has not been found. |
| 422 | Cannot delete provider <provider name>. Provider is used in services. |
| 500 | Internal server error |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Provider 'kafka-provider' has been deleted."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Cannot delete provider 'provider-name'. Provider is used in services."
  },
  "status-code": 422
}
```

Test connection to provider

Method: GET

Request format:

```
/v1/providers/{name}/connection
```

Таблица 3.9: Response

| Status code | Description |
|-------------|--|
| 200 | Provider. |
| 404 | Provider <provider name> has not been found. |
| 409 | Provider is not available. |
| 500 | Internal server error. |

Success response example (provider is available):

```
{
  "status-code": 200,
  "entity": {
    "connection": true
  }
}
```

Error response example (provider is not available):

```
{
  "entity": {
    "connection": false,
    "errors": "Can not establish connection to Kafka on '192.168.1.133:9092'"
  },
  "statusCode": 409
}
```

Get services related to a provider (by provider name)

Request method: GET

Request format:

```
/v1/providers/{name}/related
```

Таблица 3.10: Response

| Status code | Description |
|-------------|--|
| 200 | List of services. |
| 404 | Provider <provider name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "entity": {
    "services": [
      "abc",
      "def"
    ]
  },
  "statusCode": 200
}
```

Error response example:

```
{
  "entity": {
    "message": "Provider 'kafka-provider' has not been found."
  },
  "status-code": 404
}
```

Tip: A full range of error responses can be found at `Provider_Errors`

3.11.3 CRUD REST API for Services

The range of REST API methods described below allows to create or delete a service, get the information on the service, get the list of services and service types in the system, get streams and instances related to a service.

Service fields

Each particular service has its own set of fields.

Таблица 3.11: Service type names for request.

| Service Types |
|--------------------------|
| service.elasticsearch |
| service.apache-kafka |
| service.t-streams |
| service.apache-zookeeper |
| service.sql-database |
| service.restful |

Elasticsearch

| Field | Format | Description | Requirements |
|--------------|--------|---|---|
| type* | String | Service type. | |
| name* | String | Service name. | Must be unique and contain only letters, digits or hyphens. |
| description | String | Service description. | |
| index* | String | Elasticsearch index. | |
| provider* | String | Provider name. | Provider can be of 'provider.elasticsearch' type only. |
| creationDate | String | The time when a service has been created. | |

Apache Kafka

| Field | Format | Description | Requirements |
|--------------|--------|---|---|
| type* | String | Service type | |
| name* | String | Service name | Must be unique and contain only letters, digits or hyphens. |
| description | String | Service description | |
| provider* | String | Provider name. | Provider can be of 'provider.apache-kafka' type only. |
| zkProvider* | String | zk provider name. | zkProvider can be of 'provider.apache-zookeeper' type only. |
| creationDate | String | The time when a service has been created. | |

T-streams

| Field | Format | Description | Requirements |
|--------------|--------|---|---|
| type* | String | Service type. | |
| name* | String | Service name. | Must be unique and contain only letters, digits or hyphens. |
| description | String | Service description. | |
| provider* | String | Provider name. | Provider can be of 'provider.apache-zookeeper' type only. |
| prefix* | String | A znode path | Must be a valid znode path. |
| token* | String | A token | Should not contain more than 32 symbols. |
| creationDate | String | The time when a service has been created. | |

Apache Zookeeper

| Field | Format | Description | Requirements |
|--------------|--------|---|---|
| type* | String | Service type. | |
| name* | String | Service name. | Must be unique and contain only letters, digits or hyphens. |
| description | String | Service description. | |
| namespace* | String | Zookeeper namespace. | |
| provider* | String | Provider name. | Provider can be of 'provide.apache-zookeeper' type only. |
| creationDate | String | The time when a service has been created. | |

SQL database

| Field | Format | Description | Requirements |
|--------------|--------|---|---|
| type* | String | Service type. | |
| name* | String | Service name. | Must be unique and contain only letters, digits or hyphens. |
| description | String | Service description. | |
| provider* | String | Provider name. | Provider can be of 'JDBC' type only. |
| database* | String | Database name. | |
| creationDate | String | The time when a service has been created. | |

RESTful

| Field | Format | Description | Requirements |
|--------------|--------|---|--|
| type* | String | Service type. | |
| name* | String | Service name. | Must be unique and contain only letters, digits or hyphens. |
| description | String | Service description. | |
| provider* | String | Provider name. | Provider can be of 'provider.restful' type only. |
| basePath | String | Path to storage (/ by default) | |
| httpScheme | String | The time when a service has been created. | Scheme of HTTP protocol, one of ('http', 'https'); ('http' by default) |
| httpVersion | String | Version of HTTP protocol | One of (1.0, 1.1, 2); (1.1 by default) |
| headers | Object | Extra HTTP headers. | Values in object must be only String type. ({} by default) |
| creationDate | String | The time when a service has been created. | |

Note: * - required fields.

Create a new service

Request method: POST

Request format:

```
/v1/services
```

Таблица 3.12: Response

| Status code | Description |
|-------------|--|
| 201 | Service <service name> has been created. |
| 400 | Cannot create service. Errors: <list-of-errors>. |
| 500 | Internal server error. |

Request json example:

```
{
  "name": "test-rest-zk-service",
  "description": "ZK test service created with REST",
  "type": "service.apache-zookeeper",
  "provider": "zk-prov",
  "namespace": "namespace"
}
```

Success response example:

```
{
  "status-code": 201,
  "entity": {
    "message": "Service 'test-rest-zk-service' has been created."
  }
}
```

Error response example:

```
{
  "status-code": 400,
  "entity": {
    "message": "Cannot create service. Errors: <creation_errors_string>."
  }
}
```

Get service by name

Request method: GET

Request format:

```
/v1/services/{name}
```

Таблица 3.13: Response

| Status code | Description |
|-------------|--|
| 200 | Service. |
| 404 | Service <service name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
```

```

"service": {
  "name": "test-rest-zk-service",
  "description": "ZK test service created with REST",
  "type": "service.apache-zookeeper",
  "provider": "zk-prov",
  "namespace": "namespace"
}
}
}

```

Error response example:

```

{
  "status-code": 404,
  "entity": {
    "message": "Service <service name> has not been found."
  }
}

```

Get list of all services

Request method: GET

Request format:

```
/v1/services
```

Таблица 3.14: Response

| Status code | Description |
|-------------|------------------------|
| 200 | List of services. |
| 500 | Internal server error. |

Success response example:

```

{
  "status-code": 200,
  "entity": {
    "services": [
      {
        "name": "test-rest-zk-service",
        "description": "ZK test service created with REST",
        "type": "service.apache-zookeeper",
        "provider": "zk-prov",
        "namespace": "namespace"
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
      },
      {
        "name": "rest-service",
        "description": "rest test service",
        "namespace": "mynamespace",
        "provider": "rest-prov",
        "type": "service.restful"
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
      }
    ]
  }
}

```

```
]
}
}
```

Get list of service types

Request method: GET

Request format:

```
/v1/services/_types
```

Таблица 3.15: Response

| Status code | Description |
|-------------|------------------------|
| 200 | List of types. |
| 500 | Internal server error. |

Success response example:

```
{
  entity: {
    types: [
      {
        id: "service.elasticsearch",
        name: "Elasticsearch"
      },
      {
        id: "service.sql-database",
        name: "SQL database"
      },
      {
        id: "service.t-streams",
        name: "T-streams"
      },
      {
        id: "service.apache-kafka",
        name: "Apache Kafka"
      },
      {
        id: "service.apache-zookeeper",
        name: "Apache Zookeeper"
      },
      {
        id: "service.restful",
        name: "RESTful"
      }
    ]
  },
  status-code: 200
}
```

Delete service by name

Request method: DELETE

Request format:

```
/v1/services/{name}
```

Таблица 3.16: Response

| Status code | Description |
|-------------|---|
| 200 | Service. |
| 404 | Service <service name> has not been found. |
| 422 | Cannot delete service <service name>. Service is used in streams. |
| 422 | Cannot delete service <service name>. Service is used in instances. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Service 'abc' has been deleted."
  }
}
```

Error response example:

```
{
  "status-code": 404,
  "entity": {
    "message": "Service <service name> has not been found."
  }
}
```

Get streams and instances related to a service (by service name)

Request method: GET

Request format:

```
/v1/services/{name}/related
```

Таблица 3.17: Response

| Status code | Description |
|-------------|--|
| 200 | List of streams and instances. |
| 404 | Service <service name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "entity": {
    "streams": [
      "new-tstr"
    ],
    "instances": [
      "new",

```

```
"test",
"input1",
"input2",
"input3",
"output",
"regular",
"demo-regular",
"rew",
"input",
"neew"
]
},
"statusCode": 200
}
```

Error response example:

```
{
  "status-code": 404,
  "entity": {
    "message": "Service <service name> has not been found."
  }
}
```

Tip: A full range of error responses can be found at `Services_Errors`

3.11.4 CRUD REST API for Streams

The range of REST API methods described below allows to create or delete a stream, get the information on the stream, get the list of streams and stream types in the system, get instances related to a stream.

Stream fields

Таблица 3.18: Stream types for requests

| Types |
|----------------------|
| stream.elasticsearch |
| stream.apache-kafka |
| stream.t-streams |
| stream.sql-database |
| stream.restful |

Таблица 3.19: Response

| Field | Format | Description | Requirements |
|--------------------|---------------|---|---|
| name* | String | Stream name. | Must be unique and contain only lowercase letters, digits or hyphens. |
| description | String | Stream description | |
| service* | String | Service id | |
| type* | String | Stream type | One of the following values : stream.t-streams, stream.apache-kafka, stream.sql-database, stream.elasticsearch, stream.restful. |
| tags | Array[String] | Tags. | |
| partitions* | Int | Partitions. | For stream.t-streams, stream.apache-kafka types |
| replicationFactor* | Int | Replication factor (how many zookeeper nodes to utilize). | For stream.apache-kafka stream type only. |
| primary | String | Primary key field name used in sql database. | For stream.sql-database stream type only. |
| force | Boolean | Indicates if a stream should be removed and re-created by force (if it exists). False by default. | |

Important:

- Service type for 'stream.t-streams' stream can be 'service.t-streams' only.
- Service type for 'stream.apache-kafka' stream can be 'service.apache-kafka' only.
- Service type for 'stream.sql-database' stream can be 'service.sql-database' only.
- Service type for 'stream.elasticsearch' stream can be 'service.elasticsearch' only.
- Service type for 'stream.restful' stream can be 'service.restful' only.

Note: * - required field.

Create a new stream

Request method: POST

Request format:

```
/v1/streams
```

Таблица 3.20: Response

| Status code | Description |
|-------------|---|
| 201 | Stream <stream name> has been created. |
| 400 | Cannot create stream. Errors: <list-of-errors>. |
| 500 | Internal server error. |

Request example:

```
{
  "name": "tstream-2",
  "description": "Tstream example",
  "partitions": 3,
  "service": "some-tstrq-service",
  "type": "stream.t-streams",
  "tags": ["lorem", "ipsum"]
}
```

Success response example:

```
{
  "status-code": 201,
  "entity": {
    "message": "Stream 'tstream-2' has been created."
  }
}
```

Error response example:

```
{
  "status-code": 400,
  "entity": {
    "message": "Cannot create stream. Errors: <creation_errors_string>."
  }
}
```

Get list of all streams

Request method: GET

Request format:

```
/v1/streams
```

Таблица 3.21: Response

| Status code | Description |
|-------------|------------------------|
| 200 | List of streams. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "streams": [
      {
        "name": "tstream-2",
        "description": "Tstream example",
        "partitions": 3,
        "service": "some-tstrq-service",
        "type": "stream.t-streams",
        "tags": ["lorem", "ipsum"]
      }
    ]
  }
}
```

```
"creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
},
{
  "name": "kafka-stream",
  "description": "One of the streams",
  "partitions": 1,
  "service": "some-kfkq-service",
  "type": "stream.apache-kafka",
  "tags": ["lorem", "ipsum"],
  "replicationFactor": 2
  "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
}
]
}
}
```

Get list of streams types

Request method: GET

Request format:

```
/v1/streams/_types
```

Таблица 3.22: Response

| Status code | Description |
|-------------|------------------------|
| 200 | List of types. |
| 500 | Internal server error. |

Success response example:

```
{
  entity: {
    types: [
      {
        id: "stream.restful",
        name: "RESTful"
      },
      {
        id: "stream.elasticsearch",
        name: "Elasticsearch"
      },
      {
        id: "stream.t-streams",
        name: "T-streams"
      },
      {
        id: "stream.apache-kafka",
        name: "Apache Kafka"
      },
      {
        id: "stream.sql-database",
        name: "SQL database"
      }
    ]
  }
}
```

```
{,
  status-code: 200
}
```

Get stream by name

Request method: GET

Request format:

```
/v1/streams/{name}
```

Таблица 3.23: Response

| Status code | Description |
|-------------|--|
| 200 | Stream. |
| 404 | Stream <stream name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "entity": {
    "stream": {
      "name": "echo-response",
      "description": "Tstream for demo",
      "service": "tstream_test_service",
      "tags": [
        "ping",
        "station"
      ],
    },
    "force": false,
    "partitions": 1,
    "type": "stream.t-streams"
  },
  "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
},
  "statusCode": 200
}
```

Error response example:

```
{
  "status-code": 404,
  "entity": {
    "message": "Stream 'Tstream-3' has not been found."
  }
}
```

Delete stream by name

Request method: DELETE

Request format:

```
/v1/streams/{name}
```

Таблица 3.24: Response

| Status code | Description |
|-------------|--|
| 200 | Stream <stream name> has been deleted. |
| 404 | Stream <stream name> has not been found. |
| 422 | Cannot delete stream <stream name>. Stream is used in instances. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Stream 'tstr-1' has been deleted."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Stream 'output-stream' has not been found."
  },
  "status-code": 404
}
```

Get instances related to a stream (by stream name)

Request method: GET

Request format:

```
/v1/streams/{name}/related
```

Таблица 3.25: Response

| Status code | Description |
|-------------|--|
| 200 | List of instances |
| 404 | Stream <stream name> has not been found. |
| 500 | Internal server error |

Success response example:

```
{
  "entity": {
    "instances": [
      "pingstation-output",
      "john"
    ]
  },
  "statusCode": 200
}
```

Error response example:

```
{
  "entity": {
    "message": "Stream 'output-stream' has not been found."
  },
  "status-code": 404
}
```

Tip: A full range of error responses can be found at `Streams_Errors`

3.11.5 CRUD REST API for Configurations

The range of REST API methods described below allows to create or delete configuration, get the information on the configuration, get the list of configurations existing in the system, get list of domains.

Configuration fields

| Field | Format | Description | Requirements |
|---------|--------|----------------------------------|---|
| name* | String | Name of the configuration (key). | Should be unique and contain digits, lowercase letters, hyphens or periods and start with a letter. |
| value* | String | Value of configuration. | |
| domain* | String | Name of config-domain. | (see the table below) |

Note: * - required field.

{config-domain} must be one of the following values:

| Types |
|--------------------------------|
| configuration.system |
| configuration.t-streams |
| configuration.apache-kafka |
| configuration.elasticsearch |
| configuration.apache-zookeeper |
| configuration.sql-database |

Create a new configuration

Request method: POST

Request format:

```
/v1/config/settings
```


Таблица 3.26: Response

| Status code | Description |
|-------------|--|
| 201 | <config-domain> configuration <name> has been created. |
| 400 | Cannot create <config-domain> configuration. Errors: <list-of-errors>. |
| 500 | Internal server error. |

Request json example:

```
{
  "name": "crud-rest-host",
  "value": "localhost",
  "domain": "system"
}
```

Error response example:

```
{
  "status-code": 400,
  "entity": {
    "message": "Cannot create system configuration. Errors: <creation_errors_string>."
  }
}
```

Get a configuration by name

Request method: GET

Request format:

```
/v1/config/settings/{config-domain}/{name}
```

Таблица 3.27: Response

| Status code | Description |
|-------------|--|
| 200 | Json with requested configuration for specific config domain. |
| 400 | Cannot recognize configuration domain <config-domain>. Domain must be one of the following values: 'configuration.system, configuration.t-streams, configuration.apache-kafka, configuration.elasticsearch, configuration.apache-zookeeper, configuration.sql-database'. |
| 404 | <config-domain> configuration <name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "configSetting": {
      "name": "crud-rest-host",
      "value": "localhost",
      "domain": "configuration.system"
    },
    "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
  }
}
```

```
}  
}
```

Error response example:

```
{  
  "entity": {  
    "message": "Cannot recognize configuration domain 'elasticsearch'. Domain must be one of the following  
↪ values: 'configuration.system, configuration.t-streams, configuration.apache-kafka, configuration.elasticsearch,  
↪ configuration.apache-zookeeper, configuration.sql-database'."  
  },  
  "status-code": 400  
}
```

Get all configurations for specific config domain

Request method: GET

Request format:

```
/v1/config/settings/{config-domain}
```

Таблица 3.28: Response

| Status code | Description |
|-------------|--|
| 200 | Json of set of configurations for specific config domain. |
| 400 | Cannot recognize configuration domain <config-domain>. Domain must be one of the following values: 'configuration.system, configuration.t-streams, configuration.apache-kafka, configuration.elasticsearch, configuration.apache-zookeeper, configuration.sql-database'. |
| 500 | Internal server error. |

Success response example:

```
{  
  "status-code": 200,  
  "entity": {  
    "configSettings": [  
      {  
        "name": "crud-rest-host",  
        "value": "localhost",  
        "domain": {config-domain}  
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"  
      },  
      {  
        "name": "crud-rest-port",  
        "value": "8000",  
        "domain": {config-domain}  
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"  
      }  
    ]  
  }  
}
```

Error response example:

```
{
  "entity": {
    "message": "Cannot recognize configuration domain 'elasticsearch'. Domain must be one of the following
↪ values: 'configuration.system, configuration.t-streams, configuration.apache-kafka, configuration.elasticsearch,
↪ configuration.apache-zookeeper, configuration.sql-database'."
  },
  "status-code": 400
}
```

Delete a configuration by name

Request method: DELETE

Request format:

```
/v1/config/settings/{config-domain}/{name}
```

Таблица 3.29: Response

| Status code | Description |
|-------------|--|
| 200 | <config-domain> configuration <name> has been deleted. |
| 400 | Cannot recognize configuration domain <config-domain>. Domain must be one of the following values: 'configuration.system, configuration.t-streams, configuration.apache-kafka, configuration.elasticsearch, configuration.apache-zookeeper, configuration.sql-database'. |
| 404 | <config-domain> configuration <name> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "entity": {
    "message": "Configuration 'system.crud-rest-host' has been deleted."
  },
  "status-code": 200
}
```

Error response example:

```
{
  "entity": {
    "message": "Cannot recognize configuration domain 'system'. Domain must be one of the following values:
↪ 'configuration.system, configuration.t-streams, configuration.apache-kafka, configuration.elasticsearch,
↪ configuration.apache-zookeeper, configuration.sql-database'."
  },
  "status-code": 400
}
```

Get all configurations

Request method: GET

Request format:

/v1/config/settings

Таблица 3.30: Response

| Status code | Description |
|-------------|-------------------------------|
| 200 | Json of set of configurations |
| 500 | Internal server error |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "configSettings": [
      {
        "name": "crud-rest-host",
        "value": "localhost",
        "domain": "configuration.system"
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
      },
      {
        "name": "crud-rest-port",
        "value": "8000",
        "domain": "configuration.system"
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
      },
      {
        "name": "session.timeout",
        "value": "7000",
        "domain": "configuration.apache-zookeeper"
        "creationDate": "Thu Jul 20 08:32:51 NOVT 2017"
      }
    ]
  }
}
```

Get list of domains

Request method: GET

Request format:

/v1/config/settings/domains

Таблица 3.31: Response

| Status code | Description |
|-------------|------------------------|
| 200 | Set of domains. |
| 500 | Internal server error. |

Success response example:

```
{
  entity: {
    domains: [
```

```

{
  id: "configuration.sql-database",
  name: "SQL database"
},
{
  id: "configuration.elasticsearch",
  name: "Elasticsearch"
},
{
  id: "configuration.apache-zookeeper",
  name: "Apache Zookeeper"
},
{
  id: "configuration.system",
  name: "System"
},
{
  id: "configuration.t-streams",
  name: "T-streams"
},
{
  id: "configuration.apache-kafka",
  name: "Apache Kafka"
}
]
},
status-code: 200
}

```

Tip: A full range of error responses can be found at [Config_Settings_Errors](#)

3.11.6 CRUD REST API for Custom Files

The range of REST API methods described below allows to upload a custom jar or file, download it to your computer, get list of custom jars or files in the system and delete a custom jar or file.

Custom jars

Upload custom jar

Request method: POST

Request format:

```
/v1/custom/jars
```

Content-type: multipart/form-data

Attachment: java-archive as field 'jar'

Example of source message:

```
POST /v1/modules HTTP/1.1
HOST: 192.168.1.174:18080
content-type: multipart/form-data; boundary=----WebKitFormBoundaryPaRdSyADUNG08o8p
content-length: 1093

-----WebKitFormBoundaryPaRdSyADUNG08o8p
Content-Disposition: form-data; name="jar"; filename="file.jar"
Content-Type: application/x-java-archive
..... //file content
-----WebKitFormBoundaryPaRdSyADUNG08o8p--
```

Таблица 3.32: Response

| Status code | Description |
|-------------|---|
| 200 | Custom jar <file_name> has been uploaded. |
| 400 | Cannot upload custom jar. Errors: <list-of-errors>. ('Specification.json is not found or invalid.'; 'Custom jar <file_name> already exists.'; 'Cannot upload custom jar <file_name>'. Custom jar with name <name_from_specification> and version <version_from_specification> already exists.') |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Custom jar 'Custom_jar.jar' has been uploaded."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Cannot upload custom jar. Errors: Custom jar 'Upload_custom_jar.jar' already exists."
  },
  "status-code": 400
}
```

Download a custom jar by file name

Request method: GET

Request format:

```
/v1/custom/jars/{custom-jar-file-name}
```

Response headers example:

```
Access-Control-Allow-Credentials : true
Access-Control-Allow-Headers : Token, Content-Type, X-Requested-With
Access-Control-Allow-Origin : *
Content-Disposition : attachment; filename=sj-transaction-generator-1.0-SNAPSHOT.jar
Content-Type : application/java-archive
```

Date : Wed, 07 Dec 2016 08:33:54 GMT
 Server : akka-http/2.4.11
 Transfer-Encoding : chunked

Таблица 3.33: Response

| Status code | Description |
|-------------|--|
| 200 | Jar-file for download. |
| 404 | Jar <custom-jar-file-name> has not been found. |
| 500 | Internal server error. |

Success response:

A jar file is downloaded.

Error response example:

```
{
  "entity": {
    "message": "Jar 'sj-transaction-generator-1.0-SNAPSHOT.jar' has not been found."
  },
  "status-code": 404
}
```

Download a custom jar by name and version

Request method: GET

Request format:

```
/v1/custom/jars/{custom-jar-name}/{custom-jar-version}/
```

Таблица 3.34: Response

| Status code | Description |
|-------------|--|
| 200 | Jar-file for download. |
| 404 | Jar <custom-jar-name>-<custom-jar-version> has not been found. |
| 500 | Internal server error. |

Success response:

A jar file is downloaded.

Error response example:

```
{
  "entity": {
    "message": "Internal server error: Prematurely reached end of stream."
  },
  "status-code": 500
}
```

Delete a custom jar by file name

Request method: DELETE

Request format:

```
/v1/custom/jars/{custom-jar-file-name}/
```

Таблица 3.35: Response

| Status code | Description |
|-------------|--|
| 200 | Jar named <custom-jar-file-name> has been deleted. |
| 404 | Jar <custom-jar-file-name> has not been found. |
| 500 | Internal server error |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Jar named 'regular-streaming-engine-1.0.jar' has been deleted"
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Jar 'com.bsw.batch.stream.engine' has not been found."
  },
  "status-code": 404
}
```

Delete a custom jar by name and version (from specification)

Request method: DELETE

Request format:

```
/v1/custom/jars/{custom-jar-name}/{custom-jar-version}/
```

Таблица 3.36: Response

| Status code | Description |
|-------------|---|
| 200 | Jar named <custom-jar-name> of the version <custom-jar-version> has been deleted. |
| 404 | Jar <custom-jar-name>-<custom-jar-version> has not been found. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Jar named 'com.bsw.regular.streaming.engine' of the version '0.1' has been deleted"
  }
}
```

Error response example:


```
{
  "entity": {
    "message": "Jar 'com.bwsb.batch.streaming.engine-2.0' has not been found."
  },
  "status-code": 404
}
```

Get list of all uploaded custom jars

Request method: GET

Request format:

```
/v1/custom/jars
```

Таблица 3.37: Response

| Status code | Description |
|-------------|-------------------------------|
| 200 | List of uploaded custom jars. |
| 500 | Internal server error. |

Success response example:

```
{
  "entity": {
    "customJars": [
      {
        "name": "com.bwsb.fw",
        "version": "1.0",
        "size": "98060032"
        "uploadDate": "Wed Jul 19 13:46:31 NOVT 2017"
      },
      {
        "name": "com.bwsb.tg",
        "version": "1.0",
        "size": "97810217"
        "uploadDate": "Wed Jul 19 13:46:31 NOVT 2017"
      }
    ]
  },
  "status-code": 200
}
```

Custom files

Upload a custom file

Request method: POST

Request format:

```
/v1/custom/files
```

Content-type: multipart/form-data

Attachment: any file as field 'file', text field "description"

Таблица 3.38: Response

| Status code | Description |
|-------------|---|
| 200 | Custom file <custom-jar-file-name> has been uploaded. |
| 400 | Request is missing required form field 'file'. |
| 409 | Custom file <custom-jar-file-name> already exists. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Custom file <custom-jar-file-name> has been uploaded."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Request is missing required form field 'file'."
  },
  "status-code": 400
}
```

Download a custom file by file name

Request method: GET

Request format:

```
/v1/custom/files/{custom-jar-file-name}
```

Response format for file download:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Token, Content-Type, X-Requested-With
Content-Disposition: attachment; filename=GeoIPASNum.dat
Server: akka-http/2.4.11
Date: Wed, 07 Dec 2016 09:16:22 GMT
Transfer-Encoding: chunked
Content-Type: application/octet-stream
```

Таблица 3.39: Response

| Status code | Description |
|-------------|--|
| 200 | File for download. |
| 404 | Custom file <custom-jar-file-name> has not been found. |
| 500 | Internal server error. |

Success response format:

File for download is returned.

Error response example:

```
{
  "entity": {
    "message": "Custom file 'Custom_jar.jar' has not been found."
  },
  "status-code": 404
}
```

Delete a custom file

Request method: DELETE

Request format:

```
/v1/custom/files/{custom-jar-file-name}
```

Таблица 3.40: Response

| Status code | Description |
|-------------|--|
| 200 | Custom file <custom-jar-file-name> has been deleted. |
| 404 | Custom file <custom-jar-file-name> has not been found. |
| 500 | Internal server error |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Custom file 'text.txt' has been deleted."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Custom file 'customfile.txt' has not been found."
  },
  "status-code": 404
}
```

Get list of all uploaded custom files

Request method: GET

Request format:

```
/v1/custom/files
```

Таблица 3.41: Response

| Status code | Description |
|-------------|--------------------------------|
| 200 | List of uploaded custom files. |
| 500 | Internal server error. |

Success response example:

```
{
  "entity": {
    "customFiles": [
      {
        "name": "GeoIPASNum.dat",
        "description": "",
        "uploadDate": "Mon Jul 04 10:42:03 NOVT 2016",
        "size": "46850"
      },
      {
        "name": "GeoIPASNumv6.dat",
        "description": "",
        "uploadDate": "Mon Jul 04 10:42:58 NOVT 2016",
        "size": "52168"
      }
    ]
  },
  "status-code": 200
}
```

3.11.7 CRUD REST API for Modules

This is the CRUD REST API for modules uploaded as jar files, instantiated and running modules as well as for custom jar files.

Таблица 3.42: Module types

| Types |
|-------------------|
| input-streaming |
| regular-streaming |
| batch-streaming |
| output-streaming |

Таблица 3.43: Specification fields

| Field | Format | Description |
|-------------------------|---------|---|
| name* | String | The unique name for a module. |
| description | String | The description for a module. |
| version* | String | The module version. |
| author | String | The module author. |
| license | String | The software license type for a module. |
| inputs* | IStream | The specification for the inputs of a module. |
| outputs* | IStream | The specification for the outputs of a module. |
| module-type* | String | The type of a module. One of [input-streaming, output-streaming, batch-streaming, regular-streaming]. |
| engine-name* | String | The name of the computing core of a module. |
| engine-version* | String | The version of the computing core of a module. |
| validator-class* | String | The absolute path to class that is responsible for a validation of launch options. |
| executor-class* | String | The absolute path to class that is responsible for a running of module. |
| batch-collector-class** | String | The absolute path to class that is responsible for a batch collecting of batch-streaming module. |

IStream for inputs and outputs has the following structure:

Таблица 3.44: IStream fields

| Field | Format | Description |
|--------------|---------------|--|
| cardinality* | Array[Int] | The boundary of interval in that a number of inputs can change. Must contain 2 items. |
| types* | Array[String] | The enumeration of types of inputs. Can contain only [stream.t-streams, stream.apache-kafka, stream.elasticsearch, stream.sql-database, stream.restful, input] |

Note: * - required field, ** - required for batch-streaming field

Upload a module

Request method: POST

Request format:

```
/v1/modules
```

Content-type: multipart/form-data

Attachment: java-archive as field 'jar'

Example of source message:

```
POST /v1/modules HTTP/1.1
HOST: 192.168.1.174:18080
content-type: multipart/form-data; boundary=----WebKitFormBoundaryPaRdSyADUNG08o8p
content-length: 109355206
```

```
-----WebKitFormBoundaryPaRdSyADUNG08o8p
Content-Disposition: form-data; name="jar"; filename="sj-stub-batch-streaming-1.0-SNAPSHOT.jar"
Content-Type: application/x-java-archive
..... //file content
-----WebKitFormBoundaryPaRdSyADUNG08o8p--
```

Таблица 3.45: Response

| Status code | Description |
|-------------|--|
| 200 | Jar file <file_name> of module has been uploaded. |
| 400 | <ol style="list-style-type: none"> 1. Cannot upload jar file <file_name> of module. Errors: file <file_name> does not have the .jar extension. 2. Cannot upload jar file <file_name> of module. Errors: module <module-type>-<module-name>-<module-version> already exists. 3. Cannot upload jar file <file_name> of module. Errors: file <file_name> already exists. 4. Other errors. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Jar file 'regular-module.jar' of module has been uploaded."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Cannot upload jar file 'regular-module.jar' of module. Errors: file 'regular-module.jar' already exists."
  },
  "status-code": 400
}
```

Download jar of uploaded module

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/
```

Response headers example:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Token, Content-Type, X-Requested-With
Content-Disposition: attachment; filename=sj-stub-batch-streaming-1.0-SNAPSHOT.jar
Server: akka-http/2.4.11
```

Date: Wed, 07 Dec 2016 05:45:45 GMT
 Transfer-Encoding: chunked
 Content-Type: application/java-archive

Таблица 3.46: Response

| Status code | Description |
|-------------|--|
| 200 | Jar-file for download |
| 404 | <ol style="list-style-type: none"> 1. Module ‘<module_type>-<module_name>-<module_version>’ has not been found. 2. Jar of module ‘<module_type>-<module_name>-<module_version>’ has not been found in the storage. |
| 500 | Internal server error |

Success response:

A jar file is downloaded.

Error response example:

```
{
  "entity": {
    "message": "Module 'regular-streaming-process-1.0' has not been found."
  },
  "status-code": 404
}
```

Delete uploaded module

Request method: DELETE

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/
```

Таблица 3.47: Response

| Status code | Description |
|-------------|--|
| 200 | Module {module-name} for type {module-type} has been deleted |
| 404 | <ol style="list-style-type: none"> 1. Module ‘<module_type>-<module_name>-<module_version>’ has not been found. 2. Jar of module ‘<module_type>-<module_name>-<module_version>’ has not been found in the storage. |
| 422 | <ol style="list-style-type: none"> 1. It’s impossible to delete module ‘<module_type>-<module_name>-<module_version>’. Module has instances. 2. Cannot delete file ‘<module-filename>’ |
| 500 | Internal server error |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "message": "Module 'regular-streaming-com.bsw.sj.stub-1.0' has been deleted."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Module 'regular-streaming-RegularModule-1.0' has not been found."
  },
  "status-code": 404
}
```

Get list of all uploaded modules

Request method: GET

Request format:

```
/v1/modules
```

Таблица 3.48: Response

| Status code | Description |
|-------------|--------------------------|
| 200 | List of uploaded modules |
| 500 | Internal server error |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "modules": [
      {
        "moduleType": "regular-streaming",
        "moduleName": "com.bsw.sj.stub",
        "moduleVersion": "0.1",
        "size": "68954210"
        "uploadDate": "Wed Jul 19 13:46:31 NOVT 2017"
      },
      {
        "moduleType": "batch-streaming",
        "moduleName": "com.bsw.sj.stub-win",
        "moduleVersion": "0.1",
        "size": "69258954"
        "uploadDate": "Wed Jul 19 13:46:31 NOVT 2017"
      }
    ]
  }
}
```


Get list of types of modules

Request method: GET

Request format:

```
/v1/modules/_types
```

Таблица 3.49: Response

| Status code | Description |
|-------------|-----------------------|
| 200 | List of types |
| 500 | Internal server error |

Success response example:

```
{
  "entity": {
    "types": [
      "batch-streaming",
      "regular-streaming",
      "output-streaming",
      "input-streaming"
    ]
  },
  "statusCode": 200
}
```

Get list of all uploaded modules for such type

Request method: GET

Request format:

```
/v1/modules/{module-type}
```

Таблица 3.50: Response

| Status code | Description |
|-------------|--|
| 200 | Uploaded modules for type <module-type> + <list-modules-for-type>. |
| 400 | Module type <module-type> does not exist. |
| 500 | Internal server error. |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "modules": [
      {
        "moduleType": "regular-streaming",
        "moduleName": "com.bwsj.sj.stub",
        "moduleVersion": "0.1",
        "size": 106959926
        "uploadDate": "Wed Jul 19 13:46:31 NOVT 2017"
      }
    ]
  }
}
```

```
]
}
}
```

Error response example:

```
{
  "entity": {
    "message": "Module type 'output-stream' does not exist."
  },
  "status-code": 400
}
```

Get specification for uploaded module

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/specification
```

Таблица 3.51: Response

| Status code | Description |
|-------------|---|
| 200 | Specification json (see <code>Json_schema</code>). |
| 404 | <ol style="list-style-type: none">1. Module ‘<module_type>-<module_name>-<module_version>’ has not been found.2. Jar of module ‘<module_type>-<module_name>-<module_version>’ has not been found in the storage. |
| 500 | Internal server error (including errors related to incorrect module type or nonexistent module). |

Success response example:

```
{
  "entity": {
    "specification": {
      "name": "batch-streaming-stub",
      "description": "Stub module by BW",
      "version": "1.0",
      "author": "John Smith",
      "license": "Apache 2.0",
      "inputs": {
        "cardinality": [
          1,
          10
        ],
        "types": [
          "stream.apache-kafka",
          "stream.t-streams"
        ]
      },
      "outputs": {
        "cardinality": [
```

```
    1,  
    10  
  ],  
  "types": [  
    "stream.t-streams"  
  ]  
},  
"moduleType": "batch-streaming",  
"engineName": "com.bwsj.batch.streaming.engine",  
"engineVersion": "1.0",  
"options": {  
  "opt": 1  
},  
"validatorClass": "com.bwsj.stubs.module.batch_streaming.Validator",  
"executorClass": "com.bwsj.stubs.module.batch_streaming.Executor"  
}  
},  
"statusCode": 200  
}
```

Error response example:

```
{  
  "entity": {  
    "message": "Module 'regular-streaming-RegularModule-1.0' has not been found."  
  },  
  "status-code": 400  
}
```

Tip: A full range of error responses can be found at `Modules_Errors`

3.11.8 CRUD REST API for Instances

The range of REST API methods described below allows to create an instance of a module, get the list of existing instances, get the settings of a specific instance, start and stop an instance and get the instance tasks information as well as delete an instance of a specific module.

Create an instance of a module

Request method: POST

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/
```

Note: The name of an input stream should contain the “/split” suffix (if stream’s partitions should be distributed between the tasks) or “/full” (if each task should process all partitions of the stream). The stream has a ‘split’ mode as default. (see [Execution plan](#))

Instance fields

General instance fields

| Field name | Format | Description | Example |
|------------------------------|---------------|---|---|
| name* | String | Required field, uniq name of creating instance. Must contain only letters, digits or hyphens. | stub-reg-instance-1 |
| description | String | Description of instance | Test instance for regular module |
| parallelism | Int or String | Value may be integer or 'max' string. If 'max', then parallelims equals minimum count of partitions of streams (1 by default) | max |
| options | JsonObject | Json with options for the module. It is validated by implementation of the Streaming Validator method in the module. That field can be set as required according to the Validator. | { 'opt1' : 10 } |
| perTaskCores | Double | Quantity of cores for task (1 by default) | 0.5 |
| perTaskRam | Long | Amount of RAM for task (1024 by default) | 256 |
| jvmOptions | JsonObject | Json with jvm-options. It is important to emphasize that mesos kill a task if it uses more memory than 'perTaskRam' parameter. There is no options by default. Defined options in the example fit the perTaskRam=192 and it's recommended to lauch modules. In general, the sum of the following parameters: Xmx, XX:MaxDirectMemorySize and XX:MaxMetaspaceSize, should be less than perTaskRam; XX:MaxMetaspaceSize must be grater than Xmx by 32m or larger. | { '-Xmx': '32m', '-XX:MaxDirectMemorySize=': '4m', '-XX:MaxMetaspaceSize=': '96m' } |
| nodeAttributes | JsonObject | Json with map attributes for framework | { '+tag1' : 'val1', '-tag2' : 'val2' } |
| coordinationService | String | Service name of zookeeper service | zk_service |
| environmentVariables | JsonObject | Using in framework | { 'LIBPROCESS_IP' : '176.1.0.17' } |
| performanceReportingInterval | LongInterval | Interval for creating report of performance metrics of module in ms (60000 by default) | 5000696 |

Input-streaming instance fields

| Field name | Format | Description | Example |
|-----------------------|--------------|--|-----------|
| checkpointMode | String | Value must be time-interval or every-nth | every-nth |
| checkpointInterval | Int* | Interval for creating checkpoint | 100 |
| outputs* | List[String] | Names of output streams (must be stream.t-stream only) | [s3, s4] |
| duplicateCheck | Boolean | The flag points if every envelope (an envelope key) has to be checked on duplication or not. (false by default) Note: You can indicate the ‘duplicateCheck’ field in the instance to set up a default policy for message checking on duplication. Use the ‘InputEnvelope’ flag in the Modules of Input Type for special cases* | true |
| lookupHistory | *Int | How long an unique key of an envelope will stay in a queue for checking envelopes on duplication (in seconds). If it is not 0, entries that are older than this time and not updated for this time are evicted automatically accordingly to an eviction-policy. Valid values are integers between 0 and Integer.MAX VALUE. Default value is 0, which means infinite. | 1000 |
| queueMaxSize | *Int | Maximum size of the queue that contains the unique keys of envelopes. When maximum size is reached, the queue is evicted based on the policy defined at default-eviction-policy (should be greater than 271) | 500 |
| defaultEvictionPolicy | String | Must be only ‘LRU’ (Least Recently Used), ‘LFU’ (Least Frequently Used) or ‘NONE’ (NONE by default) | LRU |
| evictionPolicy | String | An eviction policy of duplicates for an incoming envelope. Must be only ‘fix-time’ (default) or ‘expanded-time’. If it is ‘fix-time’, a key of the envelope will be contained only {lookup-history} seconds. The ‘expanded-time’ option means, if a duplicate of the envelope appears, the key presence time will be updated | fix-time |
| backupCount | Int | The number of backup copies you want to have (0 by default, maximum 6). Sync backup operations have a blocking cost which may lead to latency issues. You can skip this field if you do not want your entries to be backed up, e.g. if performance is more important than backing up. | 2 |
| asyncBackupCount | Int | Flag points an every envelope (an envelope key) has to be checked on duplication or not (0 by default). The backup operations are performed at some point in time (non-blocking operation). You can skip this field if you do not want your entries to be backed up, e.g. if performance is more important than backing up. | 3 |

Regular-streaming instance fields

| Field name | Format | Description | Example |
|---------------------|--------------------|---|---------------------|
| checkpointMode | String | Value must be ‘time-interval’ or ‘every-nth’ | every-nth |
| checkpointInterval | Int* | Interval for creating checkpoint | 100 |
| inputs* | List[String] | Names of input streams. Name format must be <stream-name>/<‘full’ or ‘split’> (‘split’ by default). Stream must exist in database (must be stream.t-streams or stream.apache-kafka) | [s1/full, s2/split] |
| outputs* | List[String] | Names of output streams (must be stream.t-streams only) | [s3, s4] |
| startFrom | String or Datetime | Value must be ‘newest’, ‘oldest’ or datetime. If instance have kafka input streams, then ‘start-from’ must be only ‘oldest’ or ‘newest’ (newest by default) | newest |
| stateManagement | String | Must be ‘ram’ or ‘none’ (none by default) | ram |
| stateFullCheckpoint | Int | Interval for full checkpoint (100 by default) | 5 |
| eventWaitTime | Long | Idle timeout, when not messages (1000 by default) | 10000 |

Batch-streaming instance fields

| Field name | Format | Description | Example |
|---------------------|--------------------|---|-----------|
| outputs* | List[String] | Names of output streams (must be stream.t-stream only) | [s3, s4] |
| window | Int | Count of batches that will be contained into a window (1 by default). Must be greater than zero | 3 |
| slidingInterval | Int | The interval at which a window will be shifted (count of batches that will be removed from the window after its processing). Must be greater than zero and less or equal than window (1 by default) | 3 |
| inputs* | String | Names of input streams. Name format must be <stream-name>/<‘full’ or ‘split’> (‘split’ by default). Stream must exist in database (must be stream.t-streams or stream.apache-kafka) | [s1/full] |
| startFrom | String or Datetime | Value must be ‘newest’, ‘oldest’ or datetime. If instance have kafka input streams, then ‘start-from’ must be only ‘oldest’ or ‘newest’ (newest by default) | newest |
| stateManagement | String | Must be ‘ram’ or ‘none’ (none by default) | ram |
| stateFullCheckpoint | Int | Interval for full checkpoint (100 by default) | 5 |
| eventWaitTime | Long | Idle timeout, when not messages (1000 by default) | 10000 |

Output-streaming instance fields

| Field name | Format | Description | Example |
|--------------------|--------------------|---|-----------|
| checkpointMode | String | Value must be 'every-nth' | every-nth |
| checkpointInterval | Int* | Interval for creating checkpoint | 100 |
| input* | String | Names of input stream. Must be only 'stream.t-streams' type. Stream for this type of module is 'split' only. Stream must be exists in database. | s1 |
| output* | String | Names of output stream (must be stream.elasticsearch, stream.sql-database or stream.restful) | es1 |
| startFrom | String or Datetime | Value must be 'newest', 'oldest' or datetime (newest by default) | newest |

Note: * - required fields.

Input-streaming module json format:

```
{
  "name" : String,
  "description" : String,
  "outputs" : List[String],
  "checkpointMode" : "time-interval" | "every-nth",
  "checkpointInterval" : Int,
  "parallelism" : Int,
  "options" : {},
  "perTaskCores" : Double,
  "perTaskRam" : Int,
  "jvmOptions" : { "-Xmx": "32m", "-XX:MaxDirectMemorySize=": "4m", "-XX:MaxMetaspaceSize=": "96m" },
  "nodeAttributes" : {},
  "coordinationService" : String,
  "performanceReportingInterval" : Int,
  "lookupHistory" : Int,
  "queueMaxSize" : Int,
  "defaultEvictionPolicy" : "LRU" | "LFU",
  "evictionPolicy" : "fix-time" | "expanded-time",
  "duplicateCheck" : true | false,
  "backupCount" : Int,
  "asyncBackupCount" : Int
}
```

Regular-streaming module json format:

```
{
  "name" : String,
  "description" : String,
  "inputs" : List[String],
  "outputs" : List[String],
  "checkpointMode" : "time-interval" | "every-nth",
  "checkpointInterval" : Int,
  "stateManagement" : "none" | "ram",
  "stateFullCheckpoint" : Int,
  "parallelism" : Int,
  "options" : {},
}
```

```
"startFrom" : "oldest" | "newest" | datetime (as timestamp),
"perTaskCores" : Double,
"perTaskRam" : Int,
"jvmOptions" : { "-Xmx": "32m", "-XX:MaxDirectMemorySize=": "4m", "-XX:MaxMetaspaceSize=": "96m" },
"nodeAttributes" : {},
"eventWaitTime" : Int,
"coordinationService" : String,
"performanceReportingInterval" : Int
}
```

Batch-streaming module json format:

```
{
  "name" : String,
  "description" : String,
  "inputs" : [String],
  "stateManagement" : "none" | "ram",
  "stateFullCheckpoint" : Int,
  "parallelism" : Int,
  "options" : {},
  "startFrom" : "newest" | "oldest",
  "perTaskCores" : Double,
  "perTaskRam" : Int,
  "jvmOptions" : { "-Xmx": "32m", "-XX:MaxDirectMemorySize=": "4m", "-XX:MaxMetaspaceSize=": "96m" },
  "nodeAttributes" : {},
  "eventWaitTime" : Int,
  "coordinationService" : String,
  "performanceReportingInterval" : Int
}
```

Output-streaming module json format:

```
{
  "name" : String,
  "description" : String,
  "input" : String,
  "output" : String,
  "checkpointMode" : "every-nth",
  "checkpointInterval" : Int,
  "parallelism" : Int,
  "options" : {},
  "startFrom" : "oldest" | "newest" | datetime (as timestamp),
  "perTaskCores" : Double,
  "perTaskRam" : Int,
  "jvmOptions" : { "-Xmx": "32m", "-XX:MaxDirectMemorySize=": "4m", "-XX:MaxMetaspaceSize=": "96m" },
  "nodeAttributes" : {},
  "coordinationService" : String,
  "performanceReportingInterval" : Int
}
```

Request json example for creating a batch-streaming instance:

```
{
  "name" : "stub-instance-win",
  "description" : "",
  "mainStream" : "ubatch-stream",
  "batchFillType": {
    "typeName" : "every-nth",
```



```

    "value" : 100
  },
  "outputs" : ["ubatch-stream2"],
  "stateManagement" : "ram",
  "stateFullCheckpoint" : 1,
  "parallelism" : 1,
  "options" : {},
  "startFrom" : "oldest",
  "perTaskCores" : 2,
  "perTaskRam" : 192,
  "jvmOptions" : {
    "-Xmx": "32m",
    "-XX:MaxDirectMemorySize=": "4m",
    "-XX:MaxMetaspaceSize=": "96m"
  },
  "nodeAttributes" : {},
  "eventWaitTime" : 10000,
  "coordinationService" : "a-zoo",
  "performanceReportingInterval" : 50054585
}

```

Response

| Status code | Description |
|-------------|--|
| 201 | Instance <instance_name> for module <module_type>-<module_name>-<module_version> has been created. |
| 400 | <ol style="list-style-type: none"> Cannot create instance of module. The instance parameter ‘options’ haven’t passed validation, which is declared in a method, called ‘validate’. This method is owned by a validator class that implements StreamingValidator interface. Errors: {list-of-errors}. Cannot create instance of module. Errors: {list-of-errors}. |
| 404 | <ol style="list-style-type: none"> Module <module_type>-<module_name>-<module_version> has not been found. Jar of module <module_type>-<module_name>-<module_version> has not been found in the storage. |
| 500 | Internal server error (including errors related to incorrect module type or nonexistent module). |

Success response json example of a created instance:

```

"instance": {
  "stage": {
    "state": "to-handle",
    "datetime": 1481092354533,
    "duration": 0
  }
},
"status": "ready",
"name": "stub-instance-win",
"description": "",
"parallelism": 1,
"options": {

```

```
{,
  "engine": "com.bwsb.batch.streaming.engine-1.0",
  "window": 1,
  "outputs": [
    "ubatch-stream2"
  ],
  "perTaskCores": 2.0,
  "perTaskRam": 128,
  "jvmOptions": {
    "-Xmx": "32m",
    "-XX:MaxDirectMemorySize=": "4m",
    "-XX:MaxMetaspaceSize=": "96m"
  },
  "nodeAttributes": {

  },
  "coordinationService": "a-zoo",
  "environmentVariables": {

  },
  "performanceReportingInterval": 50054585,
  "inputs": [
    "ubatch-stream"
  ],
  "slidingInterval": 1,
  "executionPlan": {
    "tasks": {
      "stub-instance-win-task0": {
        "inputs": {
          "ubatch-stream": [
            0,
            2
          ]
        }
      }
    }
  },
  "startFrom": "oldest",
  "stateManagement": "ram",
  "stateFullCheckpoint": 1,
  "eventWaitTime": 10000,
  "restAddress" : ""
}
```

Error response example:

```
{
  "entity": {
    "message": "Module 'input-streaming-Instance1-2.0' has not been found."
  },
  "status-code": 404
}
```

Instance statuses

Instance may have one of the following statuses:

- ready - a newly created instance and not started yet;
- starting - a recently launched instance but not started yet (right after the “Start” button is pushed);
- started - the launched instance started to work;
- stopping - a started instance in the process of stopping (right after the “Stop” button is pushed);
- stopped - an instance that has been stopped;
- deleting - an instance in the process of deleting (right after the “Delete” button is pressed);
- failed - an instance that has been launched but in view of some errors is not started;
- error - an error is detected at stopping or deleting an instance.

Execution plan

A created instance contains an execution plan that is provided by the system.

Execution plan consists of tasks. The number of tasks equals to a parallelism parameter.

Each task has a unique name within execution plan. Also the task has a set of input stream names and their intervals of partitions.

Altogether it provides the information of the sources from which the data will be consumed.

Execution plan example:

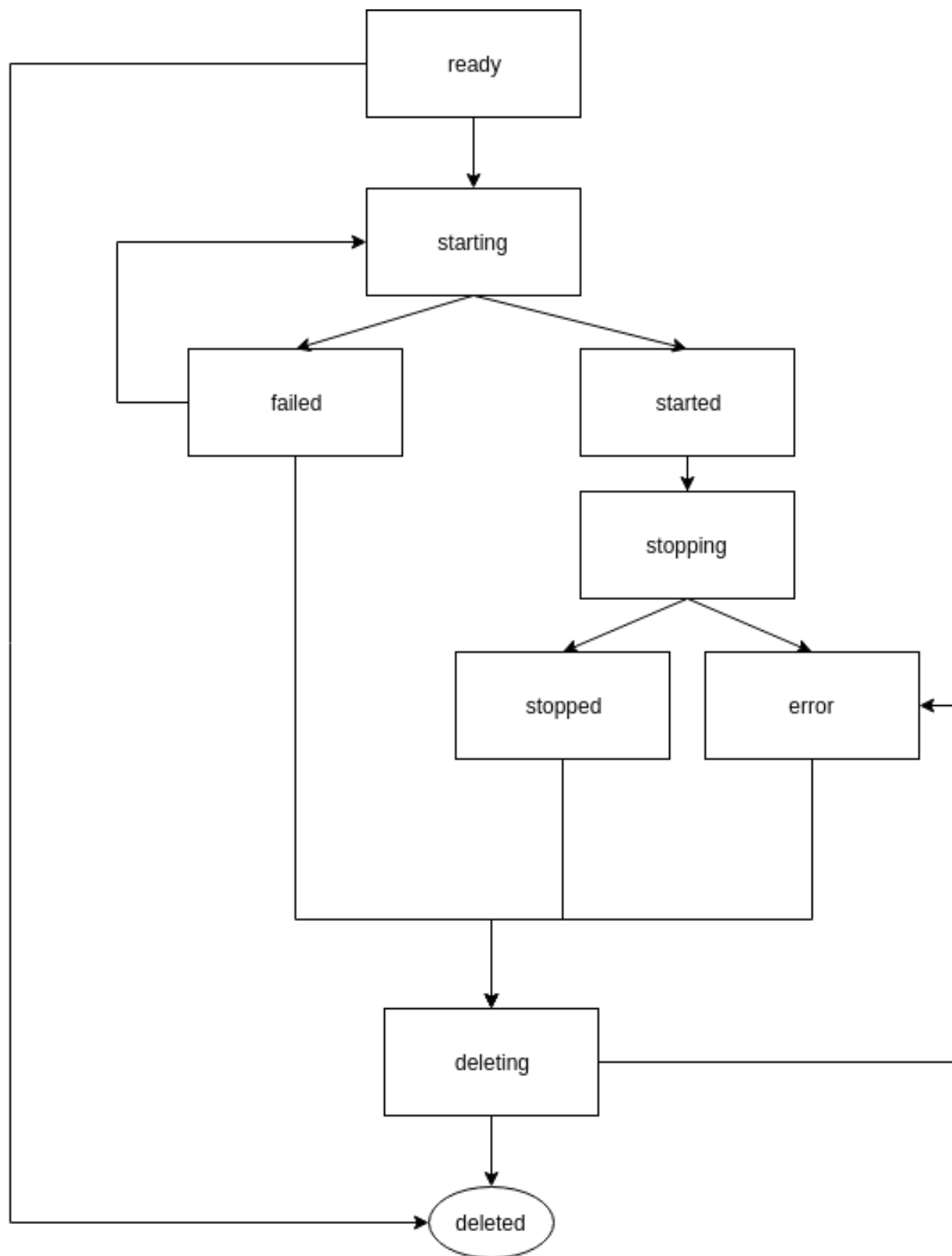
```
"executionPlan": {  
  "tasks": {  
    "stub-instance-win-task0": {  
      "inputs": {  
        "ubatch-stream": [  
          0,  
          2  
        ]  
      }  
    }  
  }  
}
```

Note: The execution plan doesn't exist in instances of an input module. An instance of an input-module contains a 'tasks' field.

Each task has a name, host and port. A host and a port define an address to which the data should be sent for the input module to process them.

Json format of 'tasks' field for instance of input module:

```
{  
  "instance-name-task0" : {  
    "host" : String,  
    "port" : Int  
  },  
}
```



```
"instance-name-task1" : {  
  "host" : String,  
  "port" : Int  
},  
"instance-name-taskN" : {  
  "host" : String,  
  "port" : Int  
}  
}
```

Stage

A created instance contains a stage that is provided by the system.

First of all it should be noted that a framework is responsible for launching instance.

The stage is used to display information about current status of framework. It allows you to follow start or stop processes of instance.

The stage consists of state, datetime and duration. Let's look at every parameter in detail.

1. State can have one of the following values. The value corresponds to an instance status:
 - to-handle - a newly created instance and not started yet;
 - starting - a recently launched instance but not started yet (right after the “Start” button is pushed);
 - started - the launched instance started to work;
 - stopping - a started instance that has been stopped (right after the “Stop” button is pushed);
 - stopped - an instance that has been stopped;
 - deleting - an instance in the process of deleting (right after the “Delete” button is pressed);
 - failed - an instance that has been launched but in view of some errors is not started;
 - error - an error is detected when stopping the instance.
2. Datetime defines the time when a state has been changed
3. Duration means how long a stage has got a current state. This field makes sense if a state field is in a ‘starting’, a ‘stopping’ or a ‘deleting’ status.

Json example of this field:

```
"stage": {  
  "state": "started",  
  "datetime": 1481092354533,  
  "duration": 0  
}
```

Get all instances

Request method: GET

Request format:

```
/v1/modules/instances
```

Таблица 3.52: Response

| Status code | Description |
|-------------|--|
| 200 | Json set of instances (in short format). |
| 500 | Internal server error. |

Success response json example:

```
{
  "status-code" : 200,
  "entity" : {[
    {
      "name": "instance-test"
      "moduleType": "batch-streaming"
      "moduleName": "com.bw.sw.sj.stub.win"
      "moduleVersion": "0.1"
      "description": ""
      "status" : "started"
      "restAddress" : "12.1.1.1:12:2900"
    },
    {
      "name": "reg-instance-test"
      "moduleType": "regular-streaming"
      "moduleName": "com.bw.sw.sj.stub.reg"
      "moduleVersion": "0.1"
      "description": ""
      "status" : "ready"
      "restAddress" : ""
    }
  ]}
}
```

Get instances related to a specific module

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/related
```

Таблица 3.53: Response

| Status code | Description |
|-------------|--|
| 200 | List of instances |
| 404 | <ol style="list-style-type: none"> 1. Module ‘<module_type>-<module_name>-<module_version>’ has not been found. 2. Jar of module ‘<module_type>-<module_name>-<module_version>’ has not been found in the storage. |
| 500 | Internal server error (including errors related to incorrect module type or nonexistent module) |

Success response json example:

```
{
  "status-code": 200,
  "entity": {
    "instances": [
      "test-instance",
      "abc"
    ]
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Module 'output-streaming-OutputModule-1.0' has not been found."
  },
  "status-code": 400
}
```

Get all instances of a specific module

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/
```

Таблица 3.54: Response

| Status code | Description |
|-------------|--|
| 200 | List of instances of module |
| 404 | 1. Module ‘<module_type>-<module_name>-<module_version>’ has not been found. 2. Jar of module ‘<module_type>-<module_name>-<module_version>’ has not been found in the storage. |
| 500 | Internal server error (including errors related to incorrect module type or nonexistent module) |

Success response json example:

```
{
  "status-code": 200,
  "entity": {
    "instances": [
      {
      },
      {
      },
      {
      },
      ...,
      {
      }
    ]
  }
}
```

```
}  
|  
}  
}
```

Error response example:

```
{  
  "entity": {  
    "message": "Module 'output-streaming-OutputModule-1.0' has not been found."  
  },  
  "status-code": 400  
}
```

Get an instance of a specific module

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/{instance-name}/
```

Таблица 3.55: Response

| Status code | Description |
|-------------|--|
| 200 | Instance |
| 404 | Instance '<instance_name>' has not been found. |
| 500 | Internal server error |

Success response example:

```
{  
  "entity": {  
    "instance": {  
      "moduleName": "pingstation-output",  
      "moduleVersion": "1.0",  
      "moduleType": "output-streaming",  
      "stage": {  
        "state": "stopped",  
        "datetime": 1500966502569,  
        "duration": 0  
      },  
    },  
    "status": "stopped",  
    "name": "output-streaming-instance",  
    "description": "No description",  
    "parallelism": 1,  
    "options": {},  
    "perTaskCores": 1,  
    "perTaskRam": 1024,  
    "jvmOptions": {},  
    "nodeAttributes": {},  
    "coordinationService": "zk-service",  
    "environmentVariables": {},  
    "performanceReportingInterval": 60000,  
    "engine": "com.bwsb.output.streaming.engine-1.0",  
  }  
}
```



```

    "restAddress": "",
    "checkpointMode": "every-nth",
    "checkpointInterval": 3,
    "executionPlan": {
      "tasks": {
        "output-streaming-instance-task0": {
          "inputs": {
            "output-tstream": [
              0,
              0
            ]
          }
        }
      }
    },
    "startFrom": "newest",
    "input": "output-tstreamk",
    "output": "stream-es"
  }
},
"status-code": 200
}

```

Error response example:

```

{
  "entity": {
    "message": "Instance 'batch-streaming' has not been found."
  },
  "status-code": 404
}

```

Start an instance

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/{instance-name}/start/
```

Таблица 3.56: Response

| Status code | Description |
|-------------|--|
| 200 | Instance ‘<instance_name>’ is being launched. |
| 404 | Instance ‘<instance_name>’ has not been found. |
| 422 | Cannot start of instance. Instance has already launched. |
| 500 | Internal server error |

Note: To start an instance it should have a status: “failed”, “stopped” or “ready”.

When instance is starting, framework starts on Mesos.

Success response example:

```
{
  "status-code" : 200,
  "entity" : {
    "message" : "Instance '<instance_name>' is being launched."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Cannot start of instance. Instance has already launched."
  },
  "status-code": 422
}
```

Get the information about instance tasks

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/{instance-name}/tasks/
```

| Status code | Description |
|-------------|--|
| 200 | Instance framework tasks info. |
| 404 | Instance ‘<instance_name>’ has not been found. |
| 422 | Cannot get instance framework tasks info. The instance framework has not been launched. |
| 500 | Internal server error (including errors related to incorrect module type or nonexistent module and «Instance ‘<instance_name>’ has not been found.») |

Success response example:

```
{
  "status-code": 200,
  "entity": {
    "tasks": [
      {
        "state": "TASK_RUNNING",
        "directories": [
          {
            "name": "Mon Dec 05 11:33:47 NOVT 2016",
            "path": "http://stream-juggler.z1.netpoint-dc.com:5050/#/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-
↪S0/browse?path=/var/lib/mesos/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-S0/frameworks/c69ce526-c420-
↪44f4-a401-6b566b1a0823-0003/executors/pingstation-process-task0/runs/d9748d7a-3d0e-4bb6-
↪88eb-3a3340d133d8"
          },
          {
            "name": "Mon Dec 05 11:56:47 NOVT 2016",
            "path": "http://stream-juggler.z1.netpoint-dc.com:5050/#/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-
↪S0/browse?path=/var/lib/mesos/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-S0/frameworks/c69ce526-c420-
↪44f4-a401-6b566b1a0823-0003/executors/pingstation-process-task0/runs/8a62f2a4-6f3c-412f-
↪9d17-4f63e9052868"
          }
        ]
      }
    ],
  }
}
```

```

    "state-change": "Mon Dec 05 11:56:47 NOVT 2016",
    "reason": "Executor terminated",
    "id": "pingstation-process-task0",
    "node": "3599865a-47b1-4a17-9381-b708d42eb0fc-S0",
    "last-node": "3599865a-47b1-4a17-9381-b708d42eb0fc-S0"
  }
}
}
}

```

Error response example:

```

{
  "entity": {
    "message": "Cannot get instance framework tasks info. The instance framework has not been launched."
  },
  "status-code": 422
}

```

Stop an instance

Request method: GET

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/{instance-name}/stop/
```

| Status code | Description |
|-------------|--|
| 200 | Instance ‘<instance_name>’ is being stopped. |
| 404 | Instance ‘<instance_name>’ has not been found. |
| 422 | Cannot stop instance. Instance has not been started. |
| 500 | Internal server error (including errors related to incorrect module type or nonexistent module and «Instance ‘<instance_name>’ has not been found.») |

Note: An instance with the “started” status only can be stopped.

When the instance stops, the framework suspends on Mesos.

Success response example:

```

{
  "status-code" : 200,
  "entity" : {
    "message" : "Instance ' <instance_name> ' is being stopped."
  }
}

```

Error response example:

```

{
  "entity": {
    "message": "Cannot stop instance. Instance has not been started."
  },
}

```

```
"status-code": 422
}
```

Delete an instance of a specific module

Request method: DELETE

Request format:

```
/v1/modules/{module-type}/{module-name}/{module-version}/instance/{instance-name}/
```

Таблица 3.57: Response

| Status code | Description |
|-------------|---|
| 200 | <ol style="list-style-type: none">1. Instance ‘<instance_name>’ is being deleted.2. Instance ‘<instance_name>’ has been deleted. |
| 404 | Instance ‘<instance_name>’ has not been found. |
| 422 | Cannot delete of instance ‘<instance_name>’. Instance is not been stopped, failed or ready. |
| 500 | Internal server error |

Note: This process includes destruction of the framework on Mesos.

Success response example:

```
{
  "status-code" : 200,
  "entity" : {
    "message" : "Instance 'stub-instance-1' has been deleted."
  }
}
```

Error response example:

```
{
  "entity": {
    "message": "Instance 'output instance' has not been found."
  },
  "status-code": 404
}
```

Mesos Framework REST API

Request method: GET

Request format:

```
http://{rest-address}
```

Таблица 3.58: Response

| Status code | Description |
|-------------|--|
| 200 | Json set of instances (in short format). |
| 500 | Internal server error |

Success response json example:

```
entity: {
  "tasks": [
    {
      "state": "TASK_RUNNING",
      "directories": [
        "http://stream-juggler.z1.netpoint-dc.com:5050/#/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-S0/browse?
        ↪path=/var/lib/mesos/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-S0/frameworks/c69ce526-c420-44f4-a401-
        ↪6b566b1a0823-0003/executors/pingstation-process-task0/runs/d9748d7a-3d0e-4bb6-88eb-3a3340d133d8",
        "http://stream-juggler.z1.netpoint-dc.com:5050/#/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-S0/browse?
        ↪path=/var/lib/mesos/slaves/3599865a-47b1-4a17-9381-b708d42eb0fc-S0/frameworks/c69ce526-c420-44f4-a401-
        ↪6b566b1a0823-0003/executors/pingstation-process-task0/runs/8a62f2a4-6f3c-412f-9d17-4f63e9052868"
      ],
      "state-change": "Mon Dec 05 11:56:47 NOVT 2016",
      "reason": "Executor terminated",
      "id": "pingstation-process-task0",
      "node": "3599865a-47b1-4a17-9381-b708d42eb0fc-S0",
      "last-node": "3599865a-47b1-4a17-9381-b708d42eb0fc-S0"
    }
  ],
  "message": "Tasks launched"
}
```

The following information on tasks is returned:

- state - the status of task performance. The following options are possible:
 - “TASK_STAGING” - the task is created but is not started executing,
 - “TASK_RUNNING” - the task is launched and is being executed now,
 - “TASK_FAILED” - the task is failed,
 - “TASK_ERROR” - an error is detected in the task execution.
- directories - directories of tasks of the instance.
- state-change - the date of the last status change.
- reason - the reason for the task status change.
- id - the task id.
- node - name of node used by the task.
- last node - name of node that was used by a task before the status change.

Tip: A full range of error responses can be found at `Instances_Errors`

3.12 Glossary

Batch A minimum data set to collect events of a stream.

Configurations Settings used for the system work.

Checkpoint A saved “snapshot” of processed data at a specific time. The system saves the information about all consumed/recorded data by a module at that particular time, and in case of failure it restores the module work to the stage it was before the last checkpoint. If no checkpoint has been performed, the processing will start from the beginning.

Engine A base of the system. It provides basic I/O functionality for a module. It uses module settings to process data.

Envelope A specialized fundamental data structure, containing data and metadata. The metadata is required for exactly-once processing.

Event A Minimal data unit of a stream.

Exactly-once processing The processing of stream events only once.

Executor A part of a module that performs data processing.

External data source An external system which provides data for processing.

External data storage An external system which stores resulting data after processing.

Instance A set of settings determining the collaborative work of an engine and a module.

Metric A numeric value received as a result of aggregation.

Module A key program component in the system that contains the logic of data processing and settings validation.

Partition A part of a data stream allocated for convenience in stream processing.

Physical service One of the following services used for SJ-Platform: Apache Kafka, Apache Zookeeper, T-streams, Elasticsearch, SQL-database, any system which provides RESTful interface.

Provider An entity which contains general information to access a physical service.

Service An entity which contains specific information to access a physical service.

Sliding interval A step size at which a window slides forward. It can be less than a window size, and in this case some data will be processed more than once.

State A sort of a key-value storage for user data which is used to keep some global module variables related to processing. These variables are persisted and are recovered after a failure.

State storage service A service responsible for storing data state into a specified location (determined by instance settings). It is performed together with the checkpoint.

Stream A sequence of events happening randomly at irregular intervals. Streams in SJ-Platform differ by a physical service (that provides or stores data) and by purpose. According to a physical service, the types of streams existing in SJ-Platform are Apache Kafka, T-streams, Elasticsearch, SQL-database, RESTful. According to the purpose, streams can be: input streams, internal streams and output streams.

- **Input stream** - a sequence of data elements from data source. It may be of types: T-streams, Apache Kafka.
- **Internal stream** - a sequence of data elements transferred between system modules. It may be of type: T-streams.

- Output stream - a sequence of data elements which is saved in data storage. It may be of types: T-streams, Elasticsearch, SQL database, RESTful.

Task It is a Mesos term and, in general, it means a piece of work to be done. Within the system, a task is a logical part of an instance. Each task does the same work that is provided by a module. They are executed separately from each other.

Transaction A part of a partition consisting of events.

T-streams (transactional streams); a Scala library providing an infrastructure component which implements transactional messaging.

Window A set of batches of an unbounded stream. Grouping batches allows processing a series of events at one time. It is important in case the processing of events in a stream depends on other events of this stream or on events in another stream.

Indices and tables

- `genindex`
- `modindex`
- `search`