Streaming API

Prepared for: Dynatrace ESA Customers

Prepared by: Bart Oostlander, Senior Enterprise Solution Architect

May, 2020

Release 6.0

Contents

(Intentionally left blank)

SUMMARY

Objective

Dynatrace customers have expressed a desire to be able to receive a continuous stream of metrics and properties for selected Dynatrace entities into enterprise messaging infrastructures like Kafka. The Streaming API enables this functionality, both for entity metrics as well as user session metrics.

Output streams can be consumed by 3rd parties and/or persisted. It may subsequently be used for various purposes:

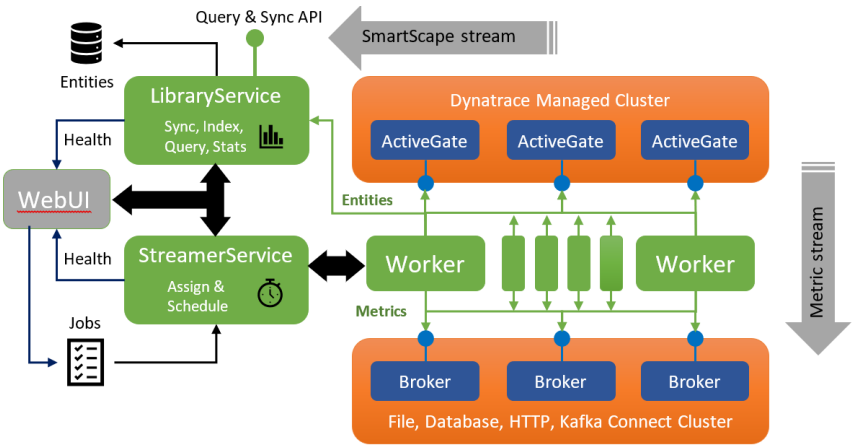
- Custom metric charting in 3rd-party dashboard products;
- Extraction of SmartScape data for CMDB synchronization or custom reporting.
- Consolidation of key performance data points, collected by various APM technologies, of entities and sessions;
- Automated remedial actions based on metric behavior.

Goals

The Streaming API continuously collects and streams metric values and properties, for subsets of entities and user sessions. It should be noted that there are inherent limits to the volume of data that can be retrieved from the Dynatrace API within a given timeframe. Likewise, there are inherent limitations to the number of messages that can be posted to, for instance, a Kafka message bus or a REST API in the allocated timeframes. The solution therefore aims to automatically adapt its deployment and parallelization strategies to maximize its throughput while not exceeding rate limits placed upon it. Every effort is made to utilize multiple, concurrent endpoints to distribute the load.

Solution

The solution consists of the following components:



- A web application to expose a UI in which the user can select entities, sessions, metrics and timeframes. The UI also reports on key deployment, performance and behavioral characteristics to help users make informed decisions with respect to the stress their selections impose on the Dynatrace API.
- A StreamerService that determines which jobs are due for execution at each minute, assigns the work, and controls the automated deployment of worker processes to maximize throughput.
- A variable collection of Worker processes that round-robin across available API endpoints to process their part of the overall workload.
- A LibraryService that analyzes and indexes all Dynatrace entities for fast filtering and retrieval. It exposes a public query interface that is (also) used by the StreamerService to select the appropriate entities for each job that is due.
- A ThrottleService that can be used as a stand-alone gateway to protect access to the Dynatrace API.
- A ConfigService that can be used to access arbitrary Dynatrace REST APIs (Managed only).

GETTING STARTED

Prerequisites

- nodeJS 10+ (12+ is recommended)
- NPM package manager
- MariaDB or MySQL database
- (Optionally) HeidiSQL to manage databases and users.

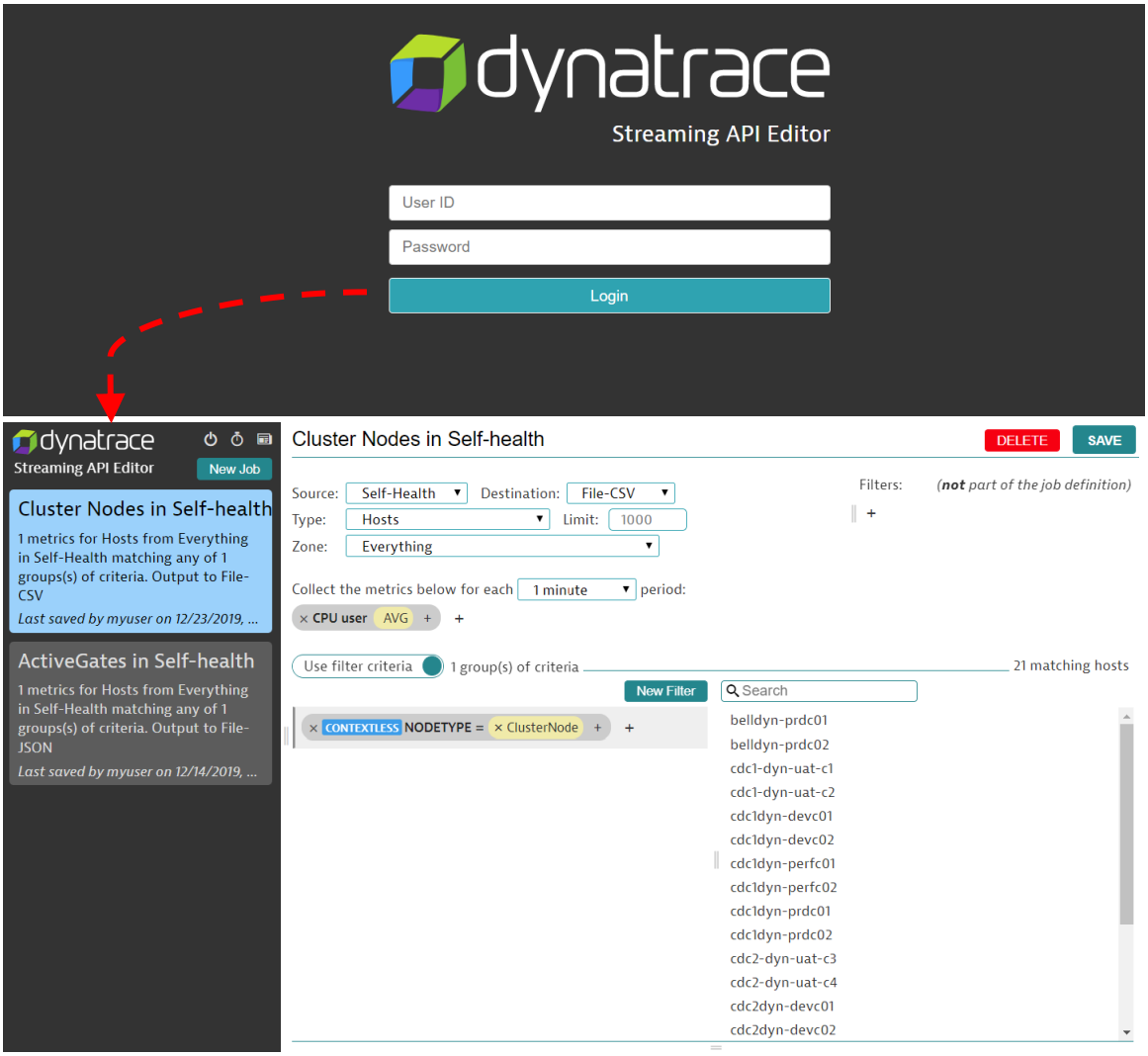
Installation

- Download the source code from GitLab (https://gitlab.com/BartOostlander/dynatrace-streamingapi) to your local system.
- Navigate to the /config directory and open 'config.json' in a text editor.
    - Under "Tenants" configure your tenants (including the Dynatrace API token).
    - Under "library" ensure that your tenant is referenced. Also, set the "launch" property to "embedded".
    - Under "database" configure the database adaptor class, location, port, user and password.
- In MariaDB or MySQL create a database named "StreamingAPI" (recommended name).
    - It is easiest to use HeidiSQL for this task.
    - Use the "`utf8-general-ci`" collation strategy.
    - Ensure that the user with which you connect to that database has "CREATE TABLE" permissions for the "StreamingAPI" database.
- Navigate to the /server directory.
    - Run '`nmp install`'. This will install all libraries required by the StreamingAPI.
    - Check that no error messages appear.
    - Run "`node StreamerService.js`".
    - Check that no error messages appear.
- Once the StreamerService is running, open your browser and navigate to http://localhost:8080.
    - Log in as "myuser" and password "mypassword". You can change this in the config.json file.
    - Click "Continue".
    - In the editor, click "New Job".
    - You should now see the hosts, etc. in the selected tenant.

USAGE

Launching the Editor

The UI is exposed at the port configured in `config.json` (eg. `http://localhost:8080`). The user must log on before the UI can access the StreamerService's data. For Managed instances, the user's credentials are validated against the first available tenant is configured in `config.json`. For SaaS instances, the user's credentials must match those specified in the 'account' setting for the first available tenant. Once logged on, the job editor will be shown.



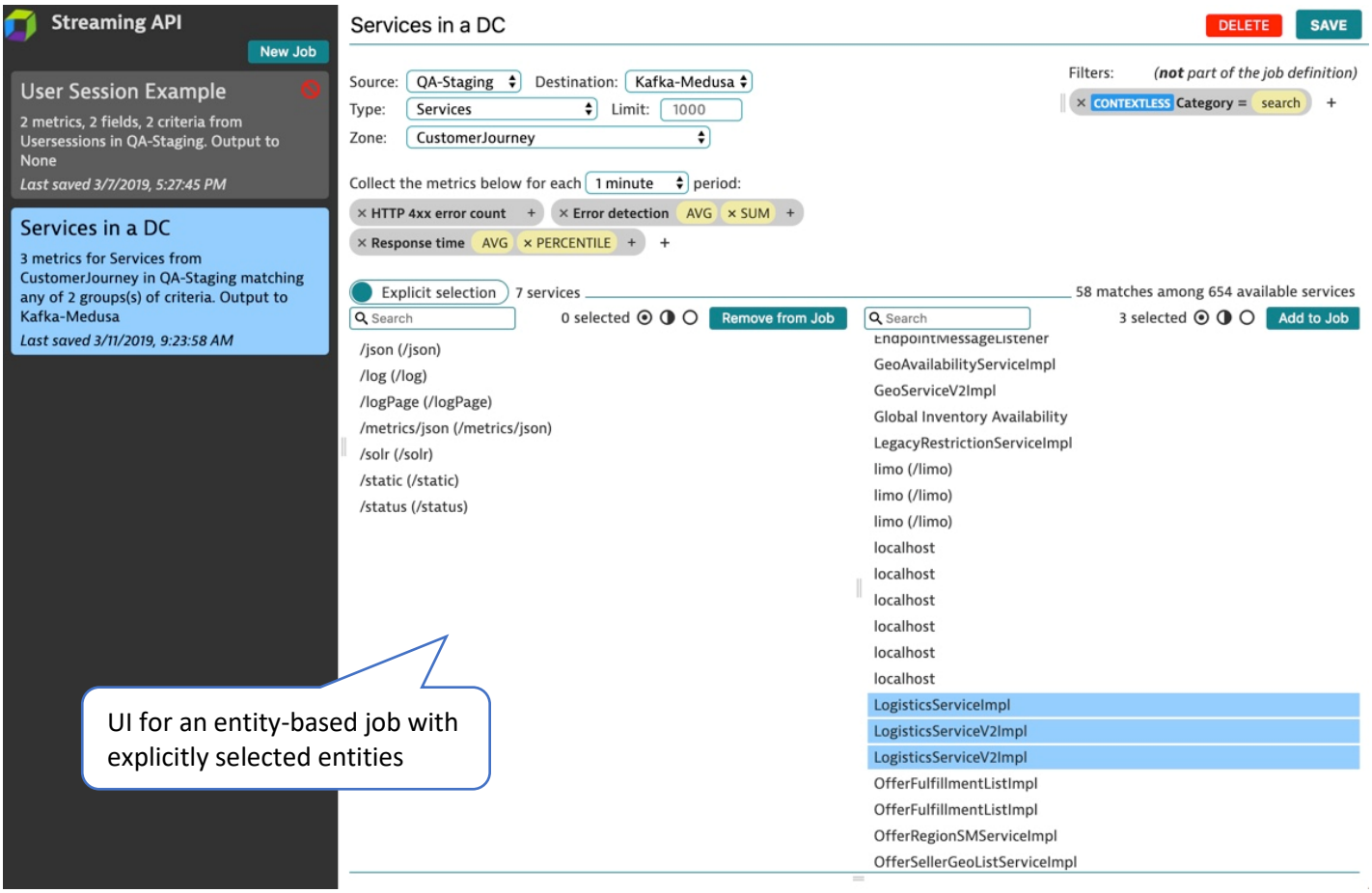
Job Editor

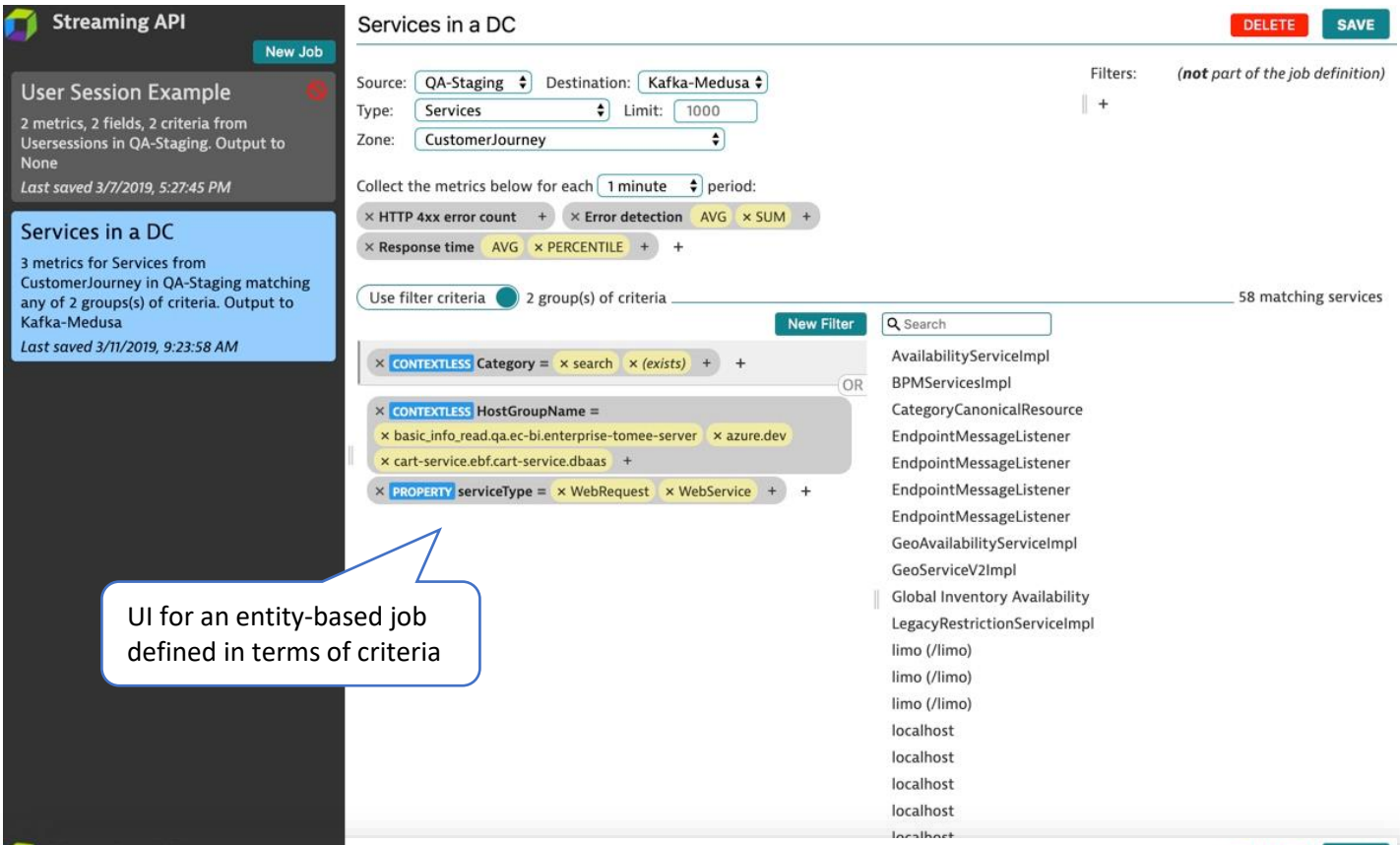
To avoid overloading Dynatrace, the job editor is designed to encourage a deliberate configuration of specific collections of entities and metrics. When a job is first created the user must select a source, destination and type (user session, host, service, etc.). For entity-based jobs the resulting set of entities available for inclusion in the job can be further constrained by Management Zone. Field and tag filters can be used to isolate the desired subset. From there, users can select all or some of the displayed entities and add them to the job, or create groups of criteria to define the contents of the job. For user session-based jobs the editor effectively enables the user to create (and test) a USQL query interactively.

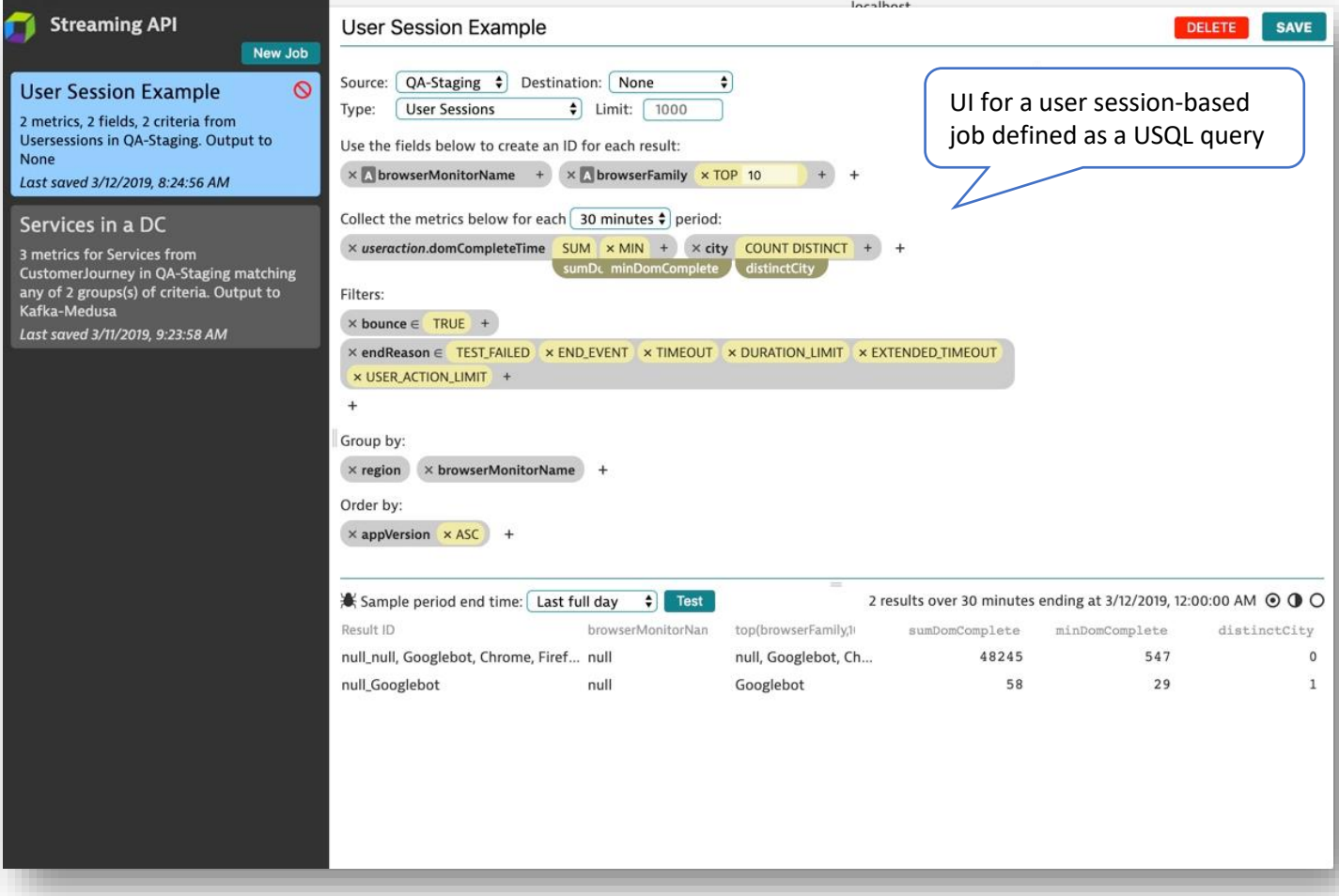
Any number of metrics can be added. By default, the average value for a metric will be collected. The user can select any of the available aggregators on each metric from a menu. Also, above the selected list of metrics is a control that allows users to configure the metric granularity for the job.

Below are the three variants of the UI for (1) entity-based jobs with explicitly selected entities, (2) entity-based jobs with a criteria-based entity selection, and (3) user-session/action-based jobs defined in terms of a USQL query.

Note that a “Test” panel is always available (expandable via a handle at the bottom of the page) at which the current job can be tested and validated. This panel can also be used for ad-hoc queries (as temporary jobs). The rows are selectable and can be copied directly into Excel or a text file for further analysis and reporting.

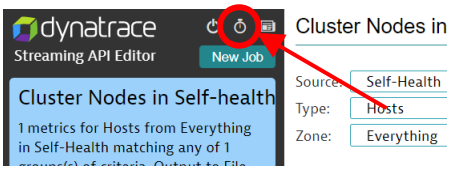


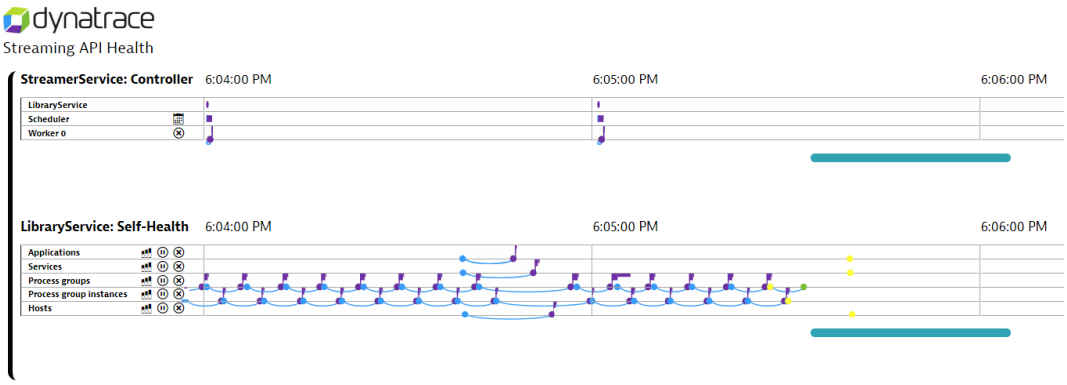

Streaming API
New Job
User Session Example
2 metrics, 2 fields, 2 criteria from Usersessions in QA-Staging. Output to None
Last saved 3/7/2019, 5:27:45 PM
Services in a DC
3 metrics for Services from CustomerJourney in QA-Staging matching any of 2 groups(s) of criteria. Output to Kafka-Medusa
Last saved 3/11/2019, 9:23:58 AM
Services in a DC
DELETE
SAVE
Source: QA-Staging
Destination: Kafka-Medusa
Type: Services
Limit: 1000
Zone: CustomerJourney
Filters:
(not part of the job definition)
Collect the metrics below for each 1 minute period:
HTTP 4xx error count
Error detection AVG SUM
Response time AVG PERCENTILE
Use filter criteria
2 group(s) of criteria
58 matching services
New Filter
Search
CONTEXTLESS Category = search (exists)
OR
CONTEXTLESS HostGroupName =
basic_info_read.qa.ec-bi.enterprise-tomee-server
azure.dev
cart-service.ebf.cart-service.dbaas
PROPERTY serviceType = WebRequest WebService
AvailabilityServiceImpl
BPMServicesImpl
CategoryCanonicalResource
EndpointMessageListener
EndpointMessageListener
EndpointMessageListener
EndpointMessageListener
GeoAvailabilityServiceImpl
GeoServiceV2Impl
Global Inventory Availability
LegacyRestrictionServiceImpl
limo (/limo)
limo (/limo)
limo (/limo)
localhost
localhost
localhost
localhost
UI for an entity-based job defined in terms of criteria


Streaming API
New Job
User Session Example
2 metrics, 2 fields, 2 criteria from Usersessions in QA-Staging. Output to None
Last saved 3/12/2019, 8:24:56 AM
Services in a DC
3 metrics for Services from CustomerJourney in QA-Staging matching any of 2 groups(s) of criteria. Output to Kafka-Medusa
Last saved 3/11/2019, 9:23:58 AM
User Session Example
DELETE
SAVE
Source: QA-Staging
Destination: None
Type: User Sessions
Limit: 1000
Use the fields below to create an ID for each result:
browserMonitorName
browserFamily TOP 10
Collect the metrics below for each 30 minutes period:
useraction.domCompleteTime SUM MIN
city COUNT DISTINCT
sumDc minDomComplete
distinctCity
Filters:
bounce TRUE
endReason TEST_FAILED END_EVENT TIMEOUT DURATION_LIMIT EXTENDED_TIMEOUT
USER_ACTION_LIMIT
Group by:
region
browserMonitorName
Order by:
appVersion ASC
Sample period end time: Last full day
Test
2 results over 30 minutes ending at 3/12/2019, 12:00:00 AM
Result ID
browserMonitorNan
top(browserFamily,1
sumDomComplete
minDomComplete
distinctCity
null_null, Googlebot, Chrome, Firef...
null
null, Googlebot, Ch...
48245
547
0
null_Googlebot
null
Googlebot
58
29
1
UI for a user session-based job defined as a USQL query

Health Monitor

To monitor the health and behavior of the StreamerService and the LibraryService process(es), a ‘Health’ page is available. It can be launched from a link in the top-left of the job editor. This page shows a continuously updated representation of key activities, based on continuous status updates emitted by the implementation.

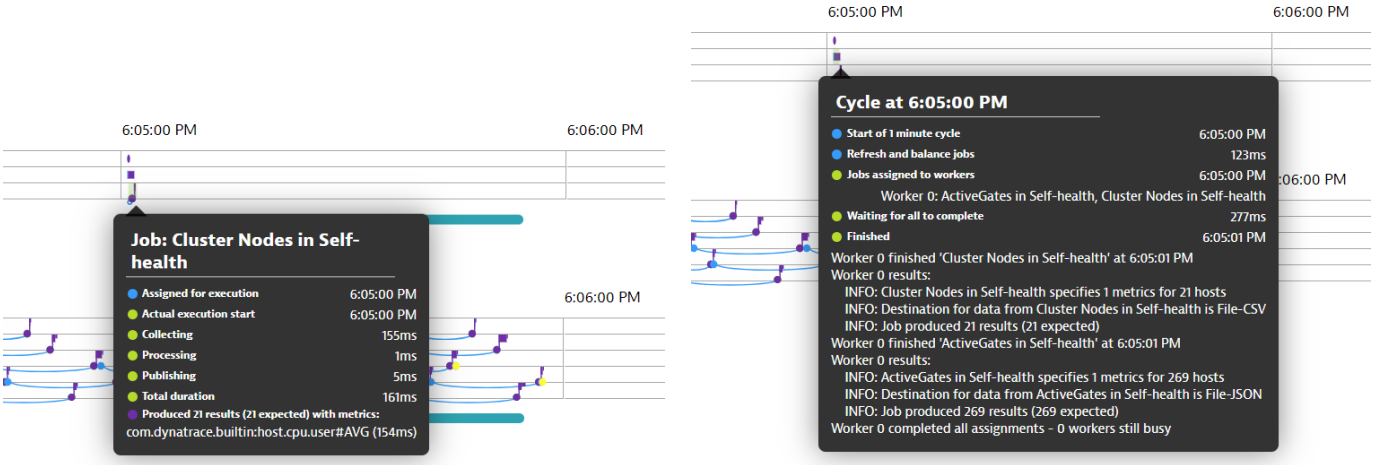


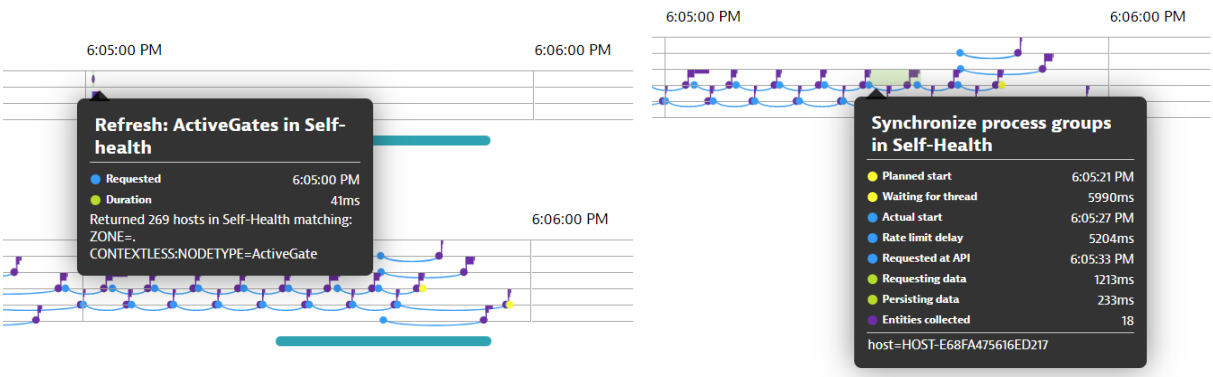


The top timeline shows the StreamerService. It hosts the Scheduler, the LibraryService instances it connects to, and all Workers in use. Click on the ‘Calendar’ icon to see and change the current job schedules.

Each marker on the timeline indicates an activity. Hovering over that activity shows more information about it.

Subsequent timelines show the various LibraryService processes, with one timeline per tenant. They show, per type of entity, the requests made to the Dynatrace API to synchronize with the cluster. Click on the ‘Statistics’ icon on each row to see more information about that synchronization pipeline’s status and effective behavior.





Documentation Hub

TODO

Using the ThrottleService

TBD

Using the ConfigService

The ConfigService is a stand-alone program that exposes REST APIs to access REST calls that are exposed through the Dynatrace UI. Such call cannot normally be made due to security constraints. This utility addresses that. It provides the following facilities:

- Establish a session under which all subsequent requests can be made to Dynatrace, at **`/login/{tenant}`**.
- Set/get the monitoring mode *for one or more* hosts at **`/config/{tenant}/HOST/monitoringMode`**.
- Set/get any monitoring setting for *one or more* services, process groups, process group instances and hosts at **`/config/{tenant}/{entityType}/monitoringSetting`**. This endpoint resolves to the following (respective) URLs in the Dynatrace UI:
    - SERVICE: "/rest/configuration/deployment/entities/SERVICE"
    - PROCESS_GROUP: "/rest/configuration/deployment/entities/PROCESS_GROUP"
    - PROCESS: "/rest/configuration/deployment/entities/PROCESS"
    - HOST: "/rest/configuration/deployment/entities/HOST"
- Execute any custom request (or series of requests) at **`/custom/{tenant}/{requestName}`**. These requests are defined in config.json (or in a file that the HTTPScripts property in config.json refers to).

Note that full documentation for these endpoints is available in the API Explorer, accessible from the StreamingAPI Editor or directly at **`http://(host:port)/doc`**.

Session management

The most critical functionality that the ConfigService provides is the ability to establish a user session against the Dynatrace UI. In the context of that session any Dynatrace REST request can be executed. To that end, the ConfigService implements the client side of Dynatrace's login protocol – but it currently only does that for Managed clusters. SaaS clusters use SAML, which the ConfigService does not yet support.

To establish a session, users call the **`/login/{tenant}`** URL first. After that, any other endpoint can be called. If the 'login' URL is not called first, the requests will automatically be performed in the context of a session established using a service account, the credentials of which can be configured in the "account" property of the appropriate tenant in config.json.

Custom request scripting

The custom requests can be configured directly in config.json, or in a separate json file. A request can be a single URL or a series of URLs to be executed in sequence. In that scenario, subsequent URLs can use responses from previous requests to configure the request parameters and (optionally) form data.

For a complete description of the options and the syntax, see the "httpscripts.json" section in the Configuration chapter.

ARCHITECTURE

Components

StreamerService

The StreamerService exposes the configuration UI. It is also responsible for kicking off a new cycle every minute, collecting the jobs for execution for each cycle, assigning them to Workers, and tracking their completions. It manages (i.e. starts, kills) the collection of Workers it needs.

Worker

The Worker is responsible for performing any jobs assigned to it every cycle. Its implementation is documented visually later in this document. It can output its data using any component that implements the DestinationAPI.

LibraryService

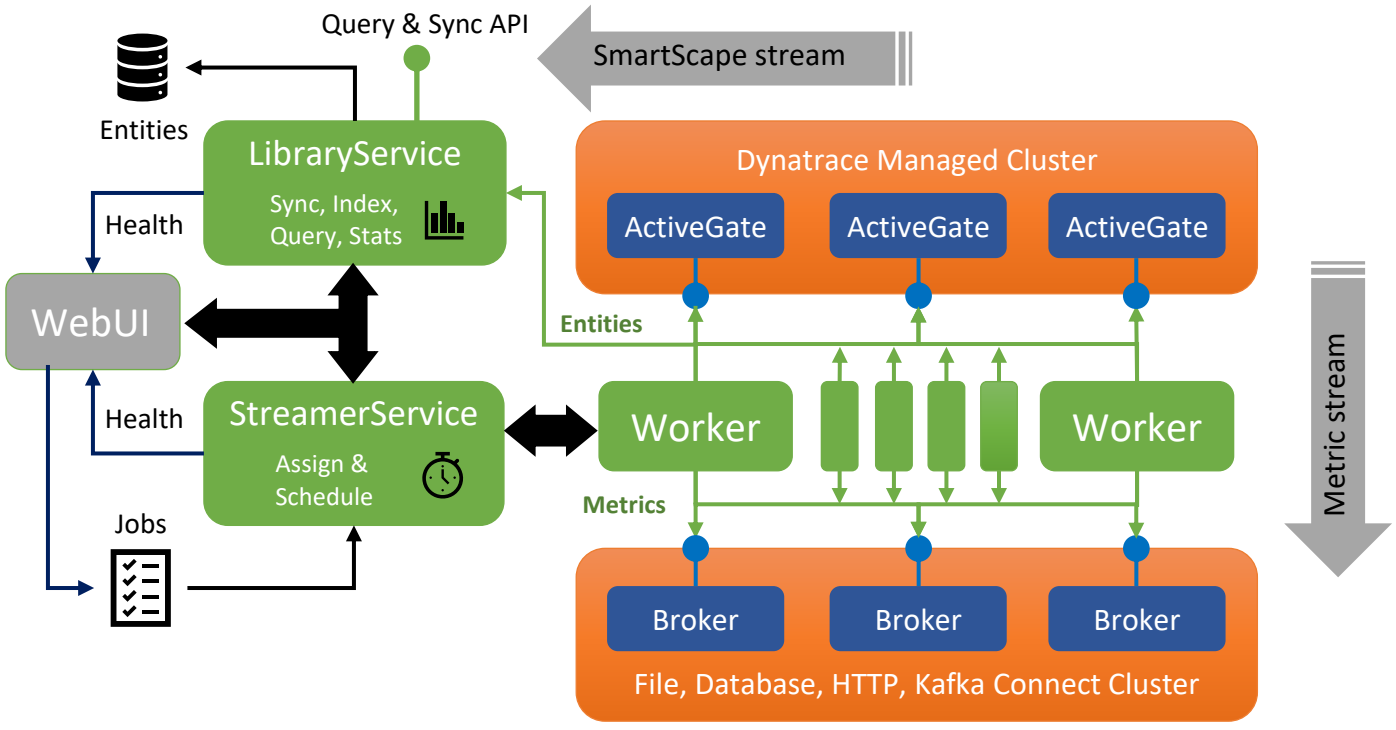
The LibraryService synchronizes with Dynatrace and exposes the LibraryService APIs at its own HTTP port. It can be embedded in the StreamerService, be started by the StreamerService as a separate process, or be started 'manually and run as a stand-alone process. Note that a swagger file is available that documents the library's interface (at `/doc`).

ThrottleService

The ThrottleService (*not shown in diagram below*) load-balances and rate-limits access to the configured Dynatrace hosts. Its main component is the DynatraceAPI class, which is also used by the LibraryService and the Worker.

ConfigService

The ConfigService (*not shown in diagram below*) is a stand-alone utility. It provides a way to log on to a Managed Dynatrace cluster and perform arbitrary REST calls that the Dynatrace UI uses as well. It can be used for automation and data extraction scenarios for which only the Dynatrace UI provides the relevant 'access points'.



Technology

The solution is implemented in NodeJS. Among the various technologies that can be used to implement the Streaming API, Python and Javascript are the most lightweight in terms of complexity of development and deployment (as opposed to Java and .NET). Among those two, NodeJS is distinctly the faster engine, which is a critical consideration for this project. Also, the minimal data processing work benefits from asynchronous processing over multithreading.

The various components in the solution (i.e. the green boxes) run as individual NodeJS instances (processes) under control of the StreamerService. Pipes are used for inter-process communication in case the StreamerService starts the LibraryService process(es); HTTP is used if the LibraryService process(es) are started independently.

Deployment

The StreamingAPI can be downloaded from https://gitlab.com/BartOostlander/dynatrace-streamingapi and requires Node.JS 10.x or higher, plus recent versions of **mariadb** or **mysql**, **kafka-node**, **express, cors, axios, winston, node-cleanup**, **cookie-parser**, **async** and **socket.io** (installable with `npm install` – see `/server/package.json`).

The solution is intended to be deployed in a separate, dedicated VM close to the Dynatrace Cluster. Endpoints for the destinations should be provisioned close (i.e. minimal hops, same network, sufficient bandwidth) as well, as the speed with which the solution pushes messages to the targeted infrastructure (e.g. a message bus) may affect the overall performance of the solution. The database should be provisioned as close as possible to the VM as well.

The StreamerService is started from the '`/server`' folder with the following command: `node StreamerService.js`. Example launch configurations for Visual Studio Code can be found in `/.vscode/launch.json`. Alternatively, PM2 may be used. An example `ecosystem.config.js` file is located in the root of the codebase. A failover instance of the StreamerService can be started as follows: `node StreamerService.js failover`.

Note that while the LibraryService may be automatically started by the streamer, it can also be started independently (using the command `node LibraryService.js`, executed from the '`server`' folder). This allows for scenarios where only the advanced query interface of the LibraryService is required, but also allows the StreamerService to run independently of the library. In the latter case it is important to configure the `config.json` file such that the StreamerService (streamer) knows not to start the library autonomously. Finally, it is possible to run the LibraryService as an embedded engine in the StreamerService.

Throughput Optimization

To maximize throughput and resilience of the solution, the retrieval and posting of metrics is performed by a variable number of Worker processes. A StreamerService process is responsible for determining and instantiating the optimal number of Workers and distribute the load evenly across them on an ongoing basis. The load is defined in terms of the job configuration: source and a destination, a set of implicitly or explicitly referenced entities (or user sessions), any number of metrics, and a preferred frequency (schedule).

Each Worker is responsible for maximizing its own throughput, based on the assigned work and the endpoints at its disposal (which it utilizes in a round-robin fashion). If it nonetheless falls behind, the StreamerService will take notice, create extra Workers, and redistribute the load. When the StreamerService fails to observe increasing throughput as it creates Worker processes, or if it reaches the maximum number of Workers, it will start to decrease the granularity with which metrics are retrieved, starting with the largest jobs in the offending schedule.

The LibraryService component is responsible for ensuring that the appropriate entities are included in each job, every time that job executes. There are two reasons for its existence: (1) the properties of each entity in a job change over time, and (2) the entities included in a criteria/filter-based job will vary as entities come into existence or get retired by Dynatrace. Querying the Dynatrace API for each job would place an impossible burden on the Cluster nodes. Additionally, the LibraryService supports criteria that the Dynatrace API does not support, like negative criteria, criteria based on properties, criteria that specify multiple options, and criteria that leverage more advanced operators.

A web application, hosted by the same NodeJS process that runs the StreamerService, exposes the UI in which a user can select entities (or criteria), user sessions, metrics and frequencies. It will save the selections in a configuration file (`/config/jobs.json`). The UI leverages the LibraryService to know which filter options are available to the user.

StreamerService

The StreamerService starts with a collection of jobs, each constituting a set of entity IDs or filter criteria, one or more metric IDs, and a desired granularity (i.e. minute-level or coarser aggregation). It starts by assigning all jobs to a single Worker process. Every minute it assigns new jobs to all Workers deemed necessary to perform the work within the allotted time) to start the next cycle. It also keeps track of Workers that have not yet sent all their "Job Done" messages for their previous cycle so that it can adjust its capacity. Note that jobs are split into smaller jobs where necessary to obtain an even spread among the available Workers.

When a Worker cannot handle the load (i.e. it has not yet published all its "Job Done" messages by the time the next cycle needs to start), it asks those Workers to finish up and retire while creating replacement Workers, plus one extra. The StreamerService then redistributes the work among the new (and now grown) collection of Workers.

If the percentage of completed work stops growing on addition of new Workers (up to a configured maximum), the StreamerService will start to decrease the granularity of individual jobs (selected in order of size). When the time to complete all work successfully falls under 30 seconds, the StreamerService will start to reduce the number of Workers. This procedure continues until the 'goldilocks' zone is found for each schedule. These tuning values are stored in the `/config/tuning.json` file so that they survive restarts. Note that any update to the `/config/jobs.json` file will reset the assigned granularities of all jobs so that previously demoted jobs are given a chance to run at their assigned schedule again.

Additionally, the "Health" monitoring page allows users to make ad-hoc adjustments to schedules.

Workers

Every Worker receives a collection of jobs from the StreamerService. The Worker executes its jobs in parallel up to a configurable limit. Upon completion of a thread it takes the next pending job and executes it. The maximum number of threads can be configured in the `/config/config.json` file. On completion of a job, the Workers sends a "Job Done" message to the StreamerService process.

LibraryService

The LibraryService contains a complete accounting of all entities in Dynatrace. It keeps itself up-to-date by scheduling a series of 'pipelines' that run in a rotating fashion, at intervals that collectively cover a complete (configurable) 'refresh cycle'. To obtain a complete set of entities, the LibraryService supports any of the following strategies:

(1) retrieve all entities of a specific type at once (including or excluding relationship info), or

(2) iterate over all provided values of a specified tag or management zone, or

(3) discover new entities by examining dependencies with already-retrieved entities and retrieving them in chunks, or

(4) don't retrieve entities of a specific type at all.

The LibraryService subsequently indexes every entity by all tags, zones, groups and stringify-able properties. The entities themselves are only kept in memory when they have been recently requested by any client of the LibraryService (like the StreamerService). Whenever it is asked for a collection of entities by a set of criteria, it 'AND's the matching index lists and retrieves the matching entities from its database (or cache). The database is always kept up-to-date with respect to what is indexed. This also enables the LibraryService to start up and 'remember' what it previously discovered and indexed.

Fault Tolerance

Data Loss Prevention

Workers work off their assigned jobs, handed to each individual Worker at each cycle. If a Worker cannot finish the work it was assigned within the allocated time, it will get three additional minutes before it is forcefully terminated, and the remainder of its metric requests discarded. This condition will result in metric value gaps for some entities in the eventual data set in the target repository.

When the collection of scheduled jobs is too big to be executed in the allotted time, additional Workers will be created. If the maximum number of Workers is reached, jobs are selected for demotion to a lower frequency (largest jobs first).

A Worker may also occasionally encounter failing requests. Where that happens, it will inform the StreamerService. This condition will be logged, and metric values may be lost.

Endpoint Failover

If a Worker has trouble connecting to an endpoint, it moves on to the next candidate. If this leads to capacity problems, the granularity will be reduced (adding Workers cannot compensate for a reduction in endpoints). It will be up to a human operator to examine and remediate the problem.

Worker Failover

The StreamerService continuously observes its Workers to see if they are still up and operational. If one dies, it will create new ones as needed. Such conditions will be logged. The availability of the StreamerService is essential as it provides the heartbeat for each cycle. While the StreamerService and the Web App run in the same process, a 'watchdog' process may be added in the future.

StreamerService Failover

The StreamerService itself may go down. While a PM2 process manager, if used, would automatically restart the StreamerService when that happens, it is also possible to launch a failover instance of the StreamerService on the same or a different machine. This failover instance periodically (every 10 seconds) checks the health of the primary instance and takes over if it fails to get an affirmative answer 5 consecutive times. When the primary instance comes back up, the failover instance will return to its 'passive' mode.

The failover instance receives its configuration and jobs from the primary instance, and is automatically kept up to date when configuration, jobs, tuning information and schedules change.

LibraryService Failover

When the StreamerService starts the LibraryService process(es) autonomously, it will automatically restart failing LibraryService processes. If LibraryService processes are started 'externally', the StreamerService will periodically attempt to check if a failing LibraryService process has been restarted yet (either manually or by a NodeJS process manager like PM2). In the meantime, it will reuse previously received answers from a failing LibraryService until it can successfully query that LibraryService again.

TODO: Run LibraryService failover instance in passive mode.

Data Duplication Prevention

Since every cycle starts on a strict 1-minute boundary and absolute timeframes (with a 1-minute delay, based on the timestamp of each cycle) are used to request data from the Dynatrace API, requesting (and processing) data twice for the same timeframe is avoided. When the implementation starts to 'stagger' requests, absolute timestamps ensure consistency of the requested timeframes.

Dynatrace API Overload Protection

The highly optimized implementation of the StreamingAPI carries the very real risk of overloading the Dynatrace API and affecting the overall performance of the Cluster nodes. To prevent this, all access to the Cluster is throttled and guaranteed not to exceed both (1) a user-configurable maximum number of requests per minute and (2) any request rate limits communicated by the Dynatrace API itself. Additionally, the LibraryService will continuously adjust its refresh cycle to stay within the configured rate limits as well as observed database access limits.

CONFIGURATION

config.json

The solution allows for the following configuration settings (in `/config/config.json`):

Structure of config.json	Description
{	
"Tenants": { … } or "pathToTenants.json",	The Dynatrace tenants/environments the StreamingAPI should connect to. Can be an embedded object or a reference to a separate file. See next sections for details.
"Templates": { … } or "pathToTemplate.json",	The templates that are available to create messages. Can be an embedded object or a reference to a separate file. See next sections for details.
"Destinations": { … } or "pathToDestinations.json",	The destinations to which the StreamingAPI may send metrics. Can be an embedded object or a reference to a separate file. See next sections for details.
"HTTPScripts": { … } or "pathToHTTPScripts.json",	Any custom HTTP scripts available to the ConfigService. Can be an embedded object or a reference to a separate file. See next sections for details.
"library": {	This entry can consist of a single object, **or an array of objects**. Each object represents a **named library instance** that handles a subset of the configured tenants.
"name": "LibraryName",	The name of the library instance.
"host": "localhost",	The host on which the instance runs.
"port": 8081,	The port at which this instance will expose its REST interface.
"launch": "auto|manual|embedded",	Specifies whether the StreamerService should automatically start a LibraryService instance (as a separate process or embedded) for this entry. If not "auto" or "embedded", the LibraryService must be manually (eg. `node LibraryService.js LibraryName`).
"offline": false,	This setting is for debugging purposes. If 'true', the LibraryService will not synchronize with Dynatrace through the DynatraceA APi. This may be useful for debugging.
"debug": false,	This setting is for debugging purposes. If 'true', LibraryService instances launched by the StreamerService will attach to the Visual Studio Code debugger.
"tenants": ["qa", "prod_asda"],	A list of tenants (keys from "Tenants", above) that should be handled by this library instance.
"refreshRate": "_hour",	The preferred timespan the StreamingAPI is allowed to spend on refreshing all entities.
"timeToPurge": "_2hours",	The timespan after "last seen by" entities are deleted from the StreamingAPI's admin.
"cachePeriod": "_15mins"	The timespan for keeping entities in memory since they have been last used in a query.
"savePeriod" : "_6hours",	The timespan for persisting the entity and index administration. Upon exit, the library will persist all indexes before terminating. This setting however allows that to happen periodically as well, while the library runs. Mote that the only purpose of saving the indexes is to allow the library to start up faster (i.e. avoid initializing from the database).
"retryLimit": 1	The number of times a Dynatrace API call should be retried before it fails. Serves as a default for the per-tenant setting with the same name. Also limits the number of retries to the database, per transaction.
},	
"database": {	Configuration of the database in which the StreamingAPI stores the entities it indexes.
"class": "(ClassName)",	Name of a Javascript class that implements the DatabaseAPI (i.e. a database adaptor). Currently supported are MariaDBAdaptor and MySQLAdaptor. You can write our own classes for other databases.
"host": "localhost",	
"port": 3309,	

`"user": "root",` `"password": "",`	Configuration options used by the database adaptor. Generally, this consists of the host, port and account at which the database can be accessed. There is no constraint on what configuration options are allowed here.
`"database": "StreamingAPI",`	Name of the database in which the StreamingAPI will create its "Entities" table.
`"connectionLimit": 5`	Maximum number of open connections to the database.
`"connectionTimeout": 10000`	Timeout before the attempt to connect to the database fails.
`},`	
`"failover": {`	(Optional) This section configures the contract between the primary StreamerService instance and the failover instance. Failover instances can be started with a `config.json` file that contains only this entry (or use the command line argument `config=myfile.config`). It will receive its configuration from the primary instance. Note that the "`logging`" entry must be in the initial config file for the failover instance in order to get any logging.
`"primary": "localhost:8080",`	This is the address at which the primary instance listens.
`"standby": "127.0.0.1",`	This is the IP address from where the failover instance will contact the primary instance to check for its health.
`"secret": "12345"`	This is the key that the failover instance must provide in order to receive any updated configuration information from the primary instance. Only if both the standby IP address and the secret match will this information be included in the 'health' response.
`},`	
`"reqRateLimit": 600,`	Maximum number of requests per minute the StreamingAPI is allowed to make at each host at which the DynatraceAPI can be accessed.
`"requestLimit": 20,`	Maximum number of concurrently running requests per host.
`"maxQueueTime": 10,`	Maximum number of seconds a request may be queued when the request rate limit is reached.
`"retryLimit": 1,`	The number of times a Dynatrace API call should be retried before it fails. Serves as a default for the per-tenant setting with the same name. Also limits the number of retries to the database, per transaction.
`"maxWorkers": 8,`	Maximum number of worker processes before the StreamingAPI starts to reduce the granularity of jobs.
`"maxThreads": 8,`	Maximum number of parallel jobs being processed in a Worker.
`"maxJobSize": 10000,`	Maximum number of entities in a job before the StreamingAPI starts to split a job into smaller jobs.
`"streamerPort": 8080,`	The port at which the StreamingAPI exposes its UI.
`"debugWorker": false,`	Specifies whether Workers should connect to the Visual Studio Code debugger.
`"logging": [ "file", "console" ],`	Specifies where the logging output should go. Running under PM2, the console output is captured for viewing automatically and no file logging is required. When "file" is included, log files are stored in the "`../log`" directory. To run in silent mode, omit this entry or specify an empty array.
`"jobLatency": 2,`	The number of minutes the StreamingAPI waits before it retrieves the metrics for a particular interval.
`"libTimeout": 60000`	The timeout after which an attempt to connect to a LibraryService instance fails.
`"allowedUsers": [ "userID1", "userID2" ]`	If specified, the list of user IDs that are allowed to use the StreamingAPI editor.
`}`	

TODO: Additional timeout options. – move to the right places.

tenants.json

The 'tenants.json' file is an alternative to defining your tenants in 'config.json'. Tenants refer to environments or tenants in your Dynatrace cluster(s). Any number of tenants can be defined. They are used by the LibraryService(s) to know from where to extract SmartScape data, and by the Workers to know from where to obtain metric values.

Structure of tenants.json	Description
{	The Dynatrace tenants/environments the StreamingAPI should connect to.
"<TenantName>": {	Symbolic name of the tenants. Will be used by the ConfigService and the ThrottleService as well as by the LibraryService API endpoints.
"host": "mycluster.dynatrace-managed.com",	A single host or an array of hosts at which the StreamingAPI can access the Dynatrace APIs.
"url": "/e/xxxx-xxx-xxx-xxx-xxxxxx",	The tenant/environment ID.
"token": "(DynatraceToken)",	The token that should be used when requesting data from the Dynatrace APIs.
"account": "serviceuser|servicepwd",	The user ID and password of a service account that the configuration API should use. Note that the configuration API actually logs on to Dynatrace as an end-user to make configuration changes. This setting is also used when the tenant is a SaaS tenant. In that scenario, the StreamingAPI is not able to log in to Dynatrace as an actual user, so to validate user credentials when a user logs in to the Editor, these are the credentials that provide that user access.
"retryLimit": 5,	The number of times a Dynatrace API call should be retried before it fails.
"pageSize": 1000,	The maximum number of entities the library is allowed to retrieve per request against the Dynatrace API. If there is more data, the StreamingAPI will retrieve the data in groups of 'pageSize' entities.
"chunking": {	(Optional) Retrieval strategies for the various types of entities. APPLICATIONS and SERVICES are retrieved by default in single requests. All other entities are by default retrieved as they are discovered by following other entities' relationships. This configuration allows users to override this per type of entity.
"<Type>": "<criterion>=a,b,c",	Available strategies ('HOST' is an example here): • "HOST": "[ENVIRONMENT]Cloud=cloud1,cloud2,cloud3" - Use individual requests to retrieve hosts, one for each tag value in the list. • "HOST": "ZONE=mz1,mz2,mz3" - Use individual requests to retrieve hosts, one for each management zone in the list. • "HOST": "HOSTGROUP=group1,group2,group3" - Use individual requests to retrieve hosts, one for each host group in the list. • "HOST": "ALL" – Retrieve all hosts at once, including relationships. • "HOST": "LIST" – Retrieve all hosts at once, excluding relationships. • "HOST": "NONE" – Don't load hosts at all.
…	
}	
},	
"<TenantName>": { … },	
…	
},	

TODO: More discussion on chunking and timeouts and so on. Also – do we have everything here?

destinations.json

The ‘destinations.json’ file is an alternative to defining all destinations in ‘config.json’. Destinations refer to the Javascript class and configuration parameters that describe where Workers need to send the output of a job to.

Structure of destinations.json	Description
`{`	
`"Kafka-Medusa": {`	Symbolic name of the destination.
`"class": "KafkaBrokerAPI",`	The name of the class to be used to connect to the destination’s endpoint. Currently supported values are “KafkaBrokerAPI”, “CSVRendererAPI” and “JSONRendererAPI”.
`"host": [`	One or more hosts at which the destination can be reached, plus the port to be used to connect to any of them. TODO: These options are whatever the destination adaptor need or can use.
`"kafka-213355567.mycompany.com",`	
`"kafka-213355561.mycompany.com",`	
`"kafka-213355552.mycompany.com"`	
`],`	
`"port": 9092,`	
`"address": "tf.medusa.metric.1",`	For the KafkaBrokerAPI this represents the topic at which the data should be published. For other adapters this might represent something else.
`"defaults": {`	(Optional) In addition to the built-in template macros documented below, you may create your own macros and the values that they should return when used in templates. The values can be complex objects, in which case you can use the `[default.subprop.subsubprop]` notation to select the value. This also implies that you can add your own default `tag` and `prop` properties.
`"env": "QA",`	
`"dc": "CDC"`	
`},`	
`"template": "DynatraceMDOM"`	The name of the template (from the “Templates” section) that specifies the format of the message before it is sent to the destination.
`},`	
`"File-CSV": {`	A second destination. You can define as many destinations as required.
`"class": "CSVRendererAPI",`	The “class” attribute specifies the name of the Javascript file that exports a class with the same name. That class must implement the DestinationAPI. Note that each class may require specific configuration settings. Their values should be specified here. For the “CSVRendererAPI” and “JSONRendererAPI” classes, ‘address’ contains the name of the file to write to. A future enhancement may support the “http” protocol to post messages to a Web service.
`"address": "$timeStamp/$jobName.json",`	
`"template": "ATabularFormat",`	
`"delim": ","`	
`},`	
`…`	
`}`	

TODO Discussion on scalability and performance of the adaptors.

httpscripts.json

The 'httpscripts.json' file is an alternative to defining your HTTP scripts in 'config.json'. HTTP scripts are series of parameterized REST calls to be made by the ConfigService to the Dynatrace UI.

Structure of httpscripts.json	Description
`{`	The scripts (or simple requests) that the ConfigService exposes.
`"<RequestName>": {`	Symbolic name of the request. The value is a "request" object with the properties specified below.
`"url": "/a/url/into/dynatrace ",`	A URL into Dynatrace. An environment URL (i.e. `"/e/xxxx-xxx-xxx-xxx-xxxxxx"`) is automatically prepended to it.
`"description": "(DynatraceToken)",`	(Optional) A description that will be shown in the API Explorer (in which this request may be tested).
`"method": "POST|PUT|GET|DELETE",`	(Optional) The HTTP method to be used. The default is "GET" unless the 'data' property is defined – then it is "POST".
`"params": { "age": "[request.years]", "greeting": "[response.message]", "fullName": "[request.name]" }`	An arbitrary object that specifies the query parameters for the request. Each value may contain one or more macros and/or literals. Macros may refer to 'request' (taken from any query parameter provided by the client/user or created in a previous request) or to 'response' (taken from any previous response – '.'-separated paths into a response object are allowed). Not that the use of '|' characters are allowed to provide for alternative sources for the information to be collected. For more information on macros, see the section "templates.json" (next).
`"data": { … },`	(Optional) An arbitrary object of data to be posted. The data will be posted as JSON. Alternative encodings can be supported if necessary.
`"next": { … }`	(Optional) The next request to be invoked. The value is again a "request" object with the properties specified above.
`},`	
`"<RequestName>": "/a/url/into/dynatrace",`	The alternative to the above is to specify a simple URL into Dynatrace. It may contain query parameters. Whatever query parameters are provided by a client/user when this request is invoked will be added to it. Parameters with the name replace any defaults specified here.
…	Define any number of named request scripts.
`}`	

Example

```
{
  "scripttest": {
    "url": "/test/first",
    "description": "Provide your age ('a') and
                    a password ('pass').",
      "params": {
        "age":  "[request.a]",
        "name": "Bart"
      },
      "next": {
        "url": "/test/second",
        "params": {
          "years": "[request.age]",
          "name":  "[response.userName]",
          "pwd":   "[request.pass]"
        },
        "next": {
          "url": "/test/third",
          "params": {
            "age":      "[request.years]",
            "greeting": "[response.message]",
            "fullName": "[response.name.full]"
          }
        }
      }
    }
  }
}
```

templates.json

The ‘templates.json’ file is an alternative to defining all templates in ‘config.json’. Templates describe how to create messages from a job’s entities and their respective metrics.

Structure of templates.json	Description
{	
"MyFirstTemplate": {	Name of the template.
"_metadata_": ["col1","col2", …]	(Optional) Name of file containing metadata. This can be YAML or a JSON file.
"_columns_": "KafkaBrokerAPI",	(Optional) Array of column names.
"default": { … },	Default template for entities of a type not explicitly defined.
"HOST": { … },	Template for entities of type ‘HOST’. Note that it is not mandatory to have a template for HOSTS, specifically. You can define templates for any type(s) (e.g. APPLICATION, SERVICE, PROCESS_GROUP, PROCESS_GROUP_INSTANCE, HOST) for which you need a specifically structured output.
…	
},	
"MySecondTemplate": { … },	You can define as many templates as you need.
…	
}	

A template for a specific type of entity (as well as the “default” template) describes a simple or complex object with properties and fields. This is an example of a simple object template:

```
"default": {
    "format": "[job.type] message",
    "id": "[prop.entityId]",
    "name": "[prop.displayName]",
    "*1": "[tag.ENVIRONMENT.*]",
    "*2": "[tag.CONTEXTLESS*]",
    "*3": "[metric.*]"
}
```

Values can be strings (e.g. `“[job.type] message”`), or objects with properties, or arrays. String values can (and typically do) contain macros, encapsulated in square brackets (e.g. ‘`[a.macro]`’). In the output, macros are replaced with the value(s) found at the path they reference. Macros follow the following syntax: `[scope.path.to.value]`.

- A value can contain multiple macros (e.g. `“This is a [job.type] named [prop.displayName]”`). When that is the case, the output is always a string. If an individual macro resolves to an object or array, it will be stringified before replacing the macro.
- A value can contain macro alternatives, separated by a ‘|’ character (e.g. `“This [job.type] is a [tag.CONTEXTLESS.TagThatNormallyExists]|[tag.CONTEXTLESS.OtherwiseThisTag]|(N/A)”`).
- A value can be surrounded by ‘[‘ and ‘]’ characters. This affects into what data type the value will be resolved. More on that later. A value can also be an object itself, with properties and macro-ed values.

Using Scopes

The rules for how macros resolve depend on the "scope" (i.e. '`prop`', '`tag`', '`metric`', '`job`' or '`default`').

- **'prop'** signals that the '`path.to.value`' part will be a '.'-separated path to an entity's property. Examples are '`prop.displayName`' and '`prop.softwareTechnologies.0.type`'. The Dynatrace API Explorer describes all properties that entities of various types can have. Some entities have properties that are complex objects or arrays. Specific values can be 'harvested' from them by describing the path to the desired field. When arrays are in play, you can use numbers to refer to a value at a particular index.
- **'tag'** signals that that the '`path.to.value`' part is a '.'-separated path to a specific entity tag. Tags use the following naming convention: "`CONTEXT.tagName`". Commonly available contexts are "`CONTEXTLESS`", "`ENVIRONMENTS`", "`K8S`", etc.
- **'metric'** signals that the '`path.to.value`' part is a metric ID. Metric IDs look like '.'-separated paths but are in fact 'just' IDs. A metric ID is always followed by a '#' character and an aggregation mnemonic (i.e. AVG. MAX, MIN, SUM). For metrics that have dimensions, an additional '#' may follow, describing how to aggregate the values across all dimensions (i.e. AVG. MAX, MIN, SUM, NUM). The default value for such scenarios is "AVG".
- **'job'** signals that the '`path.to.value`' part is a job property. Available properties are:
  - "**name**": Name of the job.
  - "**source**": Origin of the entities. This is the name of one of the tenants defined in config.json.
  - "**destination**": The destination of the entities (as defined in config.json or destinations.json).
  - "**zone**": The management zone used to constrain the set of entities selected in this job.
  - "**TYPE**": The original uppercase type of the entities selected in the job.
  - "**type**": That same type in a lowercase form, with the underscores replaced by spaces.
  - "**granularity**": The granularity of the metrics in the output. Valid values are "_min", "_5mins", etc.
  - "**timestamp**": The epoch value (ms) of the end of the timeframe for which the metrics have been collected.
  - "**timestring**": The string representation of the timestamp in simplified extended ISO format (YYYY-MM-DDTHH:mm:ss.sssZ). The time zone is always zero UTC offset, as denoted by the suffix "Z".
- **'default'** signals that the '`path.to.value`' part refers to a value in the 'defaults' section in the destination (see config.json). The values can be complex objects because you can use the '[default.path.to.value]' notation to select the desired property inside it. Note that this scope serves two use cases:
  - To include data in the output that is not present in the job or entities themselves;
  - To provide defaults when properties or tags are missing in an entity (e.g. "`[tag.CONTEXTLESS.mytag]|[default.mytag]`").
- If the scope is missing, the macro is presumed to refer to one of the predefined 'globals':
  - "**format**": The name of the template used to produce the message.
  - "**tenant**": The name of the tenant as configured in the "Tenants" section. Is an alias for [job.source].
  - "**timestamp**": Is an alias for [job.timestamp].
  - "**timestring**": Is an alias for [job.timestring].

Using Wildcards

In all cases, the 'path.to.value' part can contain a '*' somewhere in the path (only one is allowed). Examples are: '`metrics.*`', '`prop.softwareTechnologies.*.type`', '`tag.CONTEXTLESS.*`', etc. A '*' matches any property, index, tag or metric at that point in the path. The macro gets replaced with **all matching values**. The resulting output depends on a number of factors. Consider the following template:

```
"default": {
    "myprop": "[mymacro]"
}
```

- If the property name is a 'fixed' name (i.e. "myprop") and the macro resolves to *one or more strings*, the output can be a comma-separated list of strings, or an array of strings, depending on whether the value is specified as an array or as a string:
  - `"myprop": "[prop.softwareTechnologies.*.type]"` *(value specified as string)*
    `"myprop": "DotNet,Java"`
  - `"myprop": [ "[prop.softwareTechnologies.*.type]" ]` *(value specified as array)*
    `"myprop": [ "DotNet", "Java" ]`
- If the property name is a 'fixed' name and the macro resolves to *one or more objects*, the output will be an object or an array of objects (never a string). Examples:
  - `"myprop": "[prop.softwareTechnologies.0]"` *(value specified as string)*
    `"myprop": { "type": "DotNet" , "version": "5.1" }`
  - `"myprop": [ "[prop.softwareTechnologies.0]" ]` *(value specified as array)*
    `"myprop": [ { "type": "DotNet" , "version": "5.1" } ]`
  - `"myprop": "[prop.softwareTechnologies.*]"` *(value specified as string)*
    `"myprop": [ { "type": "DotNet" , "version": "5.1" }, { "type": "Java" , "version": "8" } ]`
  - `"myprop": [ "[prop.softwareTechnologies.*]" ]` *(value specified as array)*
    `"myprop": [ { "type": "DotNet" , "version": "5.1" }, { "type": "Java" , "version": "8" } ]`
- When a macro will resolve to an object or an array, the property name can also contain a '*'. The '*' property will be replaced with the actual property names found in the resulting object or array (note: *therefore, using a '*' for macros that resolve to a string would be meaningless*). The replacement then goes as follows:
  - `"*": "[prop.softwareTechnologies.0]"` *(macro resolves to an object)*
    `"type": "DotNet", "version": "5.1"`
  - `"*": "[prop.softwareTechnologies.0.*]"` *(macro resolves to an object's properties)*
    `"type": "DotNet", "version": "5.1"`
  - `"*": "[prop.softwareTechnologies.*]"` *(macro resolves to an array of objects)*
    `"0": { "type": "DotNet", "version": "5.1" },`
    `"1": { "type": "Java",   "version": "8"   }`
  - `"*": "[prop.softwareTechnologies.*.type]"` *(macro resolves to an array of strings)*
    `"0": "DotNet",`
    `"1": "Java"`

- `"*": "[tag.CONTEXTLESS.*]"` *(resolves to a tag object's properties)*

  ```
  "MyFirstTag": "SomeValue",
  "MySecondTag": "AnotherValue"
  ```

- `"*": [ "[tag.CONTEXTLESS.*]" ]` *(resolves to a tag object's properties)*

  (Don't know what this will or should do yet)

- `"*": "[metric.*]"` *(resolves to a metric object's properties)*

  ```
  "metric1": "1.234",
  "metric2": "2.345"
  ```

- `"*": [ "[metric.*]" ]` *(resolves to a metric object's properties)*

  (Don't know what this will or should do yet)

- Property names with a '*' may have a prefix or a postfix, so that multiple '*'-properties can be used in the same containing object. Prefixes will be placed in front of each found property name (e.g. `"ABC_*": "[tag.CONTEXTLESS.*]"` creates properties `"ABC_MyFirstTag"` and `"ABC_MySecondTag"`). Postfixes will be omitted – they only exist to enable multiple '*' properties that each will expand into sets of actual properties.

Mappings

Macro's can also contain mappings. This is supported for metrics only. Mappings provide a way to translate metrics collected for an entity to the corresponding names, data types, values, or any other piece of metadata available for a given metric.

- The "_metadata_" property of the template refers to a user-defined JSON or YAML file that contains any number of properties for all metrics you want to be able to 'map'. Example:

```
metrics:
    - time-series-id: com.dynatrace.builtin:service.responsetime
      dimension: SERVICE
      planum-metric-name: responseTime
      data-type: java.lang.Double
      namespace: metric.application.apm.server
    - time-series-id: com.dynatrace.builtin:service.requestspermin
      dimension: SERVICE
      planum-metric-name: throughput
      data-type: java.lang.Double
      namespace: metric.application.apm.server
    - time-series-id: com.dynatrace.builtin:service.failurerate
      dimension: SERVICE
      planum-metric-name: errorCount
      data-type: java.lang.Double
      namespace: metric.application.apm.server
```

- The '**time-series-id**' field is mandatory and will be used to index the entries. Everything else is up to you. Note that in addition to whatever metadata fields are defined in the metadata file, the following mappings are always available: "**id**" (the original metric's ID), "**name**" (Dynatrace's human-readable name for the metric) and "**value**" (the value of the metric).

- Whenever a mapping is used where the macro contains a '*', the macro resolves to a list of 'primitive' values, one for each match. The usual rules apply for resolving an array of strings/numbers/Booleans into an output. You can therefore use the metadata to map metrics as follows:

  - `"name": "[metric.com.dynatrace.builtin:service.responsetime=>planum-metric-name]"`
    `"name": "responseTime"`

  - `"names": "[metric.*=>planum-metric-name]"` *(use user-defined 'planum-metric-name')*
    `"names": "responseTime,errorCount"` *(assume that there are two metrics)*

  - `"names": [ "[metric.*=>planum-metric-name]" ]`
    `"names": [ "responseTime", "errorCount" ]`

  - `"values": [ "[metric.*=>value]" ]` *(use always-available 'value' mapping)*
    `"values": [ 1.234, 0 ]`

- You can also use the metadata to map property names as follows:

  - `"*": "[metric.*]"` *(no mapping)*
    `"com.dynatrace.builtin:service.responsetime": 1.234,`
    `"com.dynatrace.builtin:service.failurerate": 0`

  - `"*=>planum-metric-name": "[metric.*]"` *(with mapping)*
    `"responseTime": 1.234,`
    `"errorCount": 0`

Referencing Related Entities

We have seen that macros follow the following syntax: `[scope.path.to.value]`. However, a variant on this syntax is supported as well: `[scope.ENTITY_TYPE.path.to.value]`. This is only available for jobs that contain flow or stack relationships, where each entity to be output is (typically) related to other entities. Macros that contain an 'ENTITY_TYPE' work as follows:

- For stack-based jobs where for each entity we have its stack dependencies, the macro will find all related entities of the given type and resolve the macro for each. The result will therefore be an array, and the usual rules apply for resolving an array into an output:
    - `"myprop": "[prop.PROCESS_GROUP.softwareTechnologies.0.type]"` *(assume the entity is a HOST)*

      `"myprop": "ASP_NET, WCF"`
    - `"myprop": [ "[prop.PROCESS_GROUP.softwareTechnologies.*]" ]` *(assume the entity is a HOST)*

```
"myprop": [
     [ { "type": "ASP_NET", "edition": ".NET", "version": "4.7.2.0" },
       { "type": "Java",    "edition": "8",    "version": "1.8"     }
     ],
     [ { "type": "WCF",     "edition": null,   "version": "4.7.2.0"    },
       { "type": "MSSQL",   "edition": null,   "version": "4.7.3260.0" }
     ]
 ]
```

  It is possible to force the macro to resolve to an object instead of an array, and use the usual rules:

    - `"myprop": { "*": "[prop.PROCESS_GROUP.softwareTechnologies.0.type]" }` *(value in '{…}')*

      `"myprop": { "PROCESS_GROUP-1234": "ASP_NET", "PROCESS_GROUP-5678": "WCF" }`
    - `"*": { "*": "[prop.PROCESS_GROUP.softwareTechnologies.0.type]" }` *(property name is '*', value in '{…}')*

```
"PROCESS_GROUP-1234": "ASP_NET",
"PROCESS_GROUP-5678": " WCF"
```

    - `"*": { "*": "[prop.PROCESS_GROUP.softwareTechnologies.*]" }` *(property name is '*', value in '{…}')*

```
"PROCESS_GROUP-1234": [
     { "type": "ASP_NET", "edition": ".NET", "version": "4.7.2.0" },
     { "type": "Java",    "edition": "8",    "version": "1.8"     }
 ],
"PROCESS_GROUP-5678": [
     { "type": "WCF",     "edition": null,   "version": "4.7.2.0"    },
     { "type": "MSSQL",   "edition": null,   "version": "4.7.3260.0" }
 ]
```

    - `"*=>displayName": "[prop.PROCESS_GROUP.softwareTechnologies.*]"` *(name is mapped!)*

      (Note: result is similar to the previous due to the rules re. wildcarded properties!)

```
"mysite.company.com": [
     { "type": "ASP_NET", "edition": ".NET", "version": "4.7.2.0" },
     { "type": "Java",    "edition": "8",    "version": "1.8"     }
 ],
"app-backend": [
     { "type": "WCF",     "edition": null,   "version": "4.7.2.0"    },
     { "type": "MSSQL",   "edition": null,   "version": "4.7.3260.0" }
 ]
```

- For flow-based jobs, where each entity is related to one or more entities *of the same type*, you can only refer to that type. The macro will find all related entities and resolve the macro for each. There may not be a good use case for this, but it is supported.
- Note that using 'ENTITY_TYPE' is not supported for macros that reference metrics.

Context Switches

In addition to the use of 'ENTITY_TYPE' in macros (e.g. "[scope.ENTITY_TYPE.path.to.value]"), it is also supported in 'context switches', like this: "<ENTITY_TYPE>…[macro]…".

- Context switches are surrounded by '<' and '>' characters. They specify that any macros following the 'switch' should be resolved against each individual related entity of the specified type. Thus, using a 'switch' immediately duplicates the value into as many values as there are matching related entities.
- You can have multiple context switches in a value. Each starts at the current context and multiplies from there. Thus, for a host with two process groups and two related services in each, you can do this:
  - "myprop": "[prop.displayName] running <PROCESS_GROUP>[prop.displayName] doing <SERVICE>[prop.displayName]"

    ```
    "myprop": "this.host.name running mysite.company.com doing orderService,this.host.name running
    mysite.company.com doing confirmService,this.host.name running app-backend doing
    checkCredit,this.host.name running app-backend doing storeOrder"
    ```

  - "myprop": ["[prop.displayName] running <PROCESS_GROUP>[prop.displayName] doing <SERVICE>[prop.displayName]"]

    ```
    "myprop": [
        "this.host.name running mysite.company.com doing orderService",
        "this.host.name running mysite.company.com doing confirmService",
        "this.host.name running app-backend doing checkCredit",
        "this.host.name running app-backend doing storeOrder"
    ]
    ```

- When a value that uses context switches is assigned to a wildcarded property name, the result adds properties to the containing object. The property names are constructed by concatenating the IDs of each context branch (using a ':' character). The IDs can be mapped if so desired.
  - "*": "… running <PROCESS_GROUP>[prop.displayName] doing <SERVICE>[prop.displayName]"

    ```
    "PROCESS_GROUP-1234:SERVICE-7890": "… running mysite.company.com doing orderService",
    "PROCESS_GROUP-1234:SERVICE-7891": "… running mysite.company.com doing confirmService",
    "PROCESS_GROUP-5678:SERVICE-1230": "… running app-backend doing checkCredit",
    "PROCESS_GROUP-5678:SERVICE-1231": "… running app-backend doing storeOrder"
    ```

  - "*=>displayName": "… <PROCESS_GROUP>[prop.displayName] doing <SERVICE>[prop.displayName]"

    ```
    "mysite.company.com:orderService": "… mysite.company.com doing orderService",
    "mysite.company.com:confirmService": "… mysite.company.com doing confirmService",
    "app-backend:checkCredit": "… app-backend doing checkCredit",
    "app-backend:storeOrder": "… app-backend doing storeOrder"
    ```

- For flow-based jobs, where each entity is related to one or more entities of the same type, in a potentially long chain, each context switch goes one level deeper than the previous one.
- Note that macros that refer to metrics are always resolved against the 'original' entity. Tags and properties however will honor the context switches.

Column Order

Finally, a template can also contain a "_columns"_ property. This property is useful for destination adaptors that create tabular outputs. The '_columns_' property provides a means to control the order in which the fields are to be written to the output. Consider this example:

```
"MyTableFormat": {
    "_columns_": ["id", "name", "format", "CONTEXTLESS.*", "ENVIRONMENT.*", "*" ],
    "default": {
        "format": "[job.type] message",
        "id": "[prop.entityId]",
        "name": "[prop.displayName]",
        "*1": "[tag.ENVIRONMENT.*]",
        "*2": "[tag.CONTEXTLESS*]",
        "*3": "[metric.*]"
    }
}
```

Whenever a message is produced, the Destination adaptor receives a list of columns names that reflects the order specified in the "_columns" property. The Worker (specifically, the FormatterFactory) will track all fields added to the output across all messages that are produced as part of the job. This means that the destination adaptor can use a single, consistent set of columns, even if some messages did not receive (values for) some of the referenced fields.

The column names may contain '*' characters. Fields with matching prefixes are matched against the best matching column names and inserted there in a consistent order. It is recommended to always include a '*' name at the end to catch any fields that did not match any of the other wildcarded column names.

Note that the 'ledger' that is used internally to track all fields is a complex (i.e. hierarchical) object. Thus, non-tabular destinations may also leverage this ledger. For column-based outputs the template should ideally only contain one level –the ledger will then contain a simple list of column names.

Formal Template Specifications

Grammar

The specifications below describe the grammar of the template language. Note that templates must be valid JSON objects.

Color Code Legend			Syntax Character Legend	
Black	Syntax character	→	:=	Is defined as
Blue	Literal character		|	Or
Orange	Symbol, syntax of which is defined elsewhere		…	Any number of the preceding
Green	Comment		()	Group
			?	Zero or one
			*	Zero or more

```
template:= "<literal>": {                   // The name of the template.
     "_metadata_": "<pathToFile>",          // Optional – used for mappings of metrics.
     "_columns_":  [ "<columnName>", … ],   // Optional – used to control order of fields.
     "default":    <objectValue>,           // Mandatory if none specified below.
     "<type>":     <objectValue>, …         // Templates for specific entity types.
}

pathToFile:=  [\w\/]*?[\w\.]*\.(json|yml)   // Relative to working directory (/server).
literal:=     [^"{}|<>\[\]]*                // 0+ chars except ", {, }, |, <, >, [ and ].
```

```
// This column name syntax is only for the '_columns_' property of a template.
columnName:=  <prefix>.*    // A prefix is a partial prop chain, typically a tag context.
columnName:=  *             // Should always be the last element in the '_columns_' array.
columnName:=  <literal>     // Is always an actual property name as found in the output.

type:= APPLICATION | SERVICE | PROCESS_GROUP | PROCESS_GROUP_INSTANCE | HOST

objectValue:= {
     "<propertyName>(=><mapping>)?": <propertyValue>,
     …
}

mapping:=         String    // The name of a property in the metadata, or 'id', 'name' or 'value'.

// Note: '*' is only allowed if propertyValue resolves to object or array.
propertyName:=    <literal>?*<literal>?     // prefix or postfix allowed.
propertyName:=    <literal>

propertyValue:=   <objectValue>
propertyValue:=   "<resolvableValue>"
propertyValue:=   [ <propertyValue>, … ]    // Note: nested arrays in templates are not allowed.

resolvableValue:= <literal><macro><resolvableValue>

// A macro may be preceded by a (single) context switch and may be followed by one or more alternatives.
macro:=           <contextSwitch>?[<macroString>](|[<macroString>])*

contextSwitch:=   <<type>>

macroString:=     format | tenant | timestamp | timestring
macroString:=     prop.<propChain>
macroString:=     tag.<tagChain>
macroString:=     metric.<metricName>(=><mapping>)?
macroString:=     job.<propChain>
macroString:=     default.<propChain>

propChain:=   <property>(.<propChain>)?
propChain:=   <index>(.<propChain>)?
propChain:=   *             // Only if '*' not yet encountered in propChain.

property:=    String        // The name of an object property.
index:=       Number        // The position of an array element.

tagChain:=    <context>.<tag>

context:=     CONTEXTLESS | ENVIRONMENT | K8S | (other known tag context)
context:=     *

tag:=         String        // The name of a tag.
tag:=         *             // Only allowed if the context is not '*'.

metricName:=  [a-zA-Z.:]+[#](<aggregator>)([#](<aggregator>))?
metricName:=  *

aggregator:=  AVG | MAX | MIN | SUM | COUNT
```

Resolving values

For each entity, an output object is produced from an applicable template. The structure of a template has been described above. At the heart of the template engine lies the concept of `resolvableValue`. It is defined as follows (in simplified notation, with more advanced constructs removed for clarity – more on those later):

```
resolvableValue:= <literal><macro><resolvableValue>
```

This means that a `resolvableValue` is a string that contains any mix of static text and macros. A `macro` refers to a property, tag, metric or other piece of information that should be retrieved from the entity at hand or from somewhere else. It is possible to specify alternative sources for the desired information by listing several macros, separated by '|' characters. After a value is found, the other alternatives are discarded.

```
macro:= <contextSwitch>?[<macroString>](|[<macroString>])*
```

The context switch is useful for changing the current context (i.e. entity) to another, related entity. This affects from where '`prop.`' and '`tag.`' macros are resolved (but not '`metric.`' macros!).

The resulting information can be a primitive value like a string, number or boolean, a compound value like an object or an array. The result of resolving a `resolvableValue` depends on the following:

1. If a `macroString` has no wildcard (i.e. no '*') and thus refers to a specific value, the result is that specific value (e.g. a string, number, boolean, object or array).
   - **Metrics only:** If a `macroString` also has a mapping, the metric's ID is used to map to a specific field in the metadata (or to 'id', 'name' or 'value'). The result is whatever the metadata contains for that field.
2. If a `macroString` has a wildcard, the result is a collection of values that may end up being rendered as a comma-separated string, an object or an array.
   - **Metrics only:** If a `macroString` also has a mapping, the metrics' IDs are used to map to a specific field in the metadata (or to 'id', 'name' or 'value'). The result is a collection of whatever the metadata contains for that field.
3. If a `macroString` contains one or more context switches (e.g. `<PROCESS_GROUP>[prop.displayName]`), the result is a collection of values that may end up being rendered as a comma-separated string, an object or an array.
   - Each time a context switch is encountered in a 'still-resolving' `resolvableValue`, the `resolvableValue`, as resolved up till that point, is immediately duplicated into as many copies as related entities of the specified type are found, starting from the current context (which is initially the entity at hand – the one that contains the metrics).
   - This means that a `resolvableValue` that has one or more context switches resolves to a collection of resolved values, one for each distinct branch of context switches.
4. If a `macroString` refers to a related entity type (e.g. `[prop.PROCESS_GROUP.displayName]`), the result is a collection of values (one for each related entity of the specified type) that may end up being rendered as a comma-separated string, an object or an array.
   - This notation does not change the context away from the current entity and has no multiplying effect beyond the `macroString` itself.
5. If a `resolvableValue` contains more than one `macroString` or has static text in it as well (e.g. `[tag.CONTEXTLESS.*]_tags`), the result is always converted into a string and replaced as such into the 'still-resolving' value.

Resolving objects

For each entity, an output object is produced from an applicable template. The structure and syntax of a template has been described above. The relevant part for this section is this:

```
objectValue:=   {
     “<propertyName>(=><mapping>)?”: <propertyValue>,
     …
}

propertyName:=  <literal>?*<literal>?
propertyName:=  <literal>

propertyValue:= <objectValue>
propertyValue:= “<resolvableValue>”
propertyValue:= { “<resolvableValue>” }
propertyValue:= [ <propertyValue>, … ]
```

The top-level element of a ‘default’ or type-specific template is an `objectValue`. The formatter generates a message for a given entity (and its metrics) by starting with the creation of an empty output object that corresponds to that top-level `objectValue`.

For each property listed an `objectValue` it creates one or more corresponding properties in the output object.

1. If the `propertyName` is a literal value (e.g. “`myprop`”), a single property with that name is created in the output object and its value is whatever `propertyValue` resolved to.
2. If the `propertyName` contains a wildcard (e.g. “`*`” or “`k8s_*`”), the properties will be obtained from whatever the `propertyValue` resolves to.
   - If the `propertyValue` resolves to an object, the properties in that object (and their values) are copied to the current output object.
   - If the `propertyValue` resolves to a collection, the indexes are used as property names and copied into the current output object (with their respective values).
   - If the `propertyValue` resolves to a collection *because of references to other entities*, the entity IDs that identify each branch or match are used as property names (e.g. `PROCESS_GROUP-1234:SERVICE-4567`) and copied into the current output object (with their respective values).
   - If the `propertyValue` resolves to a primitive value (i.e. a string, number, boolean), the `propertyName` is kept as-is.
   - If a prefix is specified, each copied property is prefixed with it. Postfixes are ignored – they are merely supported in order to allow for multiple wildcarded properties (multiple “`*`” properties are invalid in JSON).
3. If the `propertyName` contains a wildcard and a mapping, the property names will be obtained by mapping their values, using the field specified in the mapping, to the corresponding metadata.
   - For entityIds, available mapping fields are any string-property available for one or all of the corresponding entity types (typically `displayName`).

The resolved `propertyValue` is assigned to the `propertyName` as follows:

4. If the `propertyValue` is an `objectValue` itself (e.g. "`{ … }`"), that embedded object is resolved recursively according to the usual rules. This of course creates a nested output object.
5. If the `propertyValue` resolves to a string, that string becomes the value of the specified `propertyName`. Note that a wildcard in the `propertyName` would not make sense because there is nothing to derive a name from.
6. If the `propertyValue` resolves to an object, either (if no '*' in play) that object is assigned to the `propertyName`, OR (if it contains a '*') its properties are copied into the object that contains the `propertyName`.
7. If the `propertyValue` resolves to an array, either (if no '*' is in play) that array is assigned to the `propertyName`, OR (if it contains a '*') its indexes are copied into the object that contains the `propertyName`.
8. If the `propertyValue` is embedded in an array (e.g. `[ "prop.technologies.*", "anotherElement ]`), its resolution will be inserted in that array at the position of that `propertyValue`. An object will be inserted as an object. An array will be 'spliced' into the array.
9. If the `propertyValue` contains a macro that references related entities of a specific type (e.g. `prop.PROCESS_GROUP.displayName` or "`<PROCESS_GROUP>[prop.displayName]`"), then the result is a collection of resolutions, one for each matching entity (or distinct chain of entities if multiple context switchers are used). That collection can be assigned to the `propertyName` as follows:
   - If the `propertyValue` is expressed as a string (i.e. "`prop.PROCESS_GROUP.displayName`"), the collection resolves into a single, comma-separated string of stringified values.
   - If the `propertyValue` is embedded in an array (e.g. `[ "prop.PROCESS_GROUP.displayName" ]`), the collection resolves as individual elements in that array.
   - If the `propertyValue` is embedded in an object (e.g. `{ "*": "prop.PROCESS_GROUP.displayName"}`), the collection resolves as individual properties in that object.
   - The usual rules for translating wildcards into the property names apply (see above).

Note that rule (6) works recursively upwards. That has the following implication:

This `objectValue` in a template:

```
"myprop": {
    "name": "[prop.displayName]",
    "*": {
       "app": "Orders",
       "*": "[tag.CONTEXTLESS*]"
    }
}
```

does **not** resolve into:

```
"myprop": {
    "name": "my.host.name",
     "*": { // Wildcard here!
        "app": "Orders",
        "tag1": "value1",
        "tag2": "value2",
        "tag3": "value3"
    }
}
```

but rather, into:

```
"myprop": {
    "name": "my.host.name",
    "app": "Orders",
    "tag1": "value1",
    "tag2": "value2",
    "tag3": "value3"
}
```

TODO: diagrams for value resolution. Separately, for object construction.

EXTENSIBILITY

DatabaseAPI

The DatabaseAPI is an interface that should be implemented by a NodeJS module when there is no available implementation for the selected database (MariaDB and MySQL are supported out of the box). It is responsible for handling the interactions with a database. For more details see the MariaDB and MySQL adaptors. Also refer to the JSDoc-generated documentation at /doc/www/index.html.

Below is a boilerplate implementation.

```
class MyDatabaseAdaptor {

    constructor(config, cleaner, logger) {
        // Connect to the database using the options specified in the 'config' object. Initialize a connection pool.
    }

    createConnection(taskName, resolve, reject) {
         // Return an object that implements the DatabaseConnection interface:
        return {
            name:    taskName,
            resolve: resolve,
            reject:  reject,
            state:   "Queued",
            conn:    null,      // Recommended. All methods after 'initialize()' must work on this actual connection.
            initialize:   function (conn) {
                // This is the 'connection.initialize()' method call by 'acquire()'. Set 'this.state' to "Connected".
            },

            // The methods below only need to exist after 'initialize()' is called, must an act on the 'conn' object.
            begin: function () {
                // Start a new transaction. The 'commit' or 'rollback' method will be called as appropriate.
            },
            query: function (statement, values) {
                // Return a promise to execute the statement, using the values if specified.
            },
            stream: function (statement, values) {
                // Execute the statement, using the values if specified, and stream the result back.
            },
            commit: function (onDone, onError) {
                // Commit transaction and release the connection. Call onDone() or onError(err) when done.
            },
            rollback: function (onDone, onError) {
                // Rollback transaction and release the connection. Call onDone() or onError(err) when done.
            },
            release: function (onReleased, noCleanup) {
                // Release connection. Call 'cleanup()' if noCleanup not 'true'. Call onReleased() when done.
            }
        };
    }

    acquire(connection, onAcquired) {
        /*  Set 'this.state' to "Connecting". Create a connection ('conn') from the pool. Once available, call
            connection.initialize(conn). After that, call connection.resolve(connection) and, if specified,
            call onAcquired(). If a connection cannot be created, call connection.cleanup(c) and connection.reject(err).
         */
    }

    get activeConnections() {
        // Return the number of currently active connections.
    }

    get totalConnections() {
        // Return the maximum number of available connections.
    }

    close(resolve) {
        // Close the connection pool. Call resolve() afterwards.
    }
}
module.exports = MyDatabaseAdaptor;
```

DestinationAPI

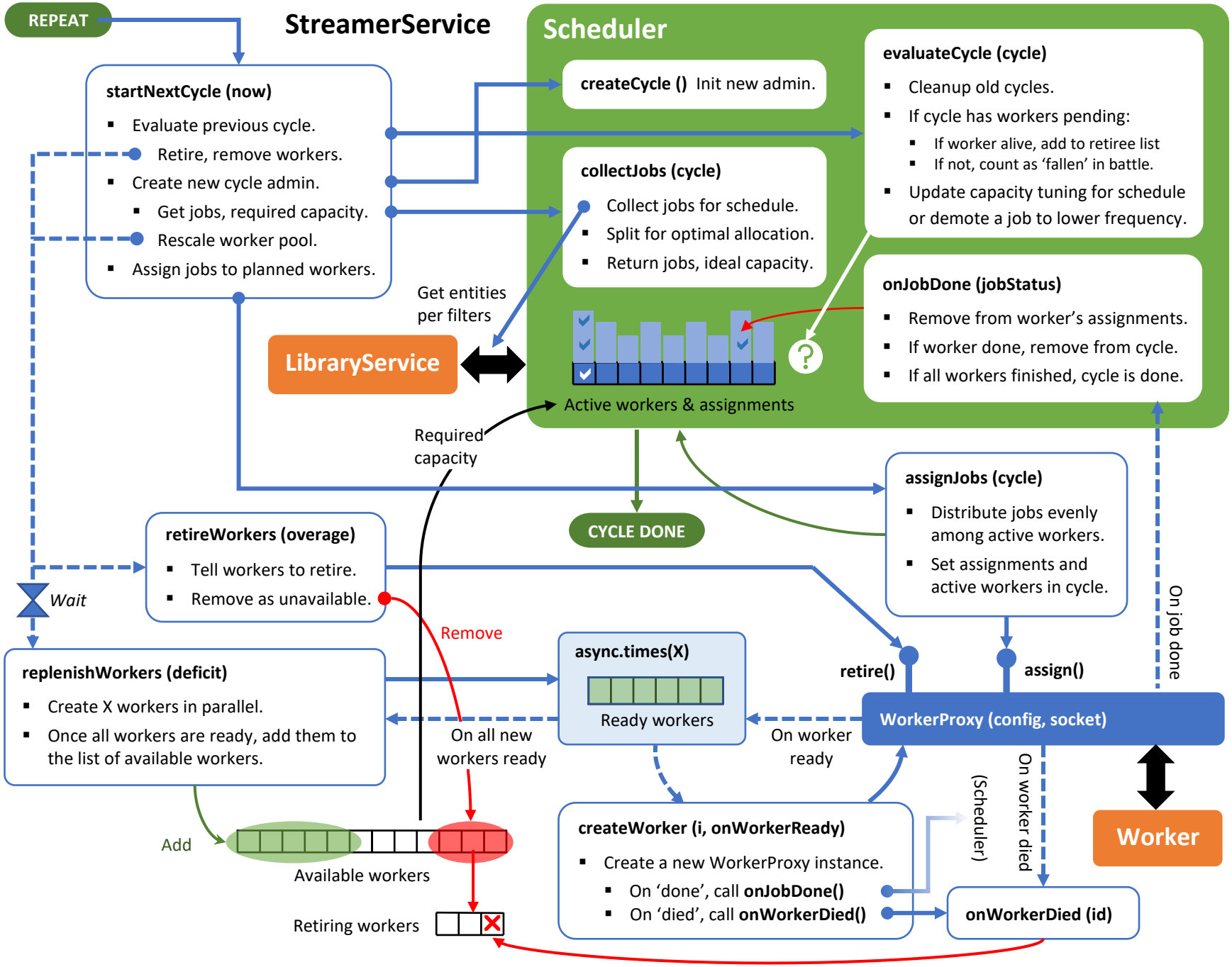
The DestinationAPI is an interface that should be implemented by a NodeJS module when there is no implementation available for the selected output (CSV-file, JSON-file, HTTP and Kafka are supported out of the box). It is responsible for handling the communication with the targeted destination technology. Each implementation should return a *singleton instance from its constructor* that implements the DestinationAPI. Also refer to the JSDoc-generated documentation at /doc/www/index.html.

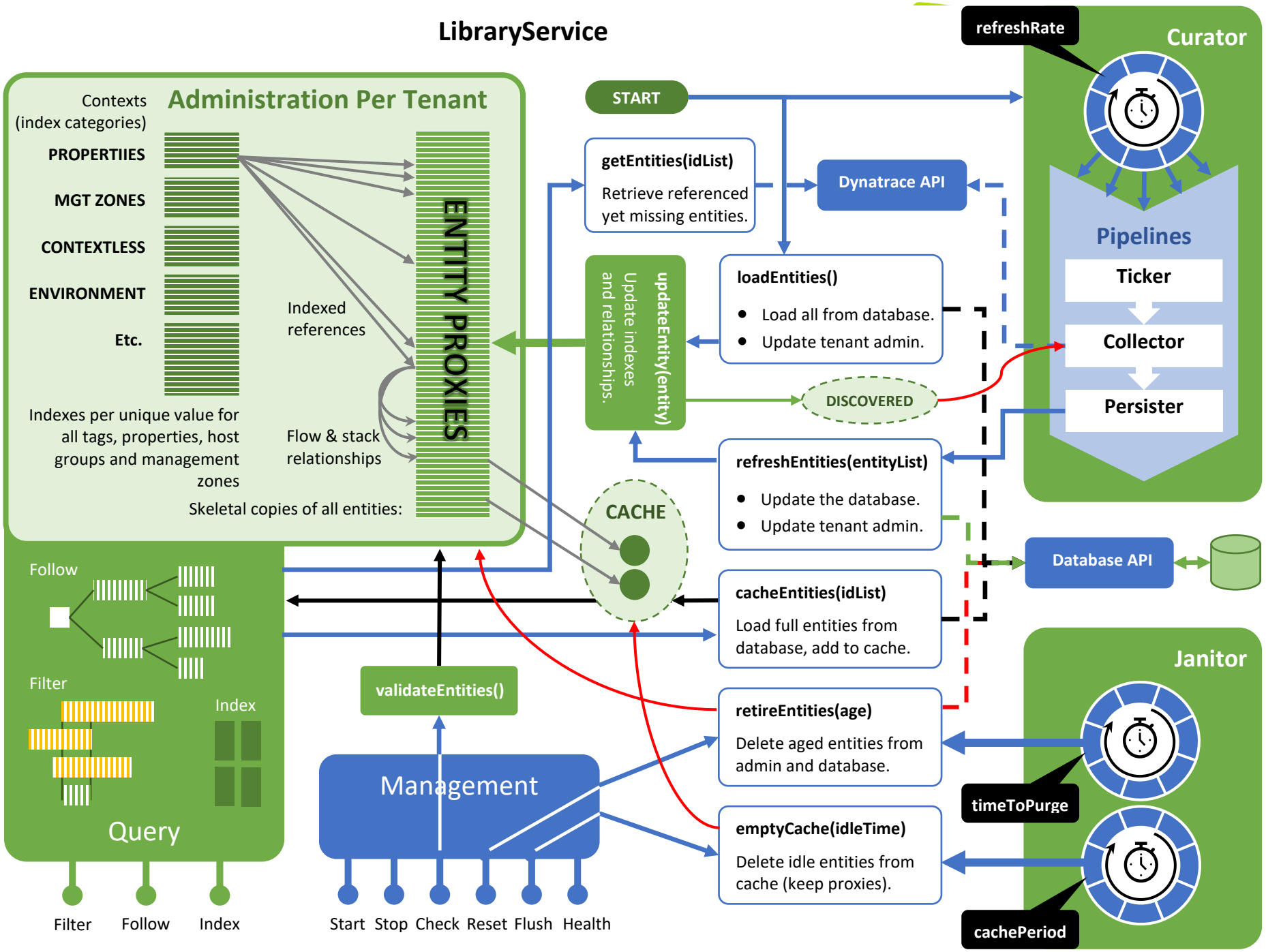
Below is a boilerplate implementation.

```
const MyDestinationAPI = function (config) {
    this.config = {
        produce:   config.produce, // The provided value is the 'ask'. Translate it into what this adaptor can support.
        // Other, implementation-specific settings. The config comes straight from the 'destination' in config.json.
    };

    const subscribe = function (eventName, fn) {
        // Add the subscription for this event.
    };

    const emit = function (eventName, data) {
        // Call the subscribed event handlers.
    };

    // Initialize. When ready emit a "ready" (or "error") event (asynchronously!).
    setImmediate(emit, "ready");

    // Return an object that implements the DestinationConnection interface.
    return {
        on:        subscribe,              // Accept subscriptions to "ready" and "error" events.
        produce:   self.config.produce,    // This is the output this adaptor can produce for graphs.
        formatter: config.formatter,       // The formatter as provided in the 'config' object.
        open:      function (name, cycle, fieldGetter, onConnectionOpen) {

            // Open a connection to the destination and call onComplete() when done (asynchronously!).
            setImmediate(() => onConnectionOpen(null, {

                // Send/write messages and call onComplete() when done (asynchronously!).
                send:  function (messages, onComplete) { … },

                // Close whatever connections were opened.
                close: function () { … }
            }));
        }
    };
};

module.exports = MyDestinationAPI;
```

ARCHITECTURAL DIAGRAMS

The following pages illustrate the internal architecture of the main components of the StreamingAPI.


REPEAT
StreamerService
Scheduler
startNextCycle (now)
Evaluate previous cycle.
Retire, remove workers.
Create new cycle admin.
Get jobs, required capacity.
Rescale worker pool.
Assign jobs to planned workers.
createCycle () Init new admin.
collectJobs (cycle)
Collect jobs for schedule.
Split for optimal allocation.
Return jobs, ideal capacity.
evaluateCycle (cycle)
Cleanup old cycles.
If cycle has workers pending:
If worker alive, add to retiree list
If not, count as 'fallen' in battle.
Update capacity tuning for schedule or demote a job to lower frequency.
onJobDone (jobStatus)
Remove from worker's assignments.
If worker done, remove from cycle.
If all workers finished, cycle is done.
Get entities per filters
LibraryService
Active workers & assignments
Required capacity
CYCLE DONE
assignJobs (cycle)
Distribute jobs evenly among active workers.
Set assignments and active workers in cycle.
retireWorkers (overage)
Tell workers to retire.
Remove as unavailable.
Wait
Remove
replenishWorkers (deficit)
Create X workers in parallel.
Once all workers are ready, add them to the list of available workers.
async.times(X)
Ready workers
retire()
assign()
On job done
WorkerProxy (config, socket)
On all new workers ready
On worker ready
Add
Available workers
Retiring workers
createWorker (i, onWorkerReady)
Create a new WorkerProxy instance.
On 'done', call onJobDone()
On 'died', call onWorkerDied()
(Scheduler)
On worker died
onWorkerDied (id)
Worker

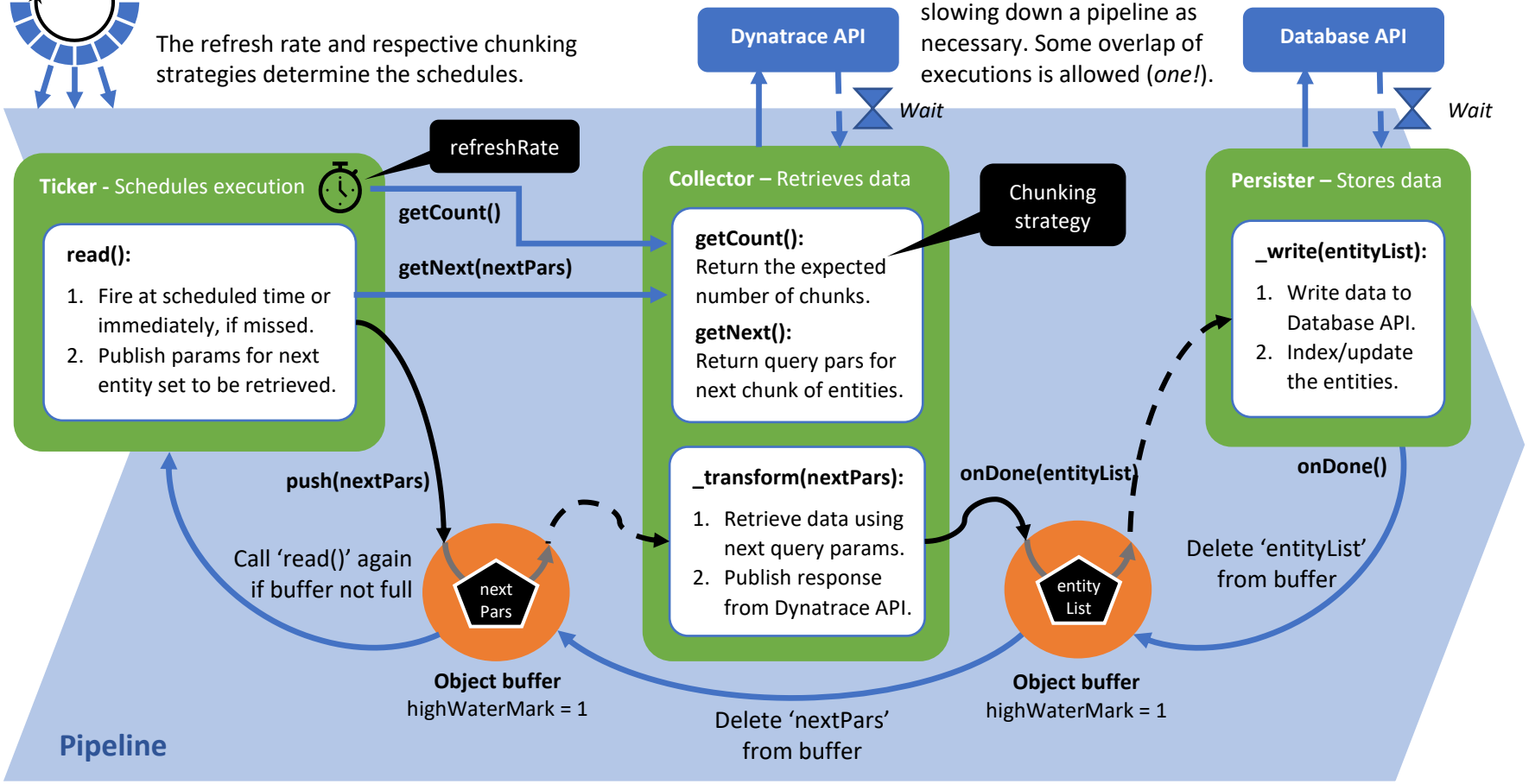

LibraryService
Administration Per Tenant
Contexts
(index categories)
PROPERTIIES
MGT ZONES
CONTEXTLESS
ENVIRONMENT
Etc.
Indexed
references
ENTITY PROXIES
Indexes per unique value for
all tags, properties, host
groups and management
zones
Flow & stack
relationships
Skeletal copies of all entities:
START
getEntities(idList)
Retrieve referenced
yet missing entities.
Dynatrace API
updateEntity(entity)
Update indexes
and relationships.
loadEntities()
Load all from database.
Update tenant admin.
DISCOVERED
refreshEntities(entityList)
Update the database.
Update tenant admin.
CACHE
cacheEntities(idList)
Load full entities from
database, add to cache.
validateEntities()
retireEntities(age)
Delete aged entities from
admin and database.
emptyCache(idleTime)
Delete idle entities from
cache (keep proxies).
Management
Start
Stop
Check
Reset
Flush
Health
Follow
Filter
Index
Query
Filter
Follow
Index
refreshRate
Curator
Pipelines
Ticker
Collector
Persister
Database API
Janitor
timeToPurge
cachePeriod

Curator

Create a pipeline for each tenant and entity type.

The refresh rate and respective chunking strategies determine the schedules.

The capacity and speed limits of the Dynatrace API and the Database API create backpressure, slowing down a pipeline as necessary. Some overlap of executions is allowed (*one!*).

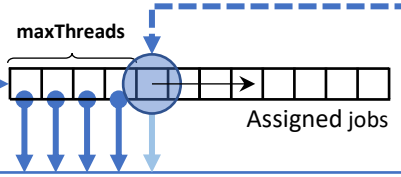


Worker

ASSIGN

On Assignment of Jobs

- Place the assigned jobs in a queue.
- Call **performJob(job, cycle)** for up to **maxThreads** jobs.



While jobs in queue

performJob (job, cycle)

- Create job log for this cycle.
- Create a requestor per metric, bound to the job and the request data for that metric: **collectMetricForEntities(job, requestData, onComplete)**
- Call all requestors in parallel:
  - Add any errors that occur to the log and ignore them.
  - When all requestors have reported their metric data, call **onMetricsCollected(job, allMetricData)**

collectMetricForEntities (job, requestData, addMetricData)

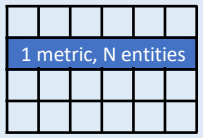
- Issue a request to the source API for the specified metric.
  - On receipt of data, call **addMetricData(metricData)**

Dynatrace API (metricRequest)

On receipt of data

addMetricData(metricData)

async.parallel(…)



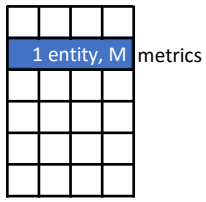
allMetricData
M metrics x N entities

When all data received (or error)

onMetricsCollected (job, allMetricData)

- ⓘ Each element in the **allMetricData** array contains the values of *a single metric for all requested entities (N)*.
- ⓘ The target API needs an array where each element contains the values of *all requested metrics (M) for a single entity*.
- Let **allEntityData** be mapped from each entity, as follows:
  - Create a new message for the entity.
  - For each metric in the job:
    - Get the metric from the **allMetricData** matrix.
    - Place the value in the message.
- Send **allEntityData** to the appropriate destination API.
  - On completion, call **onJobProcessed(job)**
- Trigger the next job in the queue, if available.

Message Formatter



allEntityData
N entities x M metrics

Destination API (allEntityData)

On delivery of data

onJobProcessed (job)

- Collect the times for each step.
- Send the job log to the StreamerService.
- If asked to retire and all jobs are

?

COMPLETED

RETIRE

TERMINATED

Dynatrace API

REQUEST

Tenant, endpoint

On Request (no queueing)

- Round-robin over tenant's hosts.
- First host to accept request gets it.
- If all reject: rate exceeded (429).

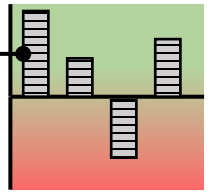
OR

On Request (with queueing)

- Find host with best availability.
- Tell best host to accept the request.
- If none available, put in global queue.

429

Host Availabilities



Bucket capacity

Max waiting list size

RESULT

Response stream or Javascript object

accept()

availability()

accept(request, forced)

- If forced to accept, append request to local queue.
- Refill bucket. If bucket is empty or too many requests are running, return true (if forced) or false (rejected).
- Take a request from top of queue, take a droplet from the bucket, add the request to 'issued' list, and issue the request to an endpoint.
- Schedule next request present in global or local queue.

Get availability

- If local queue full, return "unavailable" (−∞).
- If local queue not empty, return negative list size.
- If issued requests, return a portion of bucket fill.
- Return fullness of bucket (fuller is better).

IDLE

YES

Queues empty ?

NO

dripRate

Take 1 drip

Add request to 'issued' list.

issueRequest(request)

On response

Bucket empty or running list full

? YES NO

ACCEPT

Queued

Issued

? NO Error YES

Forced to accept ? NO YES

ACCEPT

dripRate

Add 1 drip

Bucket capacity = max req/min

Remove request from 'issued' list.

? NO Retry YES

REJECT

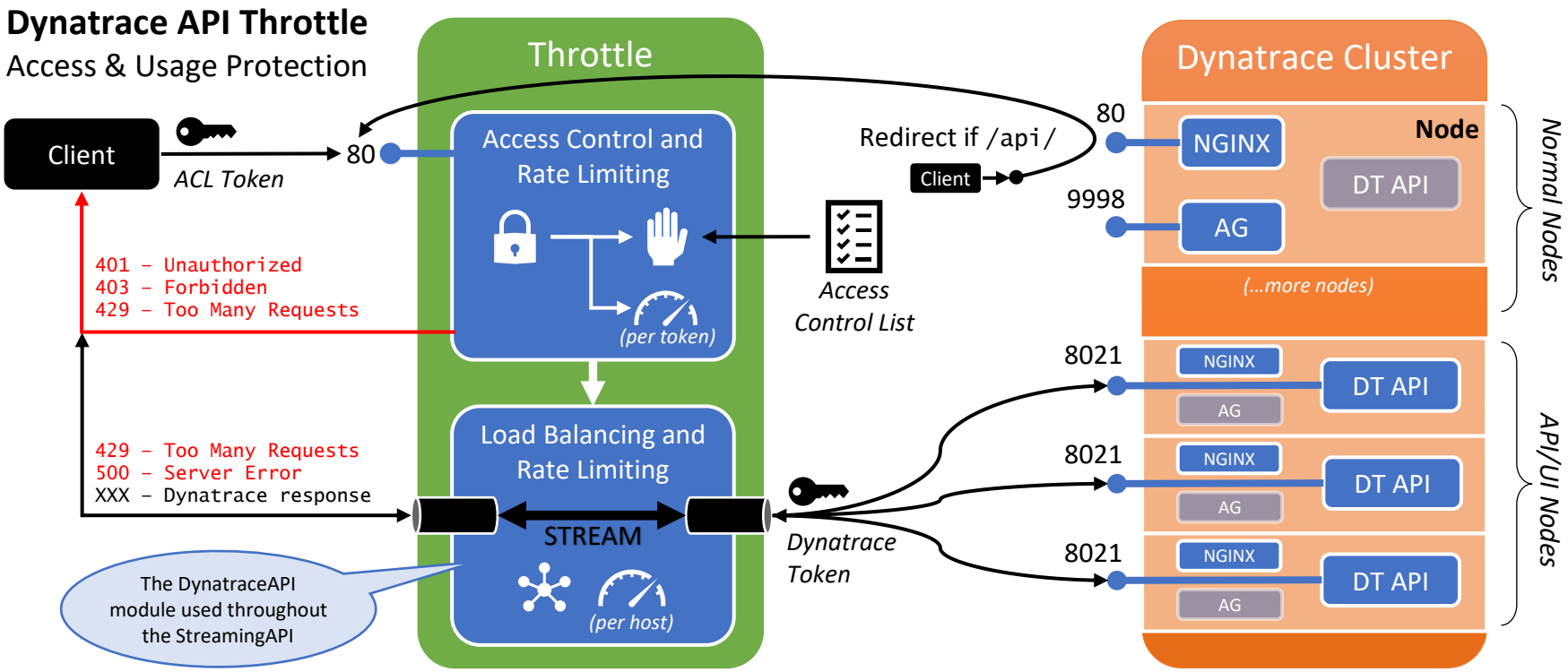
Queue time exceeded

429

Host

XXX

DONE



TODO: ConfigService