| 1  | Detailed Documentation | 3  |
| 2  | Source Code            | 11 |
| 3  | Further information    | 13 |
| 4  | License                | 15 |
Mozilla Monolith is a project to provide statistic gathering, aggregation, a web-service API and a dashboard. The first consumer of Monolith is the Firefox marketplace. Statistics include amongst others public global page views / hits, application specific downloads and even payment related information.

This aggregator part deals with gathering data from multiple sources, bringing them into a common format and storing them.

Currently data is stored in MySQL for durable archival and ElasticSearch is used provide the actual data access for the web-service and dashboard.

The web-service and dashboard are implemented in monolith.

There’s also a Python client library: monolith-client.

Here’s the high-level overview of the whole system:

**monolith-aggregator** adds data to a MySQL database, using a command-line script called *monolith-extract*.

This script that grabs data from various sources and adds them in the SQL Database.

The script also indexes all the content in Elastic Search, which can be used to do time-series queries on the data.

The SQL database is the *single source of truth* of Monolith. The Elastic Search can be recreated at anytime for any date range, using the SQL database.

A typical setup is to run *monolith-extract* every day as a cron.
1.1 Installation

1.1.1 Prerequisites

Currently we only support Linux and Mac OS.

- Python 2.7
- virtualenv (available as such on the $PATH)
- Java Runtime Environment 6 or 7
- libevent-dev
- Git

1.1.2 Steps

Get the code:

```
git clone git://github.com/mozilla/monolith-aggregator.git
```

Build:

```
cd monolith-aggregator
make
```

Run tests:

```
make test
```

To build the docs available in ./docs/_build/html/index.html:

```
make docs
```

1.2 Configuration

monolith-extract is based on a configuration file that defines sequences to run.

Each source and each target are defined in a section prefixed by source: or target.
The file also defines **phases**, that reunite a list of **sources** and **targets**.

Last, a **sequence** is built using a list of **phases**.

Everything is run asynchronously but the system makes sure a phase is over when starting a new one in a sequence.

This is useful when you need a two-phase strategy.

Each **target** and **source** section has two mandatory options:

- **id**: a unique identifier. The identifier is prefixed by **source:** or **target:** in the system, so you can have
  - **use**: the fully qualified name of the plugin class that will be used.

The rest of the section is passed to the plugin.

Here’s a full example:

```
[monolith]
timeout = 10
database = mysql+pymysql://monolith:monolith@localhost/monolith
sequence = extract, load

[phase:extract]
sources = ga
targets = sql

[phase:load]
sources = sql
targets = es

[target:es]
id = es
use = monolith.aggregator.plugins.es.ESWrite
url = http://es/is/here

[source:sql]
id = sql
use = monolith.aggregator.db.Database
database = mysql+pymysql://monolith:monolith@localhost/monolith

[target:sql]
id = sql
use = monolith.aggregator.db.Database
database = mysql+pymysql://monolith:monolith@localhost/monolith

[source:ga]
id = ga-pageviews
use = monolith.aggregator.plugins.ganalytics.GoogleAnalytics
metrics = ga:pageviews
dimensions = browser
oauth_token = %(here)s/auth.json
profile_id = 12345678
```

**use** points to a callable that will be invoked with all the other variables of the section and the variables defined in **monolith** to perform the work.

**monolith-extract** invokes in parallel every source & target callable, using a queue where the data is produced and consumed - once the import is successful, the script calls the purge method on every source, allowing them to cleanup if needed.

The call on purge() implies that the data was safely pushed in the MySQL database.
monolith-extract can run the predefined sequence, or one passed as an option. This is useful when you just need to replay a specific phase.

1.3 Monolith MySQL

MySQL role in Monolith is just to keep the data in case we want to rebuild various Elastic Searches indexes. The database stores directly JSON objects, as blobs. It has the following fields:

- **id**: a unique id per row.
- **date**: the date of the data.
- **type**: the type of data we are storing.
- **source_id**: a unique identifier of the source
- **data**: the JSON object, as a blob.

We have two types of interactions with the database:

1. the script that grabs data from various sources and feeds the Database
2. the script that queries the Database on specific date ranges and feed the Elastic Search indexes.

All metrics that are collected from various sources are stored into MySQL, in a single table that has a **type**, a **date** and a **source_id** field. The data itself is stored as-is in a binary field.

We're planning to store roughly 1M lines per month, so 12M lines per year, and eventually shard the storage into one table per year - so we limit the size of the table to 12M lines.

The sharding will not impact metrics queries that are made through Elastic Search - but the reindexation script will have to take into account this sharding.

1.3.1 MySQL configuration

```sql
SET GLOBAL innodb_file_format='Barracuda'
SET GLOBAL innodb_file_per_table=1
```

1.4 Monolith ElasticSearch

Notes on configuration and setup of ElasticSearch for monolith.

1.4.1 Assumptions

In order to gain performance, we can lower precision and latency for a lot of the metrics. But looking forward, being near real-time and high precision would still be nice.

Currently external systems do some of the aggregation, but in the future we might want to do this work inside monolith to gain more real-time and precision.
1.4.2 Time-series indexes

Multiple indexes are created to cover a distinct time period each. The best time unit depends on the exact data volume and retention policies. High-volume log/event setups use daily indexes, but that’s likely complete overkill for our concerns.

Currently all stats are kept in a single (sharded) index with roughly 45gb of raw data in total.

The suggested starting setup is to use monthly indexes. Old months can then be easily archived / rolled up into lower granularity (1 minute / daily / weekly etc. time precision). So there would be indexes like:

<table>
<thead>
<tr>
<th>Index Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_2013-01</td>
</tr>
<tr>
<td>time_2012-12</td>
</tr>
<tr>
<td>time_2012-11</td>
</tr>
<tr>
<td>time_2012-10</td>
</tr>
</tbody>
</table>

All writes would happen to the correct index based on the timestamp of each entry.

In addition each index can be split up into multiple shards, to distribute load across different servers. At first a shard size of 1 is used, so each shard holds roughly one month of data. Typical queries are for the last 30 days, so usually involve queries for the current and former month. In this setup that’s querying two index shards.

Replication isn’t needed to gain protection against data-loss, as ES isn’t the primary persistent storage. We’d still use a replication factor of 1 (meaning there’s two copies of all data) to spread read-load across multiple servers. Depending on the load, we could increase the shard count for the current and last month, as these are likely queried a lot more often than the older data.

**Note** These index/shard settings are aimed to keep the data per index at a manageable size (for example for the JVM / memory requirements) per server. And at the same time minimize the number of indexes involved in each query, to avoid the associated overhead. In addition it’s easy to drop out or replace old data, as its just disabling an index, but there’s no need to rewrite/update any data. All but the current index can also be highly compacted / optimized (down to one lucene segment), as they’ll never change and backup tools likely appreciate a large amount of static data as well.

Note that you don’t need to manually specify the indexes yourself, but Elastic Search allows you to read from _all or time_* indexes at once.

1.4.3 elasticsearch.yml

We don’t need to have any custom elasticsearch.yml settings, as we are managing all of these settings via index templates and cluster API calls.

1.4.4 Articles / videos

- [http://edgeofsanity.net/article/2012/12/26/elasticsearch-for-logging.html](http://edgeofsanity.net/article/2012/12/26/elasticsearch-for-logging.html)
- Shay Banon at BB 2012 - [http://vimeo.com/44716955](http://vimeo.com/44716955)
- Shay Banon at BB 2011 - [http://vimeo.com/26710663](http://vimeo.com/26710663)
1.5 High-availability

1.5.1 Multi-DC setup

ElasticSearch doesn’t have any good built-in multi-DC support. Inside a single DC you can configure various degrees of HA via setting up replica counts for each index. In our planned 3-node cluster we use a single replica, and thus can lose a single node out of three.

For monolith, we can use a trick for the multi-DC case: Since all the data is stored in MySQL, we can simply run two (or any number) completely separate ElasticSearch clusters in each DC. And then have a job to update the data in each DC from the single MySQL source. Since we introduced per-record uuid’s we can run integrate checks and push new / remove old records in each ES cluster.

Connection problems or downtimes of some of the DC’s can lead to stale data for some period of time, but won’t cause any data-integrity issues.

The MySQL source is only setup in a single DC (though with a master-master) setup. A downtime of the DC can lead to stale data, but won’t cause downtime or complete service unavailability for the end-user facing site.

For MySQL we can run a slave in each extra DC. The replication can be async, to prevent it from slowing down or impacting availability in the primary DC. With that setup, we ship all new data as large chunks of compressed MySQL data to each DC. We can then run the MySQL to ES load job in each DC. This avoids running many HTTP connections across inter-DC network links, to fill data from MySQL to each ES-cluster.

In a later stage we can look at Cassandra as a replacement for MySQL to have a reliable cross-DC setup for the persistent storage component of the system.

1.6 Plugins

A plugin is a class that Monolith uses to extract or inject data.

Writing a plugin is done by using the `aggregator.plugins.Plugin` base class and overriding a few methods.

You can create:

- `source` plugins, that will be used to extract data from a source.
- `target` plugins, that will be used to inject data previously extracted.
- `hybrid` plugins, that implement both behaviors.

When a plugin is instanciate, it gets all options that were defined in the configuration file section. The base class constructor takes care of setting the `options` inject attribute.

When Monolith is run, a single instance of plugin is created per source and target sections.

1.6.1 Source plugins

A source plugin must implement one method called `extract`. The method takes two parameters: `start_date` and `end_date`, which defines the range of the extraction. The plugin must return an iterator containing lines of data.

Each line is a mapping that contains the following keys:

- `_date`: the date of the data line - mandatory
- `_type`: the type of the data - mandatory
Monolith Documentation, Release 1.0beta

Every extra key will be stored as data.

Example:

```python
from aggregator.plugins import Plugin

class MyPlugin(Plugin):
    def extract(self, start_date, end_date):
        date = start_date
        while date <= end_date:
            # extract data from somewhere ...
            data = get_data(date)

            # add date and type keys
            data['_date'] = date
            data['_type'] = 'app_installs'
            yield data
            date += datetime.timedelta(days=1)
```

Some plugins may need to purge the data once the extraction occurred.

To do this you need to implement the `purge` method:

```python
from aggregator.plugins import Plugin

class MyPlugin(Plugin):
    def purge(self, start_date, end_date):
        # purge source data for this date range
```

### 1.6.2 Target plugins

Target plugins need to use the same base class, but implement the `inject` method. The method gets a iterable of lines to inject.

Example:

```python
from aggregator.plugins import Plugin

class MyPlugin(Plugin):
    def inject(self, batch):
        for line in batch:
            # put the data somewhere
```

### 1.6.3 Hybrid plugins

Hybrid plugins implement both behaviors. This can be useful if you want to share a common set of options.

Example:

```python
from aggregator.plugins import Plugin

class MyPlugin(Plugin):
```
def inject(self, batch):
    for line in batch:
        # put the data somewhere

def extract(self, start_date, end_date):
    date = start_date
    while date <= end_date:
        # extract data from somewhere ...

def purge(self, start_date, end_date):
    # purge source data for this date range

1.7 Changelog

1.7.1 0.1 (unreleased)

- Initial implementation.
All source code is available on github under monolith-aggregator.
You can read up on some background and motivation.
There is also a detailed product specification available to the public.
We also keep some progress information in an etherpad.
monolith-aggregator is offered under the MPL 2.0.