Grouperfish Documentation

Release 0.1-SNAPSHOT

Eshwaran Kumar, Michael Kurze, Xavier Stevens, Hamilton Ulmer

Sep 27, 2017

Contents

Note: This documentation serves as a specification. It describes a system that has not reached a usable state yet.

Contents:

CHAPTER 1

Introduction

Grouperfish is built to perform text clustering for Firefox Input. Due to its generic nature, it also serves as a testbed to prototype machine learning algorithms.

How does it work?

Grouperfish is a *document transformation system*, for high throughput applications.

Roughly summarized:

- users put *documents* into Grouperfish using a REST interface
- *transformations* are performed on one or several subsets of these documents.
- *results* can be retrieved by users over the REST interface
- all components are distributed for high volume applications

What can be done?

Assume a scenario where a steady stream of documents is generated. For example:

- user feedback
- software crash reports
- twitter messages

Now, these documents can be processed to make them more useful. For example:

- clustering (grouping related documents together, detecting common topics)
- classification (associating documents with predefined categories including spam)
- trending (identifying new topics over time).

Vocabulary

Grouperfish users can assume one of three roles (or any combination thereof):

Document Producer Some user (usually another piece of software) that will put documents into the System.

Result Consumer Some user/software that gets the generated results.

Admin A user who configures which subsets of documents to transform, but also how and when to do that.

CHAPTER 2

Architecture

Grouperfish consists of three independently functioning components.

The REST Service

Consumers of Grouperfish interact exclusively with this service. It exposes a RESTful API to insert documents and query results. There are APIs for configuring which sets of documents to process (using queries) and what to do with them (using transforms).

For example, you may want to create a query for all documents from January and transform this set by clustering them together.

Namespaces

So that multiple users (of a single Grouperfish cluster) can each work with their own set of documents and transforms, without affecting one another, the concept of *namespaces* is used throughout all parts of Grouperfish. Namespaces are similar to databases in MySQL or indexes in ElasticSearch.

Filters

Incoming documents might require post-processing to make them usable in transforms. For example, you may want to do language detection so that you can cluster documents in the same language together.

Storage

The storage system stores that is inserted into Grouperfish and makes it available for retrieval. Under the hood, the Bagheera component is used to manage storage, indexing and caching.

The Batch System

The processing of document is kicked off by POST-ing to a special REST URL (e.g. triggered by cron or a client system). This triggers a *batch run*. But how does the batch system know which documents to process, and what transforms to apply?

Queries

Queries help the batch system determine which documents to process. In Grouperfish, a query is represented as a JSON-document that conforms to the ElasticSearch Query DSL. Internally, all stored documents are indexed by ElasticSearch, and each query is actually processed only by ElasticSearch itself (and not by Grouperfish).

Transforms

Transforms are programs that operate on a set of documents to generate a result such as clusters, trends, statistics and so on. They can be implemented in arbitrary technologies and programming languages, e.g. using the hadoop Map/Reduce, as long as they can be setup and executed on a Unix-like platform.

Other than the batch system and the REST service, transforms are not aware of things such as queries or namespaces. They act only based on data that is immediately presented to them by the system.

See also:

Transforms

The Batch Process

When the batch process is triggered for a namespace:

- The batch system retrieves all queries and all transform configurations that are defined within the namespace.
- The system uses the first query to get all matching documents from ElasticSearch.

  It exports this first set of documents to a file system, as input to the transform. Transforms run either

  - *locally*, working in a directory on a locally mounted file system
  - or *distributed*, that is, managed by Hadoop

    Such transforms are working in an HDFS directory.
- The system also puts the *transform parameters* (from the transform's configuration) into a place where the transform can use them.
- Finally, the transform is launched and receives the location of the source documents and of the configuration as a parameter.

  If such a *batch run* succeeds, the generated result is put into storage.
- These steps are repeated for all other combinations of query and transform configuration.

CHAPTER 3

Installation

Prerequisites

These are the requirements to run Grouperfish. For development, see *Hacking*.

- A machine running a **Unix-style OS** (such as *Linux*).

  Support for windows currently not planned (and probably not easy to add).

  So far, we have been using Red Hat 5.2 and – for development – Mac OS X 10.6+.
- **JRE 6** or higher
- **Python 2.6** or higher (*not* tested with 3.x)
- **ElasticSearch 0.17.6**

The ElasticSearch cluster does not need to be running on the same machines as Grouperfish. For Hadoop/HBase you will need to make sure that the configuration is on your classpath (easiest with a local installation).

Prepare your installation

- Obtain a grouperfish tarball[1] and unpack it into a directory of your choice.

```
> tar xzf grouperfish-0.1.tar

> cd grouperfish-0.1
```

- Under `config`, modify the `elasticsearch.yml` and `elasticsearch_hc.yml` so that Grouperfish will be able to discover your cluster. **Advanced:** You can modify the `elasticsearch.yml` to make each Grouperfish instance run its own ElasticSearch data node. By default, Grouperfish depends on joining an existing cluster though. Refer to the ElasticSearch configuration documentation for details.

[1] right now, the only way is to build it from source. See *Hacking*.

- In the `hazelcast.xml`, have a look at `<network>` section. If your network does not support multicast based discovery, make changes as described in the Hazelcast documentation.

Launch the daemon

To run grouperfish (currently, no service wrapper is available):

```
grouperfish-0.1> ./bin/grouperfish -f
```

Grouperfish will be listening on port 61732 (mnemonic: `FISH = 0xF124 = 61732`).

You can safely ignore the logback warning (which will only appear with `-f` given). It is due to an error in logback.

Omit the `-f` to run grouperfish as a background process, detached from your shell. You can use `jps` to determine the process id.

CHAPTER 4

Rest API

This is a somewhat formal specification of the Grouperfish REST api. Look at the *Usage* chapter for specific examples.

Primer

The REST architecture talks about *resources*, *entities* and *methods*:

- In grouperfish, each *entity* (*document*, *result*, *query*, *configuration*) is represented as a piece of JSON.
- All entities are JSON documents, so the request/response Content-Type is always `application/json`.
- The *resources* listed here contain `<placeholders>` for the actual parameter values. Values can use any printable unicode character, but URL-syntax (`?#=+/` etc.) must be escaped properly.
- Most resources in Grouperfish allow for several HTTP *methods* to create/update (`PUT`), read (`GET`), or `DELETE` entities. Where allowed, resources respond like this to these methods:

  `PUT` The request body contains the entity to be stored. Response status is always `201 Created` on success. The response status does not allow to determine if an existing resource was overridden.

  `GET` Status is either `200 OK` accompanied with the requested entity in the response body, or `404 Not Found` if the entity name is not used.

  `DELETE` Response code is always `204 No Content`. No information is given on wether the resource existed before deletion.

For Document Producers

Users that push documents can have a very limited view of the system. They may see it just as a sink for their data.

Documents

Resource	`/documents/<ns>/<document-id>`
Entity type	*document* e.g. `{"id": 17, "fooness": "Over 9000", "tags": ["ba", "fu"]}`
Methods	`PUT`, `GET`

This allows to add documents, and also to look them up later.

It may take some time (depending on system configurations: seconds to minutes) for documents to become indexed and thus visible to the batch processing system.

For Result Consumers

Users that are (also) interested in getting results need to know about queries, because each result is identified using the source query name. They might even specify queries on which batch transformation should be performed.

Queries

A query is either an *concrete query* in ElasticSearch Query DSL, or a *template query*.

Resource	`/queries/<ns>/<query-name>`
Entity type	*query* e.g. `{"prefix": {"fooness": "Ove"}}`
Methods	`PUT`, `GET`, `DELETE`

After a `PUT`, when batch processing is performed on this namespace for the next time, documents matching the query will be processed for each configured transform.

The result can then be retrieved using `GET /results/<ns>/<query-name>`.

To submit a template query, nest a normal query in a map like this:

```
curl -XPUT /queries/mike/myQ -d '{
    "facet_by": ["product", "version"],
    "query": {"match_all": {}}
}'
```

See also:

Queries

Results

For each combination of ES query and transform configuration, a result is put into storage during the batch run.

Resource	`/results/<ns>/<transform-name>/<query-name>`
Entity type	*result* e.g. `{"output": ..., "meta": {...}}`
Methods	`GET`

Return the last transform result for a combination of transform/query. If no such result has been generated yet, return `404 Not Found`.

To retrieve results for template queries, you need to specify actual values for your facets. Just add the `facets` parameter to your get requests, containing a `key:value` pair for each facet. Assuming the query given in the previous example has been stored in the system, along with a transform configuration named *themes*, you can get results like this:

```
curl -XGET /results/mike/themes/myQ?facets=product%3AFirefox%20version%3A5
```

What exactly a result looks like is specific to the transform. See *Transforms* for details.

For Admins

There is a number of administrative APIs that can either be triggered by scripts (e.g. using `curl`), or using the admin web UI.

Configuration

To use a filter for incoming documents, or a transform in the batch process, a named piece of configuration needs to be added to the system.

Resource	`/configuration/<ns>/<type>/<name>`
Entity type	`configuration` e.g. `{"transform": "LDA", "parameters": {"k": 3, ...}}`
Methods	`PUT`, `GET`, `DELETE`

Type is currently one of `"transform"` and `"filter"`.

Note: Filters are not yet available as of Grouperfish 0.1

See also:

Transforms, *Filters*

Batch Runs

Batch runs can be kicked off using the REST API as well.

Resource	`/run/<ns>/<transform-name>/<query-name>`
Entity Type	N/A
Methods	`POST`

Either transform name, or both query and transform name can be omitted to run all transforms on the given query, or on all queries in the namespace.

If a batch run is already executing, this run is postponed.

The response status is `202 Accepted` if the run was scheduled, or `404 Not Found` if either query or transform of the given names do not exist.

See also:

Batch System

CHAPTER 5

Usage

Having started up at least one Grouperfish node, these examples should get you started with using the REST api to add documents, queries and transforms, and to retrieve results.

Throughout the examples, we are using the `input` namespace because we are dealing with Firefox Input style data. Keep in mind that this is just an arbitrary identifier that has no further meaning by itself.

Add individual documents

To add a document, use the `documents` resource:

```
> curl -XPUT "http://localhost:61732/documents/input/123456789" \
     -H "Content-Type: application/json" \
     -d '
{
    "id": "123456789",
    "product": "firefox",
    "timestamp": "1314781147",
    "platform": "win7",
    "text": "This is an interesting test document.",
    "manufacturer": "",
    "locale": "id",
    "device": "",
    "type": "issue",
    "url": "",
    "version": "6.0"
}'
```

Verify the success of your operation by getting the document right back:

```
> curl -XGET "http://localhost:61732/documents/input/123456789"
```

Batch-load a large number of documents

Grouperfish only becomes really interesting if you use it with a larger number of documents. There is a tool that allows you to load the entire input data set.

```
grouperfish-0.1> cd tools/firefox_input
firefox_input> curl -XGET http://input.mozilla.com/data/opinions.tsv.bz2 \
                    -o opinions.tsv.bz2
firefox_input> cat opinions.tsv.bz2 | bunzip2 | ./load_opinions input
```

This will run a parallel import of Firefox user feedback data into your Grouperfish cluster (specifically, into the `input` namespace). Depending on your hardware, this should take between 5 and 25 minutes.

Add a query and a transform configuration

Now that we have added a couple of million documents, we need to determine which subset to select, and what to do with the selected documents:

```
curl -XPUT "http://localhost:61732/queries/input/myQ" \
    -H "Content-Type: application/json" \
    -d '
{
    "query": {
        "query_string": {
            "query": "version:6.0 AND platform:win7 AND type:issue"
        }
    }
}'

curl -XPUT "http://localhost:61732/configurations/input/transforms/myT" \
    -H "Content-Type: application/json" \
    -d '
{
    "transform": "textcluster",
    "parameters": {
        "fields": {
            "id": "id",
            "text": "text"
        },
        "limits": {
            "clusters": 10,
            "top_documents": 10
        }
    }

}'
```

Now we have a named query (*myQ*), which selects Firefox 6 issues from Windows 7 users, and a named transform configuration (*myT*). The query is in ElasticSearch QueryDSL syntax which (using a lucene query string). The configured transform is *textcluster*, and it is configured for the top 10 topics, with the top 10 messages each.

Run the transform

```
curl -XPOST "http://localhost:61732/run/input/myT/myQ"
```

This fetches everything matching *myQ* (about 20 thousand documents) and invokes textcluster on them (this takes a couple of seconds).

Get the results

Getting the transform result works fairly similarly:

```
curl -XGET "http://localhost:61732/results/input/myT/myQ"
```

Results can be fairly large since they contain the full top-documents for each cluster. A tool such as the JSONView Firefox addon can be used to browse results more comfortably.

CHAPTER 6

Filters

As of Grouperfish 0.1, filters are not yet available.

CHAPTER 7

Batch System

Batch runs are launched by a post request to a `/runs` resource, as described in the section *Rest API*.

Batch Operation

The Batch System performs these steps for every batch run:

1. **Get queries to process**
   - If a query was specified when starting the run:

     Fetch that one query.
   - Otherwise:
     (a) Fetch all concrete queries for this namespace
     (b) Fetch all template queries for this namespace
     (c) Resolve the template queries (see *Template Queries*)

     Add the results to the concrete queries obtained in (I)
2. **Get transform configurations to use**
   - If a transform configuration was specified in the POST request:

     Fetch that one transform
   - Otherwise:

     Fetch *all* transform configurations for this namespace
3. **Run the transforms**

   For each concrete query
   (a) Get the matching documents
   (b) Write them to some `hdfs://` directory

(c) Call the transform executable with that directory's path (see *The Transform API*)

(d) Tag documents in ElasticSearch (if the transform has generated tags, see *Tagging*)

(e) POST the results summary document to the rest service.

From this point it will be served to consumers.

The Transform API

Each batch transform is implemented as an executable that is invoked by the system. This allows for a maximum of flexibility (free choice of programming language and library dependencies).

Directory Layout

All transforms have to conform to a specific directory layout. This example shows the fictitious `bozo-cluster` transform.

`grouperfish`

- `transforms`
  - `bozo-cluster`
    - `bozo-cluster*`
    - `install*`
    - `parameters.json`
    - ...
  - `lda`
    - ...
  - ...

To be recognized by grouperfish, there has to be both an `install` script, and (possibly only after `install` script has been called), the executable itself. The executable must have the same name as the directory. Third, there should be a file `parameters.json`, containing the possible parameters for this transform.

Invocation

Each transform is invoked like with HDFS path (in URL syntax) as the first argument, something like `hdfs://namenode.com/grouperfish/..../workdir`.

Here are the contents of this working directory when the transform is started:

- `input.tsv`

  Each line (delimited by `LF`) of this `UTF-8` coded file contains two columns, separated by `TAB`:

  1. The ID of a document to process (as a base10 string)
  2. The full document as JSON, on the same line: Any line breaks within strings are escaped. Apart from formatting, this document is the same that the user submitted originally.

  Example:

```
4815162342``  ``{"id":"4815162342", "text": "some\ntext"}
4815162343``  ``{"id":"4815162343", "text": "moar text"}
...
```

- `parameters.json`

  The parameters section from the transform configuration. This corresponds to the possible parameters from the transform home directory.

  Example:

```
{
  "text" : {
    "STOPWORDS": [ "the", "cat" ]
    "STEM": "false",
    "MIN_WORD_LEN": "2",
    "MIN_DF": "1",
    "MAX_DF_PERCENT": "0.99",
    "DOC_COL_ID": "id",
    "TEXT_COL_ID": "text"
  },
  "mapreduce": {"NUM_REDUCERS": "7"},
  "transform": {
    "KMEANS_NUM_CLUSTERS": "10",
    "KMEANS_NUM_ITERATIONS": "20",
    "SSVD_MULTIPLIER": "5",
    "SSVD_BLOCK_HEIGHT": "30000",
    "KMEANS_DELTA": "0.1",
    "KMEANS_DISTANCE": "CosineDistanceMeasure"
  }
}
```

When the transform succeeds, it produces these outputs in addition:

- `output/results.json`

  This JSON documents will be visible to the result consumers through the REST interface. It should contain all major results that the transform generates.

  The batch system will add a `meta` map before storing the result, containing the name of the transform configuration (`transform`), the date (`date`), the query (`query`), and the number of input documents (`input_size`).

  The transform is also allowed to create the `meta` map, to add transform-specific diagnostics.

- `output/tags.json` (optional)

  The batch system will take this map from document IDs to tag names, and modify the documents in ElasticSearch, so they can be looked up using these labels. See *Tagging* for details.

The transform should exit with status `0` on success, and `1` on failure. Errors will be logged to standard error.

Tagging

When an transform produces a `tags.json` as part of its result, the batch system uses it to markup results in ElasticSearch. Transforms can output cluster membership or classification results as tags, which will allow clients to facet and scroll through the transform result using the full ElasticSearch API.

A document with added tags looks like this:

```
{
  "id": 12345,
  ...
  "grouperfish": {
    "my-query": {
      "my-transform": {
        "2012-12-21T00:00:00.000Z": ["tag-A", "tag-B"],
        ...
      }
    }
  }
}
```

The timestamps are necessary because old tags become invalid when tagged documents drop out of a result set (e.g. due to a date constraint). The grouperfish API ensures that searches for results take the timestamp of the last transform run into account.

Note: This format is not finalized yet. We might use parent/child docs instead. Also, the necessary REST API that wraps ElasticSearch is not defined yet.

CHAPTER 8

Transforms

Transforms are the heart of Grouperfish. They generate the results that will actually be interesting to consumers.

Note: The minimal transform interface is defined by the *Batch System*

Transform Configuration

The same transform (e.g. a clustering algorithm) might be used with different parameters to generate different results. For this reason, the system contains a *transform configurations* for each result that should be generated.

Primarily, a transform configuration parameterizes its transform (e.g. for clustering, it might specify the desired number of clusters). It can also be used to tell the Grouperfish batch system how to interact with a transform.

Currently, a transform configuration is a JSON document with two fields: The *transform* determines which piece of software to use, and *parameters* tells that software what to do. Example configuration for the *textcluster* transform:

```
{
    "transform": "textcluster",
    "parameters": {
        "fields": {"id": "id", "text": "text"},
        "limits": {"clusters": 10,"top_documents": 10}
    }
}
```

Result Types

Topics (or Clusters)

Clustering transforms try to extract the main topics from a set of documents. As of Grouperfish version 0.1, the only available transform is a clustering transform named textcluster. The results of clustering transform are topics, the structure of the result is as follows:

```
{
    "clusters": [
        {
            "top_documents": [{...}, {...}, ..., {...}],
            "top_terms": ["Something", "Else", ..., "Another"]
        },
        ...
    ]
}
```

Depending on the actually configured transform, only top documents *or* top terms might be generated for a topic. Also, any given transform might add other top-level fields than just *clusters*.

Available Transforms

textcluster

Textcluster is a relatively simple clustering algorithm written in Python by Dave Dash for Firefox Input. It is very fast for small input sets, but requires a lot of memory, especially when processing more than 10,000 documents at a time. Textcluster is available on github.

In Grouperfish, you can select how many topics you want textcluster to extract, and how many documents to include in the results for each topic.

- Parameters

  ```
  {
      "fields": {
          "id": "id",
          "text": "text"
      },
      "limits": {
          "clusters": 10,
          "top_documents": 10
      }
  }
  ```

  These are the default parameters (top 10 topics/clusters, with 10 documents each).

- Results

  Textcluster uses the standard clustering result format (see above), but does not inclue top terms, only documents.

CHAPTER 9

Queries

Concrete Queries

A concrete query is just a regular ElasticSearch query, e.g.:

```
{
  "query": {
    "bool": {
      "must": [
        {"field": {"os": "Android"}},
        {"field": {"platform": "ARM"}},
      ]
    }
  }
}
```

All documents matching this query will be processed together in a batch run.

Note: Find the full Query DSL documentation on the ElasticSearch Website.

Template Queries

A template query will generate a bunch of concrete queries every time it is evaluated. It is different in that it has an additional top-level field "facet_by", which is a list of field names.

Let us assume we have these documents in our namespace:

```
{"id": 1, "desc": "Why do you crash?", "os": "win7", "platform": "x64"},
{"id": 2, "desc": "Don't crash plz", "os": "xp", "platform": "x86"},
{"id": 3, "desc": "It doesn't crash!", "os": "win7", "platform": "x86"},
{"id": 3, "desc": "Over 9000!", "os": "linux", "platform": "x86"},
```

And this template query:

```
{
  "query": {"text": {"desc": "crash"}},
  "facet_by": ["platform", "os"]
}
```

This will generate the following set of queries:

```
{"query": {"filtered":
    {"query": {"text": {"desc": "crash"}}, "filter": {"and": [
        {"field": {"os": "win7"}},
        {"field": {"platform": "x64"}},
]}}}}
{"query": {"filtered":
    {"query": {"text": {"desc": "crash"}}, "filter": {"and": [
        {"field": {"os": "win7"}},
        {"field": {"platform": "x86"}},
]}}}}
{"query": {"filtered":
    {"query": {"text": {"desc": "crash"}}, "filter": {"and": [
        {"field": {"os": "xp"}},
        {"field": {"platform": "x86"}},
]}}}}
```

Note that no query for `os=linux` is generated in this case, because the query for `crash` does not match any document with that `os` in the first place.

CHAPTER 10

To Do

These components are not necessarily listed in the order they need to be implemented:

- Filtering functionality (*Filters*)
  - Language detection filter
- Allow clients to extract sub-results from a result doc (using JSON paths)
- Add template Queries
- Add tagging of ElasticSearch documents based on transform results
- *Transforms*
  - Co-Clustering
  - LDA
- Validate configuration pieces based on a schema, specific to each filter/transform
- JS client library (possibly hook in with `pyes`) E.g. to be used by the admin interface.
- Admin interface
- Python client library (possibly hook in with `pyes`)
- Online service for ad-hoc requests
  - Define online API (Client/server? JVM using Jython etc.?)
  - Integrate a fast clustering algorithm for this

CHAPTER 11

Hacking

Prerequisites

First, make sure that you satisfy the requirements for running grouperfish (*Installation*).

Maven We are using Maven 3.0 for build and dependency management of several Grouperfish components.

JDK 6 Java 6 Standard Edition should work fine.

Git & Mercurial To get the Grouperfish source and dependencies.

Sphinx For documentation. Best installed by running

```
easy_install Sphinx
```

The Source To obtain the (latest) source using **git**:

```
> git clone git://github.com/mozilla-metrics/grouperfish.git
> cd grouperfish
> git checkout development
```

Building it:

```
> ./install              # Creates a build under ./build
> ./install --package    # Creates grouperfish-$VERSION.tar.gz
```

When building, you might get Maven warnings due to expressions in the `'version'` field, which can be safely ignored.

Coding Style

In general, consistency with existing surrounding code / the current module is more important for a patch than adherence to the rules listed here (local consistency wins over global consistency).

Wrap text (documentation, doc comments) and Python at 80 columns, everything else (especially Java) at 120.

Java This project follows the default Eclipse code format, except that 4 spaces are used for indention rather than `TAB`. Also, put `else`/`catch`/`finally` on a new line (much nicer diffs). Crank up the warnings for unused identifiers and dead code, they often point to real bugs. Help *readers* to reason about scope and side-effects:

- Keep declarations and initializations together
- Keep all declarations as local as possible.
- Use `final` generously, especially for fields.
- No `static` fields without `final`.

For Java projects (service, transforms, filters), *Maven* is encouraged as the build-tool (but not required). To edit source files using Eclipse, the `m2eclipse` plugin can be used.

Python Follow PEP 8

Other Follow the default convention of the language you are using. When in doubt, indent using 4 spaces.

Repository Layout

`transforms` One sub-directory per self-contained transform. Code shared by several transforms can go into `transforms/commons`.

`service` The REST service and the batch system. This must not contain any code or any dependencies that are related to specific transforms.

`docs` Sphinx-style documentation.

`tools` One sub-directory per self-contained tool. These tools can be used by the transforms to convert data formats etc. All tools will be on the transforms' path.

`filters` One self-contained project folder per filter. Shared code goes to `filters/commons`.

`integration-tests` A maven project for building and performing integration tests. We use rest-assured to talk to the REST interface from clients.

Building

The source tree

Each self-contained component (the batch service, each transform/tool/filter) can have its own executable `install` script. Only components that do not need build steps (such as static html tools) can work without such a script. Each of these install scripts is in turn called by the main install script when creating a grouperfish tarball.

```
install*
...
service/
    install*
    pom.xml
```

```
    ...
tools/
    firefox_input/
        ...
    webui/
        index.html
        ...
    ...
transforms/
    coclustering/
        install*
        ...
```

The Build Tree

Each `install` script will put its components into the `build` directory under the main project. When a user unpacks a grouperfish distribution, she will see the contents of this directory:

Each component can have build results into `data`, `conf`, `bin`. The folder `lib` should be used where a component makes parts available to other components (other binaries should go to the respective subfolder).

```
build/
    bin/
        grouperfish*
    data/
        ...
    conf/
        ...
    lib/
        grouperfish-service.jar
        ...
    transforms/
        coclustering/
            coclustering*
            ...
    tools/
        firefox_input/
            ...
        webui/
            index.html
            ...
        ...
```

Components

The Service Sub-Project

The `service/` folder in the source tree contains the REST and batch system implementation. It is the code that is run when you "start" Grouperfish, and which launches filters and transforms as needed.

The service is started using `bin/grouperfish`. For development, the alternative `bin/littlefish` is useful, which can be called directly from the source tree (after an `mvn compile` or the equivalent eclipse build), without packaging the service as a jar first.

It is organized into some basic shared packages, and three *modules* which expose interfaces and components to be configured and replaced independent of each other, for flexibility.

The shared packages contain:

bootstrap the entry point(s) to launch grouperfish

base shared general purpose helper code, e.g. for streams, immutable collections and JSON handling

model simple objects that represent data Grouperfish deals with

util special purpose utility classes, e.g. for import/export, TODO: move these to `tools`

Service Modules

services Components that depend on the computing environment. By configuring these differently, users can chose alternative file systems, indexing or grid solutions can be integrated. Right now this flexibility is mostly used for mocking (testing).

rest The REST service is implemented as a couple of JAX-RS resources, managed by Jetty/Jersey. Other than the service itself (to be started/stopped), there is no functionality exposed api-wise. Most resources mainly encapsulate maps. The `/run` resource also interacts with the batch system.

batch The batch system implements scheduling and execution of tasks, and the preparation and cleanup for each task run. There are *handlers* for each stage of a task (fetch data, execute the transform, make results available). The *transform* objects implement the run itself: they manage child processes, or implement java-based algorithms directly. The *scheduling* is performed by a component that implements the `BatchService` interface. Usually one or more queues are used, but synchronous operation is also possible (for example in a command line version).

On Guice Usage

Components from modules are instantiated using Google Guice. Each module has multiple packages `....grouperfish.<module>.....` The `....<module>.api` package contains all interfaces of components that the module offers. The `....<module>.api.guice` package has the Guice-specific bindings (by implementing the Guice `Module` interface). Launch Grouperfish with different bindings to customize or stub parts.

Grouperfish uses *explicit dependency injection*: every class that needs a service component simply takes a corresponding constructor argument, to be provisioned on construction, without any Guice annotation. This means that Guice imports are mostly used...

- where the application is configured (the bindings)
- where it is bootstrapped
- and in REST resources that are instantiated by jersey-guice
- search