Configuring data fetchers

Overview

DSAPI2 provides a framework for fetching, parsing, and tokenizing external data sources to produce streams.

An external fetcher is a standalone Java application intended to be invoked periodically by a server, e.g., as a cron job. There is a single entry point, and a properties-based configuration determines which classes are used for fetching, parsing, and filtering stream tokens. This is a simple form of dependency injection of the sort that is often done with more complex frameworks such as Spring.

Basic configuration

Several properties are generic and apply to all fetcher and parser implementations. These are:

fetcher.class - the fully-qualified name of the class that will be used to fetch data (note: some parser implementations may ignore the fetcher and perform the fetching themselves)
fetcher.realtime - if true, only the most recent token from each execution will be written to the stream; if false, all tokens produced from each execution that are newer than the stream's most recent token will be written to the stream
fetcher.delay - how long to wait between executions (in milliseconds)

parser.class - the fully-qualified name of the class that will be used to parse data - required

date.extractor.class - the fully-qualified name of the class that will be used to extract dates (note: some parsers may ignore the date extractor and perform timestamping themselves)

stream.assigner.class - the fully-qualified name of the class that will determine which stream tokens will be written to (note: some parsers may ignore the stream assigner)

Example configuration: Twitter

DSAPI2 includes a Twitter parser. It ignores its fetcher, because Twitter-specific-parameters determine what URL needs to be fetched. The following example is annotated to describe each parameter.

```
# the fetcher is ignored, but since we must instantiate something, an HTMLFetcher is used
fetcher.class = edu.uiuc.ncsa.datastream.util.fetch.fetcher.HTMLFetcher
# non-realtime, because we're performing a Twitter search and want all new results
fetcher.realtime = false
# wait 5 minutes between fetches (Twitter is rate-limited)
fetcher.delay = 300000

parser.class = edu.uiuc.ncsa.datastream.util.fetch.dataparser.TwitterParser
# the following four parameters are OAuth authentication parameters
parser.twitter.key = qeS5HHN1s69urZ2SqtJISQ
parser.twitter.secret = sXcEHIlzMqDSsfxrUNe8D4bGOObxsqixmknpmBn8I
parser.twitter.token = 61353510-9TUfOSHMddWSklTzpV23qCqrnK23ev2WdFzlvNP1F
parser.twitter.tokenSecret = nHyQR6Mi5zZvgptgOPDr0JjqGnoASbvzW5wAa5bKBE

# the next few parameters specify a query against Twitter's query API
# for documentation on query syntax see http://search.twitter.com/api/
# this is the query itself. "car" means search for tweets containing the word "car"
parser.twitter.query = car
# here we specify a geographic centroid
parser.twitter.lat = 40.116349
parser.twitter.lon = -88.239183
# and a radius
parser.twitter.radius = 30
# in miles. this is the geographic region in which to search
parser.twitter.distanceUnits = miles

# the date extractor is ignored; the twitter4j API performs date extraction for us
date.extractor.class = edu.uiuc.ncsa.datastream.util.fetch.dateparser.SimpleDateExtractor

# here we're putting all search results into a single stream
stream.assigner.class =  edu.uiuc.ncsa.datastream.util.fetch.ConstantStreamAssigner
# the URI of the stream. this can be any valid URI
stream.assigner.constant.stream = urn:streams/snorb8/twitter


# set up multiple filters
onetimefilter.class = edu.uiuc.ncsa.datastream.util.fetch.filter.MultiFilter
filter.multi.package = edu.uiuc.ncsa.datastream.util.fetch.filter
filter.multi.classes = TypeRegisterFilter,StreamMetadataFilter
# StreamMetadataFilter allows us to include some metadata about our stream
filter.metadata.stream.label = Twitter search for car near CU
# TypeRegisterFilter allows us to associate a content type with token data
# in this case tweets are of type text/plain
filter.typeregister.mime = text/plain
```

Glossary

Fetching. “Pulling” data from a remote service, including authentication and any other protocol interactions required to produce a binary input stream representing desired data.

Parsing. Given a binary input stream produced by a fetcher, locating and interpreting time-series data points (e.g., timestamped rows in a CSV file).

Tokenizing. Given time-series data, producing DSAPI2 stream tokens for insertion into a stream.

Stream assignment. Producing the URI of a stream which will serve as a destination for tokens produced by a parser/tokenizer.

Date extraction. Conversion of a textual representation of a timestamp into an unambiguous numerical representation of the date.

Filtering. Examining a token and either 1) deciding whether or not to insert it in a stream, or 2) taking additional actions such as associating metadata with the token or its destination stream.