# External Fetcher

## 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
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 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 (some parser implementations may ignore the fetcher and perform the fetching themselves)

**date.extractor.class** - the fully-qualified name of the class that will be used to extract dates (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 (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
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
# the example values are not valid; do not attempt to authenticate with them
parser.twitter.key = qeS5HHN1s69Xrz2SqtJISQ
parser.twitter.secret = sXcEHIlzMqDSsfRrUNe8D4bGOObxsqidmknpmBn8I
parser.twitter.token = 61353510-9TUfOSHMdQWSklTzpV23kCqrnK23ev2WdFzlvNP1F
parser.twitter.tokenSecret = nHyQR6Mi5zZvgptZOPDr0JjqGnoASbvyW5wAa5bKBE


# 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/snorb7/twitter


# TypeRegisterFilter allows us to associate a content type with token data
onetimefilter.class = edu.uiuc.ncsa.datastream.util.fetch.filter.TypeRegisterFilter
# 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.

========== below is an old version of the documentation. If in doubt, please refer the above doc==

The External Data Fetcher is a tool that allows fetching data "continuously" from an external source (e.g. HTTP, FTP, POP) and publishing into a stream to be consumed at a later time using the Data Stream API. The External fetcher is extensible and configurable to fetch data from more or less arbitrary sources and perform user-provided operations when data is fetched, such as adding metadata to a stream or triggering a workflow.

# Installing and Running

# Configuration

The External Fetcher utilizes several pluggable components, all of which are set up through a single .properties files:

- Fetcher: The plug-in responsable for fetching the data through whatever protocol they are available. It is specified in the .properties file by the property fetcher.class followed by the qualified name of the class, e.g.

```
fetcher.class = edu.uiuc.ncsa.datastream.util.fetch.fetcher.FileFetcher
```

  Additional properties control the behavior of the fetcher:

```
#Whether to grab only the latest datum available, or all the data not already grabbed
fetcher.realtime = false
#Time in ms to wait between consecutive pulls from the data source
fetcher.delay = 6000000
```

- There are currently three fetchers available:
  1. FileFetcher, with properties fetcher.file.filename
  2. HTMLFetcher, with properties fetcher.html.url, fetcher.html.username, fetcher.html.password
  3. FTPFetcher. Use in conjuntion with FTPParser
- Parser: Takes the raw data fetch by the fetcher (a file, and FTP directory, etc) and extracts the relevant data points and their timestamp. For instance the LineParser will read file line by line and extract data and timestamp according to a configurable pattern. e.g:

```
parser.class = edu.uiuc.ncsa.datastream.util.fetch.dataparser.LineParser
parser.line.separator = ,
parser.line.idColumn = 0
parser.line.dateColumn = 1
parser.line.dataColumn = 14
```

  The available parsers are:

1. LineParser
2. XMLParser, with properties parser.xml.xslt. It transforms XML/HTML files using an XSLT translation into an XML file with the following schema:

```
<tokens>
  <token>
    <date/>
    <data/>
  </token>
</tokens>
```

3. XMLLinkParser, with properties parser.xmllink.xslt. It tranforms XML/HTML files using an XSLT translation into the following format with links to the actual data

```
<tokens>
  <token>
    <date/>
    <link/>
  </token>
</tokens>
```

4. FTPParser, with properties parser.ftp.url, parser.ftp.username, parser.ftp.password and parser.ftp.filename.pattern. Download files with a given name pattern from and FTP directory* Date Extractor: Transform a string (as provided by the Parser) into a timestamp. e.g:

```
date.extractor.class = edu.uiuc.ncsa.datastream.util.fetch.dateparser.SimpleDateExtractor
date.extractor.pattern = ([0-9/]{8})
date.extractor.timezone = CST
date.extractor.format = MM/dd/yy
```

- Stream Name Provider: Provides a URI for the stream corresponding to a set of data, by looking and ID provided by the parser or using a constant URI. Example:

```
stream.assigner.class =  edu.uiuc.ncsa.datastream.util.fetch.IdentityStreamAssigner
stream.assigner.identity.prefix = urn:streams/ogc/well/pnumber/
```

- Pre Filter, Post Filter and One-Time Filter: Filters to determine whether a data point should be accepted and optionally perform additional actions, such as triggering a workflow or registering metadata, after the data is posted to the stream or the first time data is posted to a stream.

# Examples

## Twitter Example

This examples fetches friends' statuses every minute for a user account to be authenticated via the command prompt.

Configuration file:

```
fetcher.class = edu.uiuc.ncsa.datastream.util.fetch.fetcher.HTMLFetcher
fetcher.html.url = http://api.twitter.com/1/statuses/friends_timeline.xml
fetcher.realtime = false
fetcher.delay = 60000

parser.class = edu.uiuc.ncsa.datastream.util.fetch.dataparser.TwitterParser

#Twitter application key
parser.twitter.key = qeS5HHN1s69urz2SqtJISQ
parser.twitter.secret = sXcEHIlzMqDSsfxrUNe8D4bGOObxsqidmknpmBn8I

date.extractor.class = edu.uiuc.ncsa.datastream.util.fetch.dateparser.SimpleDateExtractor
#date.extractor.pattern = ([a-zA-Z0-9: ]{19})
date.extractor.timezone = UTC
date.extractor.format = EEE MMM dd HH:mm:ss Z yyyy

stream.assigner.class =  edu.uiuc.ncsa.datastream.util.fetch.ConstantStreamAssigner
stream.assigner.constant.stream = urn:streams/alejandro/twitter


onetimefilter.class = edu.uiuc.ncsa.datastream.util.fetch.filter.TypeRegisterFilter
filter.typeregister.mime = text/plain
filter.typeregister.format = 1
```

## Yahoo! Weather Example

This examples fetches temperature data for Champaign from Yahoo! Web API

Configuration file:

```
fetcher.class = edu.uiuc.ncsa.datastream.util.fetch.fetcher.HTMLFetcher
fetcher.html.url = http://weather.yahooapis.com/forecastrss?p=61822&u=c
fetcher.realtime = false
fetcher.delay = 6000000

parser.class = edu.uiuc.ncsa.datastream.util.fetch.dataparser.XMLParser
parser.xml.xslt = weather.xsl

date.extractor.class = edu.uiuc.ncsa.datastream.util.fetch.dateparser.SimpleDateExtractor
#date.extractor.pattern = ([a-zA-Z0-9: ]{19})
date.extractor.timezone = CST
date.extractor.format = EEE, d MMM yyyy hh:mm aaa z

stream.assigner.class =  edu.uiuc.ncsa.datastream.util.fetch.ConstantStreamAssigner
stream.assigner.constant.stream = urn:streams/il/champaign/temperature

onetimefilter.class = edu.uiuc.ncsa.datastream.util.fetch.filter.TypeRegisterFilter
filter.typeregister.mime = text/plain
```

weather.xsl:

```
<!DOCTYPE xsl:stylesheet [
   <!ENTITY atom 'http://www.w3.org/2005/Atom'>
    <!ENTITY foaf 'http://foaf.org'>
    <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'> ]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
       version="2.0">
      <xsl:output method="xml" version="1.0" encoding="utf-8" indent="yes"/>
      <xsl:template match="/rss/channel">
      <tokens>
         <xsl:for-each select="./item">
         <xsl:call-template name="statusTemplate">
             <xsl:with-param name="node" select="."/>
        </xsl:call-template>
      </xsl:for-each>
    </tokens>
  </xsl:template>

  <xsl:template name="statusTemplate">
    <xsl:param name="node"/>
    <token>
      <date><xsl:value-of select="$node/pubDate"/></date>
      <data><xsl:value-of select="$node/yweather:condition/@temp"/></data>
    </token>
  </xsl:template>
 </xsl:stylesheet>
```