The GATECloud Paralleliser (GCP)

Large-scale multi-threaded processing with GATE Embedded

version 3.0-SNAPSHOT

Ian Roberts, Valentin Tablan
GATE Team

May 24, 2018

Contents

Chapter 1

Introduction

GCP is a tool designed to support the execution of pipelines built using GATE Developer over large collections of thousands or millions of documents, using a multi-threaded architecture to make the best use of today's multi-core processors. GCP tasks or *batches* are defined using an extensible XML syntax, describing the location and format of the input files, the GATE application to be run, and the kinds of outputs required. A number of standard input and output handlers are provided, but all the various components are pluggable so custom implementations can be used if the task requires it. GCP keeps track of the progress of each batch in a human- and machine-readable XML format, and is designed so that if a running batch is interrupted for any reason it can be re-run with the same settings and GCP will automatically continue from where it left off.

1.1 Definitions

This section defines a number of terms that have specific meanings in GCP.

Batch

A *batch* is the unit of work for a GCP process. It is described by an XML file, and includes the location of a saved GATE application state (a "gapp" file), the location of the *report* file, one *input handler* definition, zero or more *output handler* definitions and a specification of which documents from the input handler should be processed (either as an explicit list of document IDs or as a *document enumerator* which calculates the IDs in an appropriate manner).

Chapter 3 describes the format of batch definition files in detail.

Report

The progress of a running batch is recorded in an XML *report* file as the documents are processed. For each document ID, the report records whether the document was processed successfully or whether processing failed with an error. For successful documents the report includes statistics on how many annotations were found in the document, and for a completed batch it also records overall statistics on the number of documents and total amount of data processed, the total number of successful and failed documents and the total processing time.

The report file for a batch is also the mechanism which allows GCP to recover if processing is unexpectedly interrupted. If GCP is asked to process a batch where the report file already exists it will parse the existing report and ignore documents that are marked as having already been successfully processed. Thus you can simply restart a crashed GCP batch with the same command-line settings and it will continue processing from where it left off on the previous run.

GATE application

A GCP batch specifies the GATE application that is to be run over the documents as a standard "GAPP file" saved application state, which would typically be created using GATE Developer.

Input handler

The *input handler* for a GCP batch specifies the source of documents to be processed. The job of an input handler is to take a document *ID* and load the corresponding GATE `Document` object ready to be processed. There are a number of standard input handlers provided with GCP to take input documents from individual files on disk, directly from a ZIP archive file or from an ARC file as produced by the Heritrix web crawler (`http://crawler.archive.org`). If the standard handlers do not suit your needs then you can provide a custom implementation by including your handler class in a GATE CREOLE plugin referenced by your saved application.

Document enumerator

While the input handler specifies how to go from document IDs to `gate.Document` objects, it does not specify which document IDs are to be processed. The IDs can be specified explicitly in the batch XML file but more commonly an *enumerator* would be used to build a list of IDs by scanning the input directory or archive file. Standard enumerator implementations are provided, corresponding to the standard input handler types, to select a subset of documents from the input directory or archive according to various criteria. As with input handlers, custom enumerator implementations can be provided through the standard CREOLE plugin mechanism.

Output handler

Most batch definitions will include one or more *output handler* definitions, which describe what to do with the document once it has been processed by the GATE application. Standard output handler implementations are provided to save the documents as GATE XML files, plain text and XCES standoff annotations, inline XML ("save preserving format" in GATE Developer terms), and to send the annotated documents to a Mímir server for indexing. Custom implementations can be added using the CREOLE plugin mechanism.

Note that output handler definitions are optional – if you do not specify any output handlers then GCP will not save the results anywhere, but this may be appropriate if, for example, your pipeline contains a custom PR that saves your results to a relational database or similar.

1.2 Processing Model

GCP processes a batch as follows.

1. Parse the batch definition file, and run the document enumerators (if any) to build the complete list of document IDs to be processed.
2. Parse the existing (possibly partial) report file, if one exists, and remove from this list any documents that are already marked as having been successfully processed.
3. Create a thread pool of a size specified on the command line (the default is 6 threads).
4. Load the saved application state, and use `Factory.duplicate` to make additional copies of the application such that there is one independent copy of the application per thread in the thread pool.
5. Run the processing threads. Each thread will repeatedly:
   - take the next available unprocessed document ID from the list.
   - ask the input handler for the corresponding `gate.Document`.
   - put that document into a singleton `Corpus` and run this thread's copy of the GATE application over that corpus.
   - pass the annotated document to each of the output handlers.
   - write an entry to the report file indicating whether the document was processed successfully or whether an exception occurred during processing.
   - release the document using `Factory.deleteResource`.
6. Once all the documents have been processed, shut down the thread pool and call `Factory.deleteResource` to cleanly shut down the GATE applications.

Due to the asynchronous nature of the processing threads, if one document takes a particularly long time to process the other threads can proceed with many other documents in parallel, they are not forced to wait for the slowest thread.

1.3 Changelog

This section summarises the main changes between releases of GCP

1.3.1 3.0 (May 2018)

Updated to work with GATE Embedded version 8.5:

- Minimum required Java version is now Java 8.
- GCP now builds using Maven rather than Ant, and has been split into "api" and "impl" modules. GATE plugins that want to provide custom input or output handlers should declare a "provided" dependency on the appropriate version of `uk.ac.gate:gcp-api`.
- New command line parameters `-C` and `-p` to allow pre-loading of specific GATE plugins in addition to those declared by the saved application. This is useful, for example, to load plugins that provide document format parsers.

1.3.2 2.8.1 (June 2017)

GCP now depends on GATE Embedded 8.4.1. Also the `-i` option to `gcp-direct.sh` can now be a *file* which lists the documents to process, instead of just a *directory* of documents.

1.3.3 2.8 (February 2017)

GCP now depends on GATE Embedded 8.4.

1.3.4 2.7 (January 2017)

This is a minor bugfix release, the main change is that GCP now depends on GATE Embedded 8.3. There have been minor changes to the `gcp-direct.sh` script, in particular it is now possible to run a pipeline with *no* output handler at all, useful in cases where there is a PR within the pipeline that is responsible for handling the output, or if you want to run a pipeline purely for its side effects (e.g. building a frequency table or training some sort of machine learning model).

1.3.5 2.6 (June 2016)

This is a minor bugfix release, the main change is that GCP now depends on GATE Embedded 8.2.

1.3.6 2.5 (June 2015)

- Now depends on GATE Embedded 8.1

- Introduced "streaming" style input and output handlers for JSON data (e.g. from Twitter), which can read a series of documents from a single JSON input file, and write JSON results to a single concatenated output file (sections 3.2.4 and 3.3.1).
- Introduced the `gcp-direct.sh` script to cover simple invocations of GCP without the need to write a batch definition XML file (section 2.2.2).
- For "controller-aware" PRs[1], the various callbacks are now invoked just once per batch rather than before and after every single document.

1.3.7 2.4 (May 2014)

- Now depends on GATE Embedded 8.0 (and thus requires Java 7 to run)
- Added input handler for WARC format archives, to complement the existing ARC handler (section 3.2.3).
- ARC and WARC handlers can optionally load individual records from remotely hosted archives using HTTP requests with a "Range" header. This facility can be used with publicly-hosted data sets such as Common Crawl[2]. To support this functionality, document identifiers in a batch definition can now take XML attributes as well as the actual string identifier (exactly how such attributes are used is up to the handler implementations).
- Added output handler to save documents in a JSON format modelled on that used by Twitter to represent "entities" (e.g. username mentions) in Tweets.
- Efficiency improvements in the Mímir output handler, to send documents to the server in batches rather than opening a new HTTP connection for every document.

1.3.8 2.3 (November 2012)

- Now depends on GATE Embedded 7.1
- Introduced support for *conditional saving of documents* (section 3.3.3)
- Added the *serialized object output handler* (section 3.3.1)
- More robust and reliable counting of the size of each input document.

1.3.9 2.2 (February 2012)

- Now depends on GATE Embedded 7.0
- Introduced Java-based command line interface to replace the `gcp.sh` shell script, which behaves more consistently across platforms.

[1] `http://gate.ac.uk/gate/doc/javadoc/gate/creole/ControllerAwarePR.html`
[2] `http://www.commoncrawl.org`

Chapter 2

Installing and Running GCP

2.1 Installing GCP

Binary releases are available for release versions of GCP starting with version 2.5, as a ZIP file which can be downloaded from GitHub at `https://github.com/GateNLP/gcp/releases`[1]. For development versions the software must be built from source. The source code is available on GitHub at `https://github.com/GateNLP/gcp`.

To build GCP you will need a Java 8 JDK. Sun/Oracle and OpenJDK have been tested and are known to work. GCJ is known *not* to work. You will also need Apache Maven 3.3 or later. Running "mvn install" will build the various components of GCP and create a ZIP file containing the binary distribution under `distribution/target`. Unpack that file somewhere to create your GCP installation.

2.2 Running GCP

Once GCP is installed you can run it in one of two ways:

- using the `gcp-cli.jar` executable JAR file in the installation directory.
- using the `gcp-direct.sh` bash script.

2.2.1 Using `gcp-cli.jar`

The usual way to run GCP is to write one or more *batch definition* XML files (see chapter 3 for details) defining the application you want to run, the documents to process, and the output formats to produce. You then pass these batch definitions to `gcp-cli.jar` for processing. The CLI tool takes a number of optional arguments:

[1] Versions prior to 3.0 are available from SourceForge at `http://sf.net/projects/gate/files/gcp/`.

-m Specifies the maximum Java heap size, in the format expected by the usual `-Xmx` Java option, e.g. `-m 10G` for a 10GB heap limit. The default setting is `12G`. The `gcp-cli` will spawn a separate java process to run each batch, passing this memory limit to that process. This is different from specifying a `-Xmx` option to `gcp-cli`, which would define the heap size limit for the CLI process itself, not the batch runner processes it spawns.

-t Specifies the number of threads that GCP should use to execute the GATE application. Typically this should be set to between 1 and 1.5 times the number of processing cores available on the machine. The default value is 6, which is generally suitable for a 4-core machine.

-D Java system property settings, for example `-Djava.io.tmpdir=/home/bigtmp`. `-D` options specified before the `-jar` apply to the virtual machine running the CLI, those specified after `-jar gcp-cli.jar` will be passed to the batch runner processes. If you have an installed copy of GATE Developer you may wish to set `-Dgate.home=...` to point to your installation (*after* the `-jar`, as this is a setting that needs to apply to the batch runner VM). This is required if your saved GATE application refers to standard GATE plugins (using `$gatehome$` paths in the xgapp), but is optional if the application is self-contained – GCP includes its own copy of GATE Embedded and does not require a separate installed copy of the core libraries.

GATE plugins can also be pre-loaded using the `-C` and `-p` options, see the "gcp-direct" section below for details on these.

The tool will determine the location of where GCP is installed in the following order, taking the first it can find:

1. The value of the environment variable `GCP_HOME` if it is set.
2. The value of the propery `gcp.home` if it is set.
3. The location of the JAR file used for running the program.

Note that if the environment variable `GCP_HOME` is set to a different directory than the one used to run gcp-cli, the version of the batch runner in the directory pointed to by `GCP_HOME` will get invoked which is probably not what is intended.

The settings for the `-m` and `-t` options are typically a trade-off – if your application is particularly memory-hungry or you are processing particularly large or complex documents you may need to lower the number of processing threads in order to give more memory (on average) to each one.

GCP can run in two modes. In the basic "single-batch" mode the final command-line argument is simply the path to a single *batch definition* XML file (see chapter 3 for details), and GCP will process that batch and then exit.

The other (and more commonly used) mode is "multi-batch" mode, signified by the `-d` command line option. In this mode the final command-line argument is the path to a directory referred to as the *working directory*.

```
java -jar gcp-cli.jar -t 4 -m 8G -d /data/gcp
```

The working directory is expected to contain a subdirectory named "in", and any file in this directory with the extension `.xml` (in lower case) is assumed to be a batch definition file. For each batch *batch*`.xml` in the "in" directory, the script will:

- run the batch, redirecting the standard output and error streams to a file *working-dir*/logs/*batch*.xml.log
- if the batch completes successfully, move the definition file to *working-dir*/out/*batch*.xml
- or, if the batch fails (i.e. the Java process exits with a non-zero exit code, which occurs if, for example, one of the processing threads encounters an OutOfMemoryError), move the definition file to *working-dir*/err/*batch*.xml

Additional batches can be added to the "in" directory at any time – whenever a batch completes the script will re-scan the "in" directory to locate the next available batch. In particular, failed batches can be moved back from "err" to "in" and they will be re-processed, and if the report file for the failed batch is intact GCP will continue on from where it left off on the previous run.

Creating a file named `shutdown.gcp` in the "in" directory will cause the script to exit at the end of the batch it is currently processing (or immediately if it is currently idle).

2.2.2 Using `gcp-direct.sh`

The `gcp-direct.sh` script can be used for simple cases where you want to process all the files under one particular directory and output the resulting annotations in GATE XML or FastInfoset format. For this specific case it is not necessary to write an XML batch descriptor, you can specify the required parameters using command line options to `gcp-direct.sh`:

-t the number of parallel threads to use.

-x the path to the saved GATE application that you want to run.

-f the output format to use for saving results, must be either "xml" (GATE XML format) or "finf" (FastInfoset format). To use FastInfoset the GATE `Format_FastInfoset` plugin must be loaded by the saved application.

-i the directory in which to look for the input files or a file that contains relative path names to the input files. If this points to a directory, all files in this directory and any subdirectories will be processed (except for standard backup and temporary file name patterns and source control metadata – see `http://ant.apache.org/manual/dirtasks.html#defaultexcludes` for details). If this points to a file, the content of the file is expected to be one relative file path per line, using UTF-8 encoding. The file paths are interpreted to be relative to the directory that contains the list file. If processed documents are written, then this will also be their relative path to the output directory.

-o (optional) the directory in which to place the output files. Each input file will generate an output file with the same name in the output directory. If this option is missing, and the option `-b` is missing as well, the documents are not saved!

-b (optional) if this option is specified it can be used to specify a batch file. In that case, the options `-x, -i, -o, -r, -I` are not required and ignored if specified and the corresponding information is taken from the batch configuration file instead.

-r (optional) path to the report file for this batch – if omitted GCP will use `report.xml` in the current directory.

-ci the input files are all gzip-compressed

-co the output files should all be gzip-compressed. This only makes sense if `-f xml` is also specified since the default output format `finf` already is a compressed format. If this option is specified, the output file name gets the extension `.gz` appended, in addition to any other extension it already may have.

-p (optional, may be specified multiple times) a GATE plugin to pre-load in addition to those specified by the saved application. The value of this option can be one of three formats (tried in this order):

1. a set of Maven co-ordinates `group:artifact:version`, to load a plugin from a Maven repository.
2. an absolute URL e.g. starting `file:/...` or `http://`, to load a directory-based plugin from that URL.
3. a local file path, either absolute or relative to the working directory of the GCP process, to load a directory-based plugin from disk.

-C (optional, may be specified multiple times) when loading Maven-style plugins using `-p` GATE will typically go out to the internet to fetch the plugin and its dependencies from a remote Maven repository. If you have a local Maven cache of plugins on your disk you can specify its location with this option and the local cache will be searched first before attempting to download plugins from the network.

Additionally, you can specify `-D` and `-X` options which will be passed through to the Java VM, for example you can set the maximum amount of heap memory that the JVM can use with an option like `-Xmx2G`

The `gcp-direct.sh` script is deliberately opinionated, in order to reduce the number of different options that need to be set, and it has a number of hard-coded assumptions. It assumes that your input documents use the UTF-8 character encoding, that the correct document format parser to use can be determined from the file extension, and that you always want to save *all* the annotations that your application generates. If you need to process documents in a different encoding, you have more complex output requirements (XCES, JSON, Mímir, ...) or want to output only a subset of the GATE annotations from each document, then you should write a batch definition in XML and use `gcp-cli.jar` as discussed above.

Chapter 3

The Batch Definition File

3.1 The Structure of a Batch Descriptor

GCP batches are defined by an XML file whose format is as follows. The root element defines the batch identifier:

```
1  <batch id="batch-id" xmlns="http://gate.ac.uk/ns/cloud/batch/1.0">
```

The children of this `<batch>` element are:

application (required) specifies the location of the saved GATE application state. `<application file="../annie.xgapp"/>`

report (required) specifies the location of the XML report file. If the report file already exists GCP will read it and process only those documents that have not already been processed successfully. `<report file="../report.xml" />`

input (required) specifies the input handler which will be the source of documents to process. Most handlers load documents one by one based on their IDs, but certain handlers operate in a *streaming* mode, processing a block of documents in one pass.

output (zero or more) specified what to do with the documents once they have been processed.

documents (required, except when using a streaming input handler) specifies the document IDs to be processed, as any combination of the child elements:

id a single document ID. `<id>bbc/article001.html</id>`

documentEnumerator an enumerator that generates a list of IDs. The enumerator implementation chosen will typically depend on the specific type of input handler that the batch uses.

The following example shows a simple XML batch definition file which runs ANNIE and saves the results as GATE XML format. The input, output and documents elements are discussed in more detail in the following sections.

```
1  <?xml version="1.0" encoding="UTF-8"?>
```

```
<batch id="sample" xmlns="http://gate.ac.uk/ns/cloud/batch/1.0">
  <application file="../annie.xgapp"/>

  <report file="../reports/sample-report.xml" />

  <input dir="../input-files"
         mimeType="text/html"
         compression="none"
         encoding="UTF-8"
         class="gate.cloud.io.file.FileInputHandler" />

  <output dir="../output-files-gate"
          compression="gzip"
          encoding="UTF-8"
          fileExtension=".GATE.xml.gz"
          class="gate.cloud.io.file.GATEStandOffFileOutputHandler" />

  <documents>
    <id>ft/03082001.html</id>
    <id>gu/04082001.html</id>
    <id>in/09082001.html</id>
  </documents>
</batch>
```

It is important to note that all relative file paths specified in a batch descriptor are resolved against the location of the descriptor file itself, thus if this descriptor file were located at `/data/gcp/batches/sample.xml` then it would load the application from `/data/gcp/annie.xgapp`.

3.2 Specifying the Input Handler

Each batch definition must include a single `<input>` element defining the source of documents to be processed. Given a document ID, the job of the input handler is to locate the identified document and load it as a `gate.Document` to be processed by the application. Note that the input handler describes how to find the document for each ID but does *not* define which IDs are to be processed, that is the job of the `<documents>` element below.

The `<input>` element must have a `class` attribute specifying the name of the Java class implementing the handler. GCP will create an instance of this class and pass the remaining `<input>` attributes to the handler to allow it to configure itself. Thus, which attributes are supported and/or required depends on the specific handler class.

GCP provides four standard input handler types:

- `gate.cloud.io.file.FileInputHandler` to read documents from individual files on the filesystem
- `gate.cloud.io.zip.ZipInputHandler` to read documents directly from a ZIP archive
- `gate.cloud.io.arc.ARCInputHandler` and `gate.cloud.io.arc.WARCInputHandler` to read documents from an ARC or WARC archive as produced by the Heritrix web crawler (`http://crawler.archive.org`).

and one *streaming* handler:

- `gate.cloud.io.json.JSONStreamingInputHandler` to read a stream of documents from a single large JSON file (for example a collection of Tweets from Twitter's streaming API).

3.2.1 The `FileInputHandler`

`FileInputHandler` reads documents from individual files on the filesystem. It can read any document format supported by GATE Embedded, and in addition it can read files that are GZIP compressed, unpacking them on the fly as they are loaded. It supports the following attributes on the `<input>` element in the batch descriptor:

encoding (optional) The character encoding that should be used to read the documents (i.e. the value for the encoding parameter when creating a `DocumentImpl` using the GATE Factory). If omitted, the default GATE Embedded behaviour applies, i.e. the platform default encoding is used.

mimeType (optional) The MIME type that should be assumed when creating the document (i.e. the value of the `DocumentImpl` mimeType parameter). If omitted GATE Embedded will attempt to guess the appropriate MIME type for each document in the usual way, based on the file name extension and magic number tests.

compression (optional) The compression that has been applied to the files, either "none" (the default) or "gzip".

The actual mapping from document IDs to file locations is controlled by a *naming strategy*, another Java object which is configured from the `<input>` attributes. The default naming strategy (`gate.cloud.io.file.SimpleNamingStrategy`) treats the document ID as a relative path[1], and takes the following attributes:

dir (required) The base directory under which documents are found.

fileExtension (optional) A file extension to append to the document ID.

Given a document ID such as "ft/03082001", a base directory of "/data" and a file extension of ".html" the `SimpleNamingStrategy` would load the file "/data/ft/03082001.html"

To use a different naming strategy implementation, specify the Java class name of the custom strategy class as the `namingStrategy` attribute of the `<input>` element, along with any other attributes the strategy requires to configure it.

3.2.2 The `ZipInputHandler`

The ZIP input handler reads documents directly out of a ZIP archive, and is configured in a similar way to the file-based handler. It supports the following attributes:

encoding (optional) exactly as for `FileInputHandler`

[1] Technically a relative *URI*, so forward slashes must be used in document IDs even when running on Windows where file paths normally use backslashes.

mimeType (optional) exactly as for `FileInputHandler`

srcFile (required) The location of the ZIP file from which documents will be read. This parameter was previously named "zipFile", the old name is supported for backwards compatibility but not recommended for new batches.

fileNameEncoding (optional) The default character encoding to assume for file names inside the ZIP file. This attribute is only relevant if the ZIP file contains files whose names contain non-ASCII characters *without* the "language encoding flag" or "Unicode extra fields", and can be omitted if this does not apply. There is a detailed discussion on file name encodings in ZIP files in the Ant manual (`http://ant.apache.org/manual/Tasks/zip.html#encoding`), but the rule of thumb is that if the ZIP file was created using Windows "compressed folders" then `fileNameEncoding` should be set to match the encoding of the machine that created the ZIP file, otherwise the correct value is probably "Cp437" or "UTF-8".

The ZIP input handler does not use pluggable naming strategies, and simply assumes that the document ID is the path of an entry in the ZIP file.

3.2.3 The `ARCInputHandler` and `WARCInputHandler`

These two input handlers read documents out of ARC- and WARC format web archive files as produced by the Heritrix web crawler and other similar tools. They support the following attributes:

srcFile (optional) The location of the archive file[2]. These input handlers can operate in one of two modes – if `srcFile` *is* specified then the handler will load records from this specific archive file on disk, but if `srcFile` is *not* specified then each document ID must provide a fully qualified http or https URL to an archive. In the second mode the selected records will be downloaded individually using "byte range" HTTP requests.

defaultEncoding (optional) The *default* character encoding to assume for entries that do not specify their encoding in the entry headers. If an entry specifies its own encoding explicitly this will be used. If this attribute is omitted, "Windows-1252" is assumed as the default.

mimeType (optional) The MIME type that should be assumed when creating the document (i.e. the value of the `DocumentImpl` mimeType parameter). If omitted, the usual GATE Embedded heuristics will apply. The input handlers make the HTTP headers from the archive entry available to GATE as if the document had been downloaded directly from the web, so the `Content-Type` header from the archive entry is available to these heuristics.

The web archive input handlers expect document IDs of the following form:

```
<id recordPosition="NNN" [url="optional url of archive"]
    recordOffset="NNN" recordLength="NNN">{original entry url}</id>
```

[2]For ARC, this parameter was previously called "arcFile", the old name is supported for backwards compatibility but not recommended for new batches.

The content of the `id` element should be the original URL from which the entry was crawled, and the attributes are:

recordPosition a numeric value that is used as a sequence number. If the IDs are generated by the corresponding enumerator (see below), then the this attribute will contain the actual record position inside the archive file.

recordOffset and recordLength the byte offset of the required record in the archive, and the record's length in bytes.

url (optional) a full HTTP or HTTPS URL to the source archive file. If this is provided, GCP will download just the specific target record using a "Range" header on the HTTP request, rather than loading the record from the input handler's usual `srcFile`.

The standard enumerator implementations (see below) create IDs in the correct form.

The ARC input handler adds all the HTTP headers and archive record headers for the entry as features on the GATE `Document` it creates. HTTP header names are prefixed with "http_header_" and ARC/WARC record headers with "arc_header_".

3.2.4 The streaming JSON input handler

An increasing number of services, most notably Twitter and social media aggregators such as DataSift, provide their data in JSON format. Twitter offers streaming APIs that deliver Tweets as a continuous stream of JSON objects concatenated together, DataSift typically delivers a large JSON array of documents. The streaming JSON input handler can process either format, treating each JSON object in the "stream" as a separate GATE document.

The `gate.cloud.io.json.JSONStreamingInputHandler` accepts the following attributes:

srcFile the file containing the JSON objects (either as a top-level array or simply concatenated together, optionally separated by whitespace).

idPointer the "path" within each JSON object of the property that represents the document identifier. This is an expression in the *JSON Pointer*[3] language. It must start with a forward slash and then a sequence of property names separated by further slashes. A suitable value for the Twitter JSON format would be `/id_str` (the property named "`id_str`" of the object), and for DataSift `/interaction/id` (the top-level object has an "interaction" property whose value is an object, we want the "id" property of *that* object). Any object that does not have a property at the specified path will be ignored.

compression (optional) the compression format used by the `srcFile`, if any. If the value is "none" (the default) then the file is assumed not to be compressed, if the value is one of the compression formats supported by Apache Commons Compress ("gz"[4], "bzip2", "xz", "lzma", "snappy-raw", "snappy-framed", "pack200", "z") then it will be unpacked using that library. If the value is "any" then the handler uses the auto-detection capabilities of Commons Compress to attempt to detect the appropriate compression format. Any other

[3] `http://tools.ietf.org/html/draft-ietf-appsawg-json-pointer-03`

[4] For backwards compatibility, "gzip" is treated as an alias for "gz"

value is taken to be the command line for a native decompression program that expects compressed data on stdin and will produce decompressed data on stdout, for example `"lzop -dc"`.

mimeType (optional but highly recommended) the value to pass as the "mimeType" parameter when creating a GATE Document from the JSON string. This will be used by GATE to select an appropriate document format parser, so for Twitter JSON you should use `"text/x-json-twitter"` and for DataSift `"text/x-json-datasift"`. Note that the GATE plugin defining the relevant format parser *must* be loaded as part of your GATE application.

This is a streaming handler – it will process all documents in the JSON bundle and does *not* require a `documents` section in the batch specification. As with other input handlers, when restarting a failed batch documents that were successfully processed in the previous run will be skipped.

3.3 Specifying the Output Handlers

Output handlers are responsible for taking the GATE Documents that have been processed by the application and doing something with the results. GCP supplies a number of standard output handlers to save the document text and annotations to files in various formats, and also a handler to send the annotated documents to a remote Mímir server for indexing.

Most batches would specify at least one output handler but GCP does support batches with no outputs (if, for example, the application itself contains a PR responsible for outputting results).

Output handlers are specified using `<output>` elements in the batch definition, and like input handlers these require a `class` attribute specifying the implementing Java class name. Other attributes are passed to the instantiated handler object to allow it to configure itself.

By default, an output handler will save all annotations from all annotation sets in each document. A given output handler may be configured to save only a subset of the annotations by providing `<annotationSet>` sub-elements inside the `<output>` element, for example

```
1 <annotationSet name="ANNIE">
2   <annotationType name="Person" />
3   <annotationType name="Location" />
4 </annotationSet>
5
6 <annotationSet />
```

The `<annotationSet>` element may have a `name` attribute giving the annotation set name (if omitted the default annotation set is used), and zero or more `<annotationType>` sub-elements giving the annotation types to extract from that set (if no `<annotationType>` elements are provided, all annotation from the set are saved, so line 6 specifies that the handler should save all annotations from the default set).

Note that these filters are provided as a convenience, and some output handler implementations may ignore them. For example the Mímir output handler always sends the complete Document to the Mímir server, regardless of the filters specified.

3.3.1 File-based Output Handlers

GCP provides a set of six standard file-based output handlers to save data to files on the filesystem in various formats.

- `gate.cloud.io.file.GATEStandOffFileOutputHandler` to save documents in the GATE XML format ("save as XML" in GATE Developer).
- `gate.cloud.io.file.GATEInlineOutputHandler` to save documents with inline XML tags for their annotations ("save preserving format" in GATE Developer).
- `gate.cloud.io.file.PlainTextOutputHandler` to save just the text content of the document. This is rarely useful on its own but is frequently used in conjunction with
- `gate.cloud.io.xces.XCESOutputHandler` to save annotations in the XCES standoff format. Annotation offsets in XCES refer to the plain text as saved by a `PlainTextOutputHandler`.
- `gate.cloud.io.file.JSONOutputHandler` to save documents in a JSON format modelled on that used by Twitter to represent "entities" in Tweets.
- `gate.cloud.io.json.JSONStreamingOutputHandler` saves documents in the same JSON format as the previous handler, but concatenated together in one or more output batches rather than saving each document in its own individual output file.
- `gate.cloud.io.file.SerializedObjectOutputHandler` to save documents using Java's built in *object serialization* protocol (with optional compression). This handler ignores annotation filters, and always writes the complete document. This is the same mechanism used by GATE's `SerialDataStore`.

The handlers share the following `<output>` attributes:

encoding (optional, not applicable to `SerializedObjectOutputHandler`) The character encoding used when writing files. If omitted, "UTF-8" is the default.

compression (optional) The compression algorithm to apply to the saved files. Can be either "none" (no compression, the default) or "gzip" (GZIP compression).

As with the file-based input handler, these output handlers use a *naming strategy* to map from document IDs to output file names. The default strategy is the same `SimpleNamingStrategy` configured with a base `dir` and a `fileExtension`, treating the document ID as a path relative to the given directory and appending the given extension. If the `replaceExtension` parameter is set to `"true"` then the `fileExtension`, if specified, replaces any existing file extension of the intput path.

This is appropriate when using a file or ZIP input handler but for batches that use an `ARCInputHandler` a different strategy is required.

As document IDs for an `ARCInputHandler` are based on URLs the simple strategy would try to put the output files into directories named after absolute URLs, which can include characters that are not permitted in file names on all platforms. An alternative strategy is provided that makes use of the `recordPosition` attribute on the IDs to put output files into a hierarchy of numbered directories. To use this strategy, specify an attribute `namingStrategy="gate.cloud.io.arc.ARCDocumentNamingStrategy"`, and the usual `dir` and `fileExtension` attributes of the default strategy. The ARC strategy also accepts an optional additional attribute `pattern` defining the pattern to use to map the ID number to a directory.

The default pattern is "3/3", which will left-pad the `recordPosition` to a minimum of 6 digits and then create one level of directories from the first three digits and use the last three as part of the file name[5]. The ID text (i.e. the original URL) is cleaned up to remove the protocol, query string and fragment (if any) and replace slash and colon characters with underscores (so the resulting file name will not include any more levels of subdirectories) and appended to the numeric part following an underscore. For full details of this process, see the JavaDoc documentation. As an example, the ID with `recordPosition="1"` and URL `http://example.com/file.html` with the default pattern of "3/3" would map to the target path "000/001_example.com_file.html", and this would then be combined with the `dir` and `fileExtension` to produce the final file name.

The `PlainTextOutputHandler` simply saves the plain text of the GATE document with no annotations (so `<annotationSet>` filters are ignored). The `GATEStandOffFileOutputHandler` writes the document text and selected annotations in the standard "save as XML" GATE XML format. The `XCESOutputHandler` saves the selected annotations as XCES XML format.

The `GATEInlineOutputHandler` saves the document text plus selected annotations as inline XML tags as produced by "save preserving format" in GATE Developer. This handler supports one additional `<output>` attribute named `includeFeatures` – if this is set to "true", "yes" or "on" then the annotation features will be included as attributes on the XML tags, otherwise (including if the attribute is omitted) it will save just the tags with no attributes.

The `JSONOutputHandler` saves the document in a JSON format modelled on that used by Twitter to represent entities in Tweets. This is a JSON object with two properties, "text" holding the plain text of the document and "entities" holding the annotations. The "entities" value is itself an object mapping a "label" to an array of annotations.

```
{
  "text":"The text of the document",
  "entities":{
    "Person":[
      {
        "indices":[start,end],
        "feature1":"value1",
        "feature2":"value2"
```

[5]In fact the pattern is processed from right to left, so any surplus digits end up in the first place, i.e. the ID 1234567 becomes 1234/567 rather than 123/4567.

```
      },
      {
        "indices":[start,end],
        "feature1":"value1",
        "feature2":"value2"
      }
    ]
  }
}
```

For each annotation the "indices" property gives the start and end offsets of the annotation as character offsets into the "text", and the other properties of the object represent the features of the annotation.

This handler supports a number of additional `<output>` attributes to control the format.

groupEntitiesBy controls how the annotations are grouped under the "entities" object. Permitted values are "type" (the default) or "set". Grouping by "type" produces output like the example above, with one entry under "entities" for each annotation type containing all annotations of that type from across all annotation sets that were selected by the `<annotationSet>` filters. Conversely, grouping by "set" creates one entry under "entities" for each annotation set name (with the name "default" used for the default annotation set – technically JSON permits the empty string as a property name but this is likely to cause problems for some consumer libraries), containing all the annotations in that set that were selected by the filters, regardless of type. Grouping by "set" will often be used in combination with the "annotationTypeProperty" attribute.

annotationTypeProperty if set, the type of each annotation is added to the output as this property (i.e. treated as if it were an additional feature of the annotation). This is useful in combination with `groupEntitiesBy="set"` when different types of annotation are grouped under a single label.

documentAnnotationASName the annotation set in which to search for a *document annotation* (see below). If omitted, the default set is used.

documentAnnotationType if specified, the output handler will look for a single annotation of this type within the specified annotation set and assume that this annotation spans the "interesting" portion of the document. Only the text and annotations covered by this annotation will be output, and furthermore the features of the document annotation will be added as top-level properties (alongside "text" and "entities") of the generated JSON object. This option is intended to support round-trip processing of documents that were originally loaded from JSON by GATE's Twitter support.

The `JSONStreamingOutputHandler` writes the same JSON format, but instead of storing each GATE document in its own individual file on disk, this handler creates one large file (or several "chunks") and writes documents to this file in one stream, separated by newlines. In addition to the parameters described above this handler adds two further parameters:

pattern (optional, default `part-%03d`) the pattern on which chunk file names should be created. This is a standard Java `String.format` pattern string

which will be instantiated with a single integer parameter, so should include a single `%d`-based placeholder. Output file names are generated by instantiating the pattern with successive numbers starting from 0 and passing the result to the configured naming strategy until a file name is found that does not already exist. With the default naming strategy this effectively means `{dir}/{pattern}{fileExtension}`, e.g. `output/part-003.json.gz`

chunkSize (optional, default `99000000`) approximate maximum size in bytes of a single output file, after which the handler will close the current file and start the next chunk. The file size is checked after every MB of uncompressed data, so each chunk should be no more than 1MB larger than the configured chunk size. The default chunkSize is 99 million bytes, which should produce chunks of no more than 100MB.

This handler, like the `JSONStreamingInputHandler` can cope with a wider variety of compression formats than the standard one-file-per-document output handlers. A value other than "none" or "gzip" for the "compression" parameter will be taken as the command line for a native compression program that expects raw data on its stdin and produces compressed data on stdout, for example `"bzip2"` or `"lzop"` (with the default naming strategy, the configured fileExtension should take the compression format into account, e.g. `".json.lzo"`).

3.3.2 The Mímir Output Handler

GCP also provides `gate.cloud.io.mimir.MimirOutputHandler` to send annotated documents to a Mímir server for indexing. This handler supports the following `<output>` attributes:

indexUrl (required) the *index URL* of the target index. See the Mímir documentation for details.

uriFeature (optional) Mímir requires a URI to identify each document. This attribute tells GCP that the URI for a document should be taken from the document feature with this name.

namespace (optional) if `uriFeature` is not specified GCP will construct a suitable URI by appending the document ID to a fixed "namespace" string. If omitted an empty namespace will be used (i.e. the URI passed to Mímir will be just the document ID).

username (optional) HTTP basic authentication username to pass to the Mímir index. If omitted, no authentication token will be passed.

password (required if and only if username is specified) the corresponding basic authentication password.

connectionInterval the default behaviour of the Mímir output handler is to open a new connection for each document. When processing very short documents, such as tweets, or paper abstracts, using many parallel threads, it is possible that new connections will be opened several hundred times a second. This has the potential to overload the receiving Mímir server, or to trigger some security measures, leading to refused connections. To avoid this, the output handler can be configured to accumulate the processed documents in memory, and only connect to the remote server from time to time, sending all the pending documents each time. To enable this functionality, set the value

for the `connectionInterval` to a positive value representing the number of milliseconds between connections. For example, a setting of `1000` means that a new connection will be opened every second.

This handler ignores annotation set filters – the complete document will be sent to Mímir.

3.3.3 Conditional Output

All output handlers support conditional output: the option to only save some of the documents, based on the value of a document feature. To make use of this facility, you need to specify which document feature should be read when deciding whether or not to save a given document, by adding an attribute named `conditionalSaveFeatureName` to the `output` XML tag in the batch definition, like in the following example:

```
<output conditionalSaveFeatureName="save"
    dir="../output-files-gate"
    compression="gzip"
    encoding="UTF-8"
    fileExtension=".GATE.xml.gz"
    class="gate.cloud.io.file.GATEStandOffFileOutputHandler" />
```

In this example, for each processed document, a document feature named `save` will be sought. If this is found, and if its value is logical true (i.e. the feature value is a String with the content '*true*', '*yes*', or '*on*', regardless of case) then the document will be saved, otherwise it will be ignored.

3.4 Specifying the Documents to Process

If you are not using a streaming input handler then the final section of the batch definition specifies which document IDs GCP should process. The IDs can be specified in two ways:

- Directly in the XML as `<id>doc/id/here</id>` elements.
- By defining a *document enumerator* which generates a list of IDs from some external source.

GCP provides document enumerator implementations corresponding to the default input handlers, so a typical batch with a ZIP input handler, for example, would use a ZIP enumerator (configured for the same ZIP file) to generate the list of document IDs.

A document enumerator is configured using a `<documentEnumerator>` XML element inside the `<documents>` element. As with input and output handlers, this element requires a `class` attribute specifying the Java class of the enumerator implementation and other attributes are handed off to the enumerator object to configure itself.

3.4.1 The File and ZIP enumerators

The default enumerator implementation corresponding to the file and ZIP input handlers are closely related to one another.

The `gate.cloud.io.file.FileDocumentEnumerator` takes a `dir` attribute and the `gate.cloud.io.zip.ZipDocumentEnumerator` takes `srcFile` and `fileNameEncoding` attributes (as described above for their corresponding input handlers) specifying where to find the directory or ZIP file to be enumerated. To define which files (or ZIP entries) to enumerate, the enumerators use the "fileset" abstraction from Apache Ant, controlled by the following attributes:

includes (optional) comma-separated file name patterns specifying which files to include in the search, e.g. `"**/*.xml,**/*.XML"`. If omitted, all files or ZIP entries are included.

excludes (optional) comma-separated file name patterns specifying which files to exclude from the search, e.g. `"**/*.ignore.xml"`. If omitted, nothing is excluded.

defaultExcludes (optional) Ant filesets exclude certain file patterns by default (`http://ant.apache.org/manual/dirtasks.html#defaultexcludes`), and the GCP enumerators behave likewise. To *disable* the default excludes, set this attribute to "off" or "no".

prefix (optional) a prefix to prepend to the paths that are returned by the fileset. For example, if a batch has a file input handler pointing to the directory `/data` and an enumerator pointing to the directory `/data/large` then the enumerator would need a prefix of "large/" to produce IDs that are meaningful to the input handler.

See the Ant documentation for full details on the include and exclude patterns supported by filesets. The IDs returned by the enumerator will be those that match at least one of the include patterns and also do not match any of the exclude patterns.

Note also that include and exclude patterns are *case sensitive*, so a pattern of "*.xml" would not match "FILE.XML", for example. To match both upper and lower-case variants, include both forms in the pattern.

3.4.2 The ARC and WARC enumerators

The `gate.cloud.io.arc.ARCDocumentEnumerator` and `WARCDocumentEnumerator` classes enumerate entries in an ARC or WARC file, and would typically be used in conjunction with the corresponding input handler. The enumerators support the following attributes:

srcFile (required) the path to the archive to enumerate.

mimeTypes (optional) whitespace-separated list of MIME types. If specified, the enumerator will only include entries in the archive whose header specifies one of the given MIME types. So a value of `"text/html application/pdf"` would enumerate only HTML and PDF files from the archive.

includeStatusCodes (optional) Each entry in an archive file records the HTTP status code (200, 301, 404, etc.) that was returned by the server when the item was crawled. This attribute gives a regular expression that is matched against the status codes that should be included in the enumeration. If omitted, all status codes are included (except those excluded by `excludeStatusCodes`).

excludeStatusCodes (optional) regular expression giving the status codes that should be excluded from the enumeration. If *both* `includeStatusCodes` *and* `excludeStatusCodes` are omitted, the default behaviour is to assume an exclude pattern of `[345].*` (i.e. omit all 3xx, 4xx and 5xx status codes).

The enumerators returns document IDs in the form required by the corresponding handlers:

```
<id recordPosition="{zero-based index into the archive}"
    recordOffset="{byte offset of the start of this record}"
    recordLength="{length of the record in bytes}"
    >{original URL from which the document was crawled}</id>
```

This format is designed to work well in combination with the `ARCDocumentNamingStrategy` for file-based output handlers.

3.4.3 The `ListDocumentEnumerator`

`gate.cloud.io.ListDocumentEnumerator` is the final enumerator implementation provided by default, and it simply reads a list of document IDs from a plain text file, one ID per line. It supports the following attributes:

file (required) the location of the text file.

encoding (optional) the character encoding of the file. If omitted, the platform default encoding is assumed.

prefix (optional) a common prefix to prepend to each ID (see the file-based enumerator for an example of this).

This enumerator treats each line of the specified file as a separate document ID, ignoring leading and trailing whitespace on the line. It is intended for use when there is a separate process (such as a Perl script) that generates the list of IDs in advance.

Chapter 4

Extending GCP

GCP has been designed to be easily extensible. Additional input/output handlers, naming strategies and document enumerators can be added by writing a Java class that implements the relevant interface and placing that class in a CREOLE plugin loaded by your application. The class can then be named in your batch definitions as appropriate, and GCP will load it from the GATE ClassLoader. The interfaces and abstract base classes required for this are distributed in the `gcp-api` JAR file which is available in Maven Central (for releases) or the GATE Maven repository (for snapshots), plugins that wish to offer GCP handlers should depend on `gcp-api` using the "provided" scope, since the classes will be available on the system classpath when running GCP.

4.1 Custom Input Handlers

Input handlers must implement the `gate.cloud.io.InputHandler` interface:

```
1  public void config(Map<String, String> configData) throws
       IOException, GateException;
2  public void init() throws IOException, GateException;
3  public void close() throws IOException, GateException;
4
5  public Document getInputDocument(String id) throws IOException,
       GateException;
```

The handler class must also have a no-argument constructor. When GCP parses the batch definition it creates an instance of the handler class using that constructor, then calls the `config` method, passing in a map containing the XML attribute values from the `<input>` element in the batch file. An additional "virtual" attribute named `batchFileLocation`[1] is also available in this map, containing the path to the XML batch definition file itself. This is made available to allow the handler to resolve relative path expressions in other attribute values against the location of the XML file (see the standard handler implementations for examples of this). The `config`

[1] Available as a constant `gate.cloud.io.IOConstants.PARAM_BATCH_FILE_LOCATION`

method should read any required parameters from the config map, and should throw a suitably descriptive `GateException` if any required parameters are missing.

Next, GCP will call the handler's `init()` method. This method is responsible for setting up the handler, and is where you should open any required external resources such as index files. Both `config` and `init` are guaranteed to be called sequentially in a single thread.

Conversely, `getInputDocument` will be called by the processing threads, and must therefore be thread-safe. However you should avoid locking or `synchronized` blocks within `getInputDocument` if at all possible, so as not to block processing threads against one another. This method is the one responsible for actually loading the GATE Document corresponding to a given document ID. The handler does not need to (indeed should not) retain a reference to the document that it returns, as the processing thread is responsible for freeing the document with `Factory.deleteResource` when processing is complete.

Finally, the `close` method will be called (in a single thread) when the entire batch is complete. This is where you should release resources that were acquired in the `init` method.

It is good manners, though not strictly required, for input handlers to provide a meaningful `toString()` implementation.

4.2 Custom Output Handlers

Output handlers must implement the `gate.cloud.io.OutputHandler` interface:

```
1 public void setAnnSetDefinitions(List<AnnotationSetDefinition>
      annotationsToSave);
2 public List<AnnotationSetDefinition> getAnnSetDefinitions();
3
4 public void config(Map<String, String> configData) throws
      IOException, GateException;
5 public void init() throws IOException, GateException;
6 public void close() throws IOException, GateException;
7
8 public void outputDocument(Document document, DocumentID
      documentId) throws IOException, GateException;
```

GCP will instantiate the handler class and call `config` exactly as for input handlers (see section 4.1). It will then call `setAnnSetDefinitions` to pass in the annotation set definitions specified by the `<annotationSet>` elements in the XML (if any), and then call `init`. As before these three methods are called in sequence in a single thread.

As documents are processed, the various processing threads will call `outputDocument`, passing an annotated document as a parameter. This method must therefore be thread-safe, but should avoid synchronization and locking if at all possible.

Finally, the `close` method will be called (in a single thread) when the entire batch is complete. This is where you should release resources that were acquired in the

`init` method.

It is good manners, though not strictly required, for output handlers to provide a meaningful `toString()` implementation.

GCP provides an abstract base class `gate.cloud.io.AbstractOutputHandler` which custom output handler implementations may choose to extend. This class provides an implementation of the `get` and `setAnnSetDefinitions` methods, a utility method to extract the annotations corresponding to these definitions from an annotated document at output time, and support for the `conditionalSaveFeatureName` config parameter, to only save documents that have a specified feature. In order to support this, the `config` and `outputDocument` methods are `final`, and subclasses provide the corresponding methods `configImpl` and `outputDocumentImpl`.

For output handlers that write to files on disk, there is another abstract base class `gate.cloud.io.file.AbstractFileOutputHandler` that configures a `NamingStrategy`, and provides an additional convenience method for subclasses to get an output stream for the file corresponding to a document ID according to the naming strategy.

4.3 Custom Naming Strategies

Custom naming strategies (for use with the file-based input and output handlers) must implement the `gate.cloud.io.file.NamingStrategy` interface:

```
1  public void config(boolean isOutput, Map<String, String>
       configData) throws IOException, GateException;
2
3  public File toFile(DocumentID id) throws IOException;
```

The `config` method is similar to the equivalent method on input and output handlers, but it has an extra parameter indicating whether this naming strategy is being used by an input (false) or output (true) handler. This allows the strategy to adjust its behaviour as appropriate, for example in the input case it is an error if the specified base directory does not exist, whereas for output this is permitted as the output handler will create intermediate directories as required.

The `toFile` method will be called from processing threads by the input (or output) handler that uses this strategy, and should return the `java.io.File` that corresponds to the given document ID.

4.4 Custom Document Enumerators

Document enumerators must implement the `gate.cloud.io.DocumentEnumerator` interface:

```
1  public void config(Map<String, String> configData) throws
       IOException, GateException;
```

```
2  public void init() throws IOException, GateException;
3
4  // DocumentEnumerator extends java.util.Iterator<DocumentID>
5  public boolean hasNext();
6  public DocumentID next();
7  public void remove();
```

They have the familiar `config` and `init` methods which are called as for input and output handlers (see section 4.1). To actually enumerate documents GCP uses the standard `Iterator<DocumentID>` methods, the enumerator is expected to return one ID from each call to `next`. The `remove` method is specified by the Iterator interface but will never be called by GCP, the standard enumerators all implement this method to throw an `UnsupportedOperationException`.

Chapter 5

Advanced Topics

5.1 GATE Configuration

By default GCP uses its own GATE user configuration file, located in the `gate-home` directory under the GCP installation root directory. It deliberately does not use the user's own `~/.gate.xml`, so if you need to configure any options that must be set through configuration files you will need to edit `gate-home/user-gate.xml`. The most likely configuration option that you will need to set this way is whether or not to add additional spaces to separate otherwise adjacent text in different XML or HTML elements when unpacking markup. This is the `Document_add_space_on_unpack` option in `user-gate.xml`.

If you want to use your own user configuration you can do so by setting the `gate.user.config` system property.

5.2 JMX Monitoring

The GCP batch runner registers an MBean with the platform JMX MBean server in its JVM that makes it possible to query the state of the running batch from the JMX management console or (using the standard JMX APIs) from another Java process. The process of connecting a JMX client to the GCP process is beyond the scope of thie guide, here we simply describe the MBean interface and the attributes it exposes to clients.

The MBean implements the `gate.cloud.batch.BatchJobData` interface which is defined in the `gcp-api` JAR file[1]:

```
1  public JobState getState();
2  public int getProcessedDocumentCount();
3  public int getTotalDocumentCount();
4  public int getRemainingDocumentCount();
5  public int getSuccessDocumentCount();
```

[1] If your application requires this dependency solely to be able to communicate with a running GCP over JMX then you can safely exclude the transitive dependency on `gate-core`.

```
6  public int getErrorDocumentCount();
7  public long getStartTime();
8  public String getBatchId();
9  public long getTotalDocumentLength();
10 public long getTotalFileSize();
```

The `getState()` method returns the current state of the batch. The state is an enum type, and will usually be `JobState.RUNNING`. The various `get*DocumentCount` methods provide access to the number of documents that have been processed, the number of those that were successful/failed and the number remaining to be processed. This allows the monitoring process to check that the GCP process is not "stuck". The `getTotalDocumentLength` and `getTotalFileSize` methods give the total length of the documents processed (in plain text characters, after GATE has unpacked the markup) and the total size (in bytes) of the files processed so far.