Jobe: REST API

Version 0.3 25 May 2014

Richard Lobb

1. Introduction

This document proposes a RESTful API for sending small student-exercise-type jobs to a "Job Engine" server, called *Jobe*. A job consists of a source program in a specified language together with possible additional files to be placed in the same directory as the source program. Jobe compiles and/or executes the given job and returns the status of the job plus any output generated. This API is similar in general intent to that of the *Ideone* API: see https://ideone.com/files/ideone-api.pdf. The major difference is that it is REST based rather than SOAP based. Also, it supports the uploading of additional support files, which may be either extra code, such as classes or modules, or may be run-time test data.

Jobe has been developed for use in the Moodle *CodeRunner* question-type plug-in (see https://github.com/trampgeek/CodeRunner). In this context, the Jobe server is expected to be a custom server behind the Moodle institutional firewall and Jobe will receive requests only from the Moodle server by means of firewalling of the server or other IP whitelisting mechanisms. In more general contexts, authentication and/or authorization can be enforced by higher-level protocols that are not part of this API specification. For example, the current implementation of Jobe uses Ellis Lab's CodeIgniter framework (see http://codeigniter.com) plus the REST-server extension written by Phil Sturgeon (see https://github.com/philsturgeon/codeigniter-restserver). The latter provides for Basic or Digest HTTP authentication, IP whitelisting and an API-key mechanism that requires all HTTP requests to include an X-API-KEY header and a known authorisation key. A Jobe administrator can enforce any combination of these mechanisms on top of the API described in this document.

2. Requests

The following table lists all the REST requests. The meanings of the possible response codes are documented in section 3. All POST and PUT requests have a content type of application/json; the request body is a json encoded object in which the fields are the parameters specified in the table.

Request Name	HTTP Type	Target Resource	Parameters (* denotes required)	Possible response codes	Comment
submit_run	POST	/runs	run_spec*	200 OK 202 Accepted 400 BadRequest 404 Not Found	A return code of 200 is accompanied by the run result; 202 denotes the job has been queued for later execution (notes 1, 2, 6).
get_run_status	GET	/runresults/id		200 OK 204 NoContent 400 BadRequest 404 NotFound	A return code of 200 is accompanied by the run result; 204 denotes the job is still pending (notes 2, 6).
get_languages	GET	/languages		200 OK 400 BadRequest	Returns a JSON encoded list of supported languages (note 3).
put_file	PUT	/files/uniqueid	file_contents*	204 No content 400 BadRequest 403 Forbidden	It is the client's responsibility to ensure a unique file ID (notes 4, 6, 7)

Requests (cont'd)

Request Name	HTTP Type	Target Resource	Parameters (* denotes required)	Possible response codes	Comment
post_file	POST	/files	file_contents*	200 OK 400 BadRequest	Add a file to the collection and get back a unique file identifier (notes 4, 6, 7)
check_file	HEAD	/files/uniqueid		204 No content 400 BadRequest 404 NotFound	Used by the client to see if the server (still) holds a particular file (note 4).

Notes:

1. The mandatory *run_spec* parameter is a JSON-encoded job record with the following allowed fields (* denotes a mandatory field/key).
    - language_id* - the computer language ID of this particular run (see note 3)
    - sourcecode* - the program to compile and/or run
    - sourcefilename* - the name to assign to the source code file in the run directory
    - input - the standard input data, if required
    - file_list - a list of (file_id, file_name) pairs, specifying which files should be loaded into the run-time directory. The file_id is the unique file identifier as supplied in *put_file* and file_name is the name assigned to that file when it is loaded. [As a possible extension, a triple of (file_id, file_name, is_source) might be permitted instead of a pair; the third parameter is true if the specified file is a program source file that should be included in the compile-and-build operation in a language-dependent manner.] File identifiers are required to be purely alphanumeric and at least 8 characters in length. Filenames must be made only from alphanumeric characters plus '-', '_' and '.'
    - parameters - a server-dependent set of (key, value) pairs (i.e. a JSON record), which might include things like maximum execution time, maximum memory usage, compile flags, ...
    - debug - if provided and true, the server is invited to include extra debugging information in the response or to retain extra information itself for later inspection. Server dependent.
2. The API allows for either an immediate run of a submitted job, returning the run results in the response or a deferred run in which the server simply accepts the job and enqueues it for later execution. The server has discretion over which of these modes to use and the client must respect that choice, polling for a result if the run submission returns 202 Accepted rather than 200 OK. The former mode is preferred for fast turn-around jobs on relatively lightly-loaded servers while the latter would be required under high load. The server may choose to use a mix of the two depending on load and/or expected maximum execution time.
3. The returned language list is a JSON-encoded list of (language_id, language_version) pairs of all supported languages. The client must use one of the returned language_ids in all its run submissions. The formats of the language_id and language_version are server-dependent: they are just strings. However, they should preferably be human-readable, e.g. ('C99', 'gcc 4.8.1'), to allow clients to present the language list to on-line users. The language_id must be unique; if multiple versions of a language are supported, each must have its own ID.
4. The use of PUT to place a specific file on the server is not satisfactory in a shared-server context where the clients might not be trusted to provide a globally unique file id. If a server does not trust the sender it should return 403 Forbidden and the client must then fall back to using *post-file* instead.
5. Ideally, if the get_status request is to be idempotent, the server should keep run results for some

"reasonable" time. However, the holding time from when the results are first returned to the client to when they are discarded from the server is at the discretion of the server, and may be zero.

6. The file interface supports server caching of support files, which might be large test data files that one does not wish to upload multiple times. In the most typical use-case, where there is a single trusted client, the client will PUT each file with a client-specified file_id. In a Moodle context, the file_id might be its MD5 hash, for consistency with the Moodle file API. The server is required to keep a copy of the file for at least some minimum time, sufficient for the file to be used in a subsequent run. In such a context the client might first query the server, using the *check_file* request, to see which files it needs to upload for a new run. It can then PUT all the required files to the server and submit the run. An alternative approach would be to simply submit the run and then, if a 404 response is obtained, try again after PUTing all the support files. Yet a third approach would be to PUT all the required files regardless before each *submit_run* request.
7. The *file_contents* parameter to *put_file* and *post_file* requests is a string containing the file contents (which might be binary) encoded in standard base_64. The server decodes the *file_contents* before writing it to the file cache. If the contents are not a valid base-64 encoding, 400 Bad Request is returned.

3. Meanings of the different response codes

This section describes the meanings of the various possible HTTP response codes returned by the requests listed in section 2. *405 MethodNotAllowed* is the only response code not explicitly listed in section 2, because it is the one issued when a request does not match any of the specified requests.

1. 200 OK is returned by a successful request that is accompanied by some response data. Not all successful requests return 200 OK, as follows. *submit_run* returns 200 together with the full run result as in section 4 below if the run can be done immediately, but returns 202 Accepted together with a run_id if the run if queued for later exection. *get_run_status* returns 200 OK plus the run result if the job is complete but returns 204 No Content if the job is still queued. *put_file* returns 204 No Content if the creation or update of the specified file is successful.
2. 202 Accepted is returned by the *submit_run* request if the server has enqueued the job for subsequent execution rather than running it immediately. The response data is then a unique run_id for use in subsequent *get_run_status* requests.
3. 204 NoContent is returned by the *get_run_status* request if the request relates to a pending job for which the result is not yet available. It's also returned by *check_file* if the file exists and by a successful *put_file* .
4. 400 BadRequest is returned by any request that has syntactic or semantic errors in any of the parameters or is missing a required parameter.
5. 403 Forbidden is returned by *put-file* if the server does not trust the client to provide a unique file ID, in which the client must use the *post-file* command instead, where the server provides file IDs.
6. 404 NotFound is returned by any request to a resource collection other than those listed or by a request for a specific unknown resource or by a submit_run request containing unknown file ids.
7. 405 Method Not Allowed is returned by any request to a resource or resource collection that uses a method that is not defined for that re *RunResult* source (collection).
8. Other return codes, such as 401 Unauthorized, by also be returned if higher-level authentication and authorisation protocols are enabled, as briefly discussed in the introduction.

4. The *RunResult* object

The data returned with a 200 OK response to either a *submit_run* or *get_run_status* request is a JSON-encoded record with four fields, as follows:

1. *run_id* - the unique ID of this particular run (which may or may not be usable in a subsequent

get_job_status request; see part 1 note 5).

2. *outcome* - the outcome of the job, as follows:

Value	Meaning
11	Compilation error. The *cmpinfo* field should offer further explanation
12	Runtime error. The job compiled but threw an exception at run time that isn't covered by any of the more-specific errors below.
13	Time limit exceeded. The job was killed before it ran to completion as a result of the server-specified time limit (or a possible time limit specified via the *parameters* field of the job request) being reached.
15	OK. The run ran to completion without any exceptions.
17	Memory limit exceeded. The job was killed before it ran to completion as a result of the server-specified maximum memory limit (or a possible memory limit specified via the *parameters* field of the job request) being reached.
19	Illegal system call. The task attempted a system call not allowed by this particular server.
20	Internal error. Something went wrong in the server. Please report this to an administrator.
21	Server overload. No free Jobe user accounts. Probably something has gone wrong.

   The precise situations under which these outcome values are returned will be server dependent. For example, a server might might limit the memory by denying memory allocation requests without terminating the job: the job would then possibly generate its own error message and exit without throwing an exception. Similarly, a server might not recognise an illegal system call as such but just lock the job in a chroot jail so that potentially dangerous system calls are unable to do any damage.

3. *cmpinfo* - any output generated by the compiler (if there is one) at compile time.
4. *stdout* - the standard output from the program run
5. *stderr* - the standard error output from the program run
