GATE REFERENCE

This document describes the "gate" service, a NodeJS service managed by PM2.

Theory of Operation

Overview

The `gate` service is `NodeJS` service that uses the `gated` service to provide a single client access to GATE (running in a JVM).

NOTE: The `gate` service supports one client making calls to `gated`. The `gate` service does not support multiple simultaneous clients and does not support parallel operations for any one client. The "GATE Developer" and "GATE Cloud" products from the University of Sheffield are more appropriate for team and parallel use.

Architecture and Components

The `gate` service is a `NodeJS` service built using the `express` framework. All `gate` code is written in Javascript (ES6). The `gate` service is designed for use by browser code that interacts with the user.

The `gate` service answers a JSON-formated literal object for each 200 request. This response always contains an "isSuccessful" binding with a value of "true" or "false". Successful responses contain bindings for client use. Unsuccessful responses contain information about the failure.

The `gate` service is partitioned into "model" and "controller" behavior. It provides no "view" behavior (other than rendering HTML error pages). Where possible, `gate` relies on conventional ES6 classes and methods.

All `gate` classes and methods (including test classes) share common abstract classes (documented elsewhere) that provide shared behavior. This shared behavior includes detecting and reporting exceptional conditions, invoking methods selected at runtime, loading and reference classes as needed, and so on.

NOTE: This document and the code it describes uses the identifier "clazz" to describe an entity in the shared framework that acts as an ES6 class. This entity is called a "clazz" in part because "class" is a keyword in ES6 and also because there are scenarios that require more than one "class" to provide the behavior of a single "clazz". Whew!

Each endpoint provided by `gate` is comprised of:

- Methods in `app.js` that load and dispatch to a `express` router
- An instance of `express.Router` that matches the operations of the endpoint
- An `AbstractController` descendant that connects the endpoint to its model
- Model classes that provide the actual model behavior

The `gate` service provides seven (7) endpoints, as follows:

1. `cleanup`
2. `corpus`

3. `document`
4. `pipeline`
5. `plugin`
6. `pr`
7. `resetSharedState`

These are each GET endpoints. The `resetSharedState` endpoint is used during cleanup to discard the contents of shared state within the `gate` service. The remaining six endpoints correspond to their counterparts in `gated`.

It is partitioned into several components, as follows:

- Client interaction

  This component implements the behavior needed to handle each client request. This component includes and uses standard `express` middleware for routing, CORS, error reporting, and so on.

- Routing

  Each of the seven endpoints has its own `express.Router`. The routers are:

  - `v0CleanupEndpoints`
  - `v0GateCorpusEndpoints`
  - `v0GateDocumentEndpoints`
  - `v0GatePipelineEndpoints`
  - `v0PluginEndpoints`
  - `v0ProcessingResourceEndpoints`
  - `v0ResetSharedStateEndpoints`

  Each of these creates an instance of `express.Router` and populates it with an asyncronous arrow function for each operation provided by that endpoint.

  Each of these is paired with a corresponding concrete descendant of `AbstractController` (described below). This is its "controller clazz".

  Each `gate` endpoint is implemented with a GET request.

  Each endpoint collects the endpoint name and instantiates its controller clazz. It collects the HTTP request, response, and next parameters provided by `express` and invokes the `handleEndpoint_request_response_next_` method on its controller clazz.

  The router classes are located in subdirectory of "protected_routes" in preparation for each to be protected by a JWT for authentication. While no such authentication is currently required, it is expected to be added before the `gate` service is made available to Zeetix users.

- Controller

  Each endpoint implements a descendant of `AbstractController` called its "controller clazz". There are seven controller clazzes, as follows:

  - `CleanupController`
  - `GateCorpusController`

- GateDocumentController
- GatePipelineController
- PluginController
- ProcessingResourceController
- ResetSharedStateController

Each controller clazz is paired with a class that provides its model behavior. This is its "model clazz". The model clazzes provided by gate are described below.

Each controller clazz performs validation on its input, intermediate results, and output. If a required input parameter is missing, the controller clazz will cause an HTTP400 ("Invalid Request") error to be returned with a description that identifies the missing parameter.

If a required interim result is missing or invalid, then gate will return an HTTP404 ("Not Found") error with a description that helps identify the issue. For example, an attempt to run a pipeline named "testpipeline" that has no corpus will result in an HTTP404 response that includes the string "testpipeline not runnable".

AbstractController provides handleOperation_request_response_next_, an async method that ensures that each request is performed in a predictable way.

The implementation of handleOperation_request_response_next_ has three steps as follows:

```
await this.preOperation_request_response_next_(anOperation, aRequest,
aResponse, aNext);
await this.performOperation_request_response_next_(anOperation,
aRequest, aResponse, aNext);
await this.postOperation_request_response_next_(anOperation, aRequest,
aResponse, aNext);
```

Each of these three steps invokes a hook that a descendants may override to provide behavior specific to that descendant, as follows:

- doPreOperation_request_response_next_

  Refreshes the responseHash and errorResponse of the receiver in preparation for performing a new operation. A descendant that adds additional instance state should override this to reset that state (after invoking super.doPreOperation_request_response_next_).

- doPerformOperation_request_response_next_

  This default implementation of this method in AbstractController does nothing.

  Each controller clazz overrides doEndpoint_request_response_next_ and answers a handler that performs the specified operation. Each handler performs whatever validation is appropriate and then invokes the appropriate behavior of its model clazz.

  Each handler may throws one of the following exceptions during execution:

        - HTTP400Error
        - HTTP404Error
        - RuntimeError

      These are caught and reported during postOperation_request_response_next_.

      Each model clazz has behavior that coerces an instance of itself into a literal object.

      Each handler sets the value of its this._responseHash to this literal object just before finishing. In the absence of any errors, this is returned to the client during the subsequent postOperation_request_response_next_ method.

    - doPostOperation_request_response_next_

      Here is the default implementation of this method:

```
async doPostOperation_request_response_next_(anOperation,
aRequest, aResponse, aNext) {
    if (this._errorResponse) {
        aNext(this._errorResponse);
    } else {
        aResponse.json(this._responseHash);
    }
}
```

      If an unhandled error has occurred, that exception will be bound to this._errorResponse, and the default implementation uses aNext to invoke the express error-handling middleware.

      In the absence of an error, this method sends its JSON-encoded this._responseHash to the client as the result of the request.

      A descendant may override this method if extend the post-operation behavior. If so, then the descendant should "super" this method (super.doPostOperation_request_response_next_) as its last step.

- Model

  The model of each endpoint is a descendant of GateComponent that provides the behavior of that endpoint. This is the "model clazz" of the endpoint.

  Each operation defined in a controller clazz ultimately maps to a method in its model clazz that forms a gated URL, uses axios to asyncronously invoke that method, collects the response from gated, and then returns that response to its caller (usually a method on its controller clazz).

  There is no model clazz for the "ResetSharedState" and "cleanup" endpoints.

  Here are the model clazzes for the remaining endpoints:

    - GateCorpus
    - GateDocument

- `GatePipeline`
- `Plugin`
- `ProcessingResource`

Each of these is described below.

Lifecycle of a typical `gate` session

A client interaction with `gate` has three distinct phases:

1. Setup
2. Use
3. Cleanup

These occur in a single "session".

During the "Setup" phase, a client loads one or more `plugin` instances.

During the "Use" phase, the client creates or loads needed resources and then performs operations on those resources.

During the "Cleanup" phase, the client invokes housekeeping endpoints that free resources in the backend and that reset the state of the `gate` service to discard any cached entities.

Each endpoint used during "Setup" and "Cleanup" is idempotent. It is therefore good practice to call the "Cleanup" endpoints during "Setup" to handle the unlikely event that any of `gate`, `gated`, or GATE were not properly cleaned up at the end of a prior session.

Using `gate` from the command line

The endpoints provided by the `gate` service can be accessed from the command line using an SSH shell.

Many endpoints of `gate` throw exceptions or fail in unexpected ways if a plugin is not loaded. The current implementation and all examples in this document assume the use of the ANNIE plugin.

The `gated` daemon is a long-running service managed by `systemd` (`systemctl` on Rocky Linux). It is therefore good practice to invoke the `cleanup` endpoint at the conclusion of a manual session where other endpoints are exercised.

For convenience, the following `wget` command lines should be executed before and after exercising other `gate` endpoints:

> UNDER CONSTRUCTION

Load the ANNIE plugin:

ANNIE

```
https://hoyo.zeetix.com:7403/gate/v0/plugin/mavenPlugin?
group=uk.ac.gate.plugins&artifact=annie&version=9.1
```

Cleanup GATE:

```
https://hoyo.zeetix.com:7899/v0/cleanup
```

The cleanup endpoint is idempotent, so it never hurts to run it before starting a new manual session. This will cleanup if some earlier session was improperly terminated.

The following example endpoint is used to run a pipeline that has already been created and configured:

```
https://hoyo.zeetix.com:7899/v0/pipeline/runPipeline?
pipelineName=testpipeline
```

The following table uses the above example to illustrate the several parts of a `GateDaemon` endpoint, using the above example.

Part	Example	Description
base	https://hoyo.zeetix.com	The https URL of the target system
port	7899	The designated port of the `gated` service
version	v0	Version specifier for the endpoint
path	pipeline	A version designator followed by the name of the resource to be used
operation	runPipeline	The operation to perform on the designated resource
query	pipelineName=testpipeline	A sequence of one or more key-value pairs

Table 1: GateDaemon Endpoint Anatomy

For convenience, every endpoint in the `gate` project is a GET endpoint. Each successful response is a JSON-encoded object containing the response from the GATE instance.

This means that any endpoint can be exercised using the "wget" command. Here is a command line string that invokes the example endpoint:

```
wget "https://hoyo.zeetix.com:7899/v0/pipeline/runPipeline?
pipelineName=testpipeline"
```

NOTE: The URL is enclosed in a double-quote pair (`"`) because many endpoints contain reserved characters interpreted by the command line shell (such as `bash`).

Here is a command line that loads the ANNIE plugin, together with its response:

Command line:

```
wget "https://hoyo.zeetix.com:7899/v0/plugin/loadMavenPlugin?
group=uk.ac.gate.plugins&artifact=annie&version=9.1"
```

Response (pretty printed and reordered for clarity):

```
{
    "isSuccessful":"true",
    "operation":"loadMavenPlugin",
    "pluginGroup":"uk.ac.gate.plugins"
    "pluginArtifact":"annie",
    "pluginVersion":"9.1",
}
```

Path	**Operations**	**Parameters (*=optional)**
plugin		
	loadMavenPlugin	group, artifact, version
cleanup		
	n.a.	*n.a.*
document		
	loadFromURL	documentURL
	loadFromPath	documentPath
	getDocumentContent	documentName
	getAnnotationSetNames	documentName
	getAnnotationsForName	documentName, annotationSetName*
	cleanupDocument	documentName
corpus		
	createCorpus	corpusName
	clearCorpus	corpusName
	addDocument	corpusName, documentName
pr		
	loadPR	prName, resourcePath
	reinitPR	prName
	unloadPR	prName
pipeline`		

Path	Operations	Parameters (*=optional)
	createPipeline	pipelineName
	addPR	pipelineName, prName
	setCorpus	pipelineName, corpusName
	getParameterValue	pipelineName, prName, parameterName
	setParameterValue	pipelineName, prName, parameterName, parameterValue*, parameterType
	runPipeline	pipelineName
	storePipelineToFile	pipelineName, pipelinePath
	loadPipelineFromFile	pipelineName, pipelinePath

Table 2: GateDaemon Endpoints

NOTE: The `parameterValue` is optional for the `v0/pipeline` endpoint so that a parameter value can be set to `null`. This is accomplished by providing values for the `pipelineName`, `prName`, `parameterName`, and `parameterType`.