|  |
| --- |
| NCBI |
| Pubseq Gateway Server (PSG) |
| Overview and the Protocol Specification |
|  |
| **Sergey Satskiy** |
| **1/24/2018** |
| **Last update date: 07/09/2021** |

|  |
| --- |
| Document version: 2.41 |

Changes history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | What changes |
| 1.0 | Jan 24, 2018 | Sergey Satskiy | Initial revision |
| 1.1 | Mar 08, 2018 | Sergey Satskiy | Added format #2 for blob retrieval; GetBlob diagram |
| 1.2 | Mar 09, 2018 | Sergey Satskiy | Changes in the resolution request URL format |
| 1.3 | Mar 23, 2018 | Sergey Satskiy | Adding ADMIN/config request |
| 1.4 | Mar 26, 2018 | Sergey Satskiy | Adding ADMIN/info request |
| 1.5 | Mar 30, 2018 | Sergey Satskiy | Adding ADMIN/status request |
| 1.6 | Apr 16, 2018 | Sergey Satskiy | Changes in the response format; adding description of the communication protocol. |
| 1.7 | Apr 19, 2018 | Sergey Satskiy | Addig [server]/maxretries setting description |
| 1.8 | Apr 20, 2018 | Sergey Satskiy | Updates on the communication protocol |
| 1.9 | Apr 24, 2018 | Sergey Satskiy | Updates on the communication protocol |
| 1.10 | May 5, 2018 | Sergey Satskiy | Updates related to logging |
| 1.11 | May 18, 2018 | Sergey Satskiy | Sat to keyspace mapping description |
| 2.0 | May 24, 2019 | Sergey Satskiy | Massive update of what is currently implemented. |
| 2.1 | May 30, 2019 | Sergey Satskiy | Adding ADMIN/get\_alerts and ADMIN/ack\_alert requests description |
| 2.2 | Jun 18, 2019 | Sergey Satskiy | Adding ADMIN/statistics and the [STATISTICS] configuration file section description |
| 2.3 | Aug 8, 2019 | Sergey Satskiy | Adding ID/get\_tse\_chunk request |
| 2.4 | Oct 9, 2019 | Sergey Satskiy | Updates for PSG 1.8.0 |
| 2.5 | Oct 28, 2019 | Sergey Satskiy | Adding the ‘name’ flag for PSG 1.8.2 |
| 2.6 | Nov 12, 2019 | Sergey Satskiy | Updates for the ID/get\_na requests |
| 2.7 | Dec 04, 2019 | Sergey Satskiy | Adding the ‘seq\_state’ flag for PSG 1.8.2 |
| 2.8 | Dec 10, 2019 | Sergey Satskiy | Addin the ‘trace’ flag for PSG 1.8.3 |
| 2.9 | Jan 10, 2020 | Sergey Satskiy | Adding new parameters for the ADMIN/statistics url |
| 2.10 | Mar 18, 2020 | Sergey Satskiy | Adding the [COUNTERS] configuration section |
| 2.11 | Jun 09, 2020 | Sergey Satskiy | Making psg\_protocol parameter obsolete for the resolve and get\_na requests for PSG 1.10.0 |
| 2.12 | Jul 09, 2020 | Sergey Satskiy | Adding hops pararmeter and config setting for PSG 1.10.0 |
| 2.13 | Jul 09, 2020 | Sergey Satskiy | Fix: adding missed client\_id parameter description for the ID/getblob request |
| 2.14 | Jul 16, 2020 | Sergey Satskiy | blob id format updates |
| 2.15 | Jul 27, 2020 | Sergey Satskiy | PSG protocol updates |
| 2.16 | Jul 31, 2020 | Sergey Satskiy | Bug fixes: description of the tse option |
| 2.17 | Aug 11, 2020 | Sergey Satskiy | Adding description of the get\_tse\_chunk parameters for PSG 2.0.0 |
| 2.18 | Sep 2, 2020 | Sergey Satskiy | Adding description of the auto\_blob\_skipping parameter for ID/get request for PSG 1.10.1 |
| 2.19 | Sep 29, 2020 | Sergey Satskiy | Update description of the ID/get\_tse\_chunk for PSG 2.0.0 and up |
| 2.20 | Oct 5, 2020 | Sergey Satskiy | Update description of the ID/get and ID/getblob replies for PSG 2.0.0.  Fix: description of the tse=<…> request option for the ID/get and ID/getblob requests.  Update description of the special id2\_chunk value for the ID/get\_tse\_chunk request |
| 2.21 | Oct 15, 2020 | Sergey Satskiy | Adding enable\_processor and disable processor parameters description |
| 2.22 | Oct 27, 2020 | Sergey Satskiy | Add description of the ID/get\_tse\_chunk request id2\_info format for the Cassandra/LMDB processor |
| 2.23 | Nov 3, 2020 | Sergey Satskiy | Add public comments description, protocol diagrams corrections, protocol grammar fixes |
| 2.24 | Jan 19, 2021 | Sergey Satskiy | Add description of the /health and /deep-health URLs |
| 2.25 | Feb 17, 2021 | Sergey Satskiy | Adding the [SSL] section description |
| 2.26 | Feb 26, 2021 | Sergey Satskiy | Changing default for [SSL]/ssl\_ciphers |
| 2.27 | Mar 10, 2021 | Sergey Satskiy | Better description of the logging option in the configuration file |
| 2.28 | Mar 17, 2021 | Sergey Satskiy | Reply updates for error cases: now PSG protocol is used. |
| 2.29 | Mar 22. 2021 | Sergey Satskiy | Updates for /health (/deep-health) and /favicon.ico requests |
| 2.30 | Mar 25, 2021 | Sergey Satskiy | Adding PSG processors description |
| 2.31 | Mar 29, 2021 | Sergey Satskiy | Adding PSG requests and timing collection description |
| 2.32 | Mar 30, 2021 | Sergey Satskiy | Adding ‘tse’ parameter for the ID/get\_na requests |
| 2.33 | Mar 31, 2021 | Sergey Satskiy | Adding Exclude Blob API section |
| 2.34 | Apr 12, 2021 | Sergey Satskiy | Adding description of the alerts and counters API |
| 2.35 | Apr 20, 2021 | Sergey Satskiy | Adding send\_blob\_if\_small URL parameter for ID/get, ID/getblob and ID/get\_na requests |
| 2.36 | May 5, 2021 | Sergey Satskiy | Adding auto\_blob\_skipping for ID/get\_na request |
| 2.37 | May 12, 2021 | Sergey Satskiy | Adding an async IO API section |
| 2.38 | June 15, 2021 | Sergey Satskiy | Adding general server structure documentation |
| 2.39 | June 25, 2021 | Sergey Satskiy | Adding a section about waiting for non-libuv events. |
| 2.40 | July 7, 2021 | Sergey Satskiy | ID/accession\_version\_history description |
| 2.41 | July 9, 2021 | Sergey Satskiy | Protocol diagram updates; ID/accession\_version\_history description update |

Table of Contents

[Pubseq Gateway Server (PSG) 6](#_Toc76567284)

[Overview 7](#_Toc76567285)

[Communication Protocol 9](#_Toc76567286)

[PSG Protocol 9](#_Toc76567287)

[Exclude Blob Cache 17](#_Toc76567288)

[Requests 17](#_Toc76567289)

[Common ID/... Request Parameters 18](#_Toc76567290)

[ID/getblob Request 19](#_Toc76567291)

[ID/get Request 21](#_Toc76567292)

[ID/get\_tse\_chunk Request 23](#_Toc76567293)

[ID/resolve Request 25](#_Toc76567294)

[ID/get\_na Request 27](#_Toc76567295)

[ID/accession\_version\_history Request 29](#_Toc76567296)

[ADMIN/config Request 30](#_Toc76567297)

[ADMIN/info Request 30](#_Toc76567298)

[ADMIN/status Request 33](#_Toc76567299)

[ADMIN/shutdown Request 35](#_Toc76567300)

[ADMIN/get\_alerts Request 37](#_Toc76567301)

[ADMIN/ack\_alert Request 37](#_Toc76567302)

[ADMIN/statistics Request 38](#_Toc76567303)

[TEST/io Request 39](#_Toc76567304)

[health Request 40](#_Toc76567305)

[deep-health Request 40](#_Toc76567306)

[favicon.ico Request 40](#_Toc76567307)

[Unknown URL Request 41](#_Toc76567308)

[Cassandra Database 41](#_Toc76567309)

[Monitoring and Maintenance 41](#_Toc76567310)

[Files Architecture 42](#_Toc76567311)

[Client API 42](#_Toc76567312)

[Command Line Arguments 43](#_Toc76567313)

[Signal Handling 43](#_Toc76567314)

[Configuration Parameters 43](#_Toc76567315)

[[LMDB\_CACHE] Section 44](#_Toc76567316)

[[SERVER] Section 44](#_Toc76567317)

[[AUTO\_EXCLUDE] Section 45](#_Toc76567318)

[[ADMIN] Section 45](#_Toc76567319)

[[STATISTICS] Section 45](#_Toc76567320)

[[DEBUG] Section 46](#_Toc76567321)

[[CASSANDRA\_DB] Section 46](#_Toc76567322)

[[CASSANDRA\_PROCESSOR] Section 47](#_Toc76567323)

[[COUNTERS] Section 47](#_Toc76567324)

[[HEALTH] Section 48](#_Toc76567325)

[[SSL] Section 48](#_Toc76567326)

[Appendix 49](#_Toc76567327)

[PSG Processors 49](#_Toc76567328)

[PSG Requests 51](#_Toc76567329)

[PSG Timing Collecting 52](#_Toc76567330)

[Exclude Blob API 54](#_Toc76567331)

[Alerts API 55](#_Toc76567332)

[Event Counter API 55](#_Toc76567333)

[Asynchronous socket polling API 56](#_Toc76567334)

[Handling Non-libuv Events Asynchronously 57](#_Toc76567335)

[Protocol Diagrams 60](#_Toc76567336)

[GetBlob Diagram 63](#_Toc76567337)

[General Server Structure 64](#_Toc76567338)

[Startup 64](#_Toc76567339)

[CTcpWorker 64](#_Toc76567340)

[CTcpWorkersList 64](#_Toc76567341)

[CTcpDaemon 64](#_Toc76567342)

[CHttpRequest 65](#_Toc76567343)

[CPSGS\_Request 65](#_Toc76567344)

[CHttpReply 65](#_Toc76567345)

[CPSGS\_Reply 65](#_Toc76567346)

[CHttpConnection 65](#_Toc76567347)

[CHttpProto 65](#_Toc76567348)

[CHttpDaemon 65](#_Toc76567349)

[CPendingOperation 66](#_Toc76567350)

[CPSGS\_Dispatcher 66](#_Toc76567351)

[New Connection Flow 66](#_Toc76567352)

[Request Flow 67](#_Toc76567353)

[Handling Request In Synchronous Manner 68](#_Toc76567354)

[Handling Request In Asynchronous Manner 69](#_Toc76567355)

# Pubseq Gateway Server (PSG)

This document provides an overview of the Pubseq Gateway server functionality. Basically the server provides the following services:

* accession resolution
* blobs retrieval based on accession or on blob identification
* named annotations retrieval
* monitoring of the server including timing of the major operations

Accessions are string identifiers.

Below is a list of major Pubseq Gateway server implementation details:

* The server operates as a Linux operating system daemon.
* The server reads all the settings from a configuration file only.
* The server serves many clients simultaneously.
* The logging facilities is provided via standard C++ toolkit facilities
* The server provides an interface for monitoring.
* The communication protocol with the clients is HTTP 1.1 or HTTP/2.
* The server does not deal neither with authentication nor with authorization. These features, if necessary, needs to be implemented outside of the server.

# Overview

Basically, the Pubseq Gateway server is stateless and operates in request – response mode.

The diagram below shows the main actors and entities involved into a typical Pubseq Gateway application.



The clients establish TCP/IP connections using HTTP 1.1 or HTTP/2 protocols with the Pubseq Gateway server via an API (psg\_client library), and they send requests over the established connection.

The Cassandra DB stores three major types of objects: resolutions for accessions, named annotations and BLOBs. All the data in Cassandra are split into keyspaces. The information of what data are stored in what keyspace is also located in one of the Cassandra tables in a specific keyspace. That keyspace name is configured for the PSG server so the server reads all the mapping at the startup time and uses it later on.

To speed up the data lookups there is a local copy of a certain portion of the Cassandra data stored in a few LMDB files. The file is populated by a synchronization utility shown on the diagram in green. Sometimes the LMDB cache excepts is enough to complete a request however in a worst case scenario a trip to Cassandra will also be required. Generally speaking the server is able to work with cache files (if properly configured) or without. If configured then first the lookups are done in LMDB first and then in Cassandra. Also, the user can control the use of the LMDB cache via a URL parameter for certain requests.

The modern PSG server supports an infrastructure for processors. A processor is a C++ class which follows a certain interface. The notion of processors allows to add the other data sources on top of Cassandra and LMDB. In fact the processors may in parallel work on the same request.

# Communication Protocol

The communication with the server is provided over HTTP 1.1 or HTTP/2 protocols.

The requests are standard URLs so the server extracts the parameters in a standard way.

The responces are standard HTTP 1.1 or HTTP/2 replies however in most of the cases the reply body introduces a higher level structure which is called PSG protocol. The PSG protocol is comprised out of reply chunks and possibly some data. Whether or not a PSG protocol appears in the reply will be described in the individual requests sections.

## PSG Protocol

Essential link: <https://confluence.ncbi.nlm.nih.gov/pages/viewpage.action?pageId=106579021> – the protocol description in confluence.

PSG protocol response is comprised of two or more PSG protocol chunks in the HTTP response body.



The sequence of the chunks is not guaranteed. The client understands that all the chunks are read when the final chunk is delivered. The final chunk has an information about the total number of chunks the client should expect in response to the request.

The PSG protocol HTTP status code is always 200. If an error is encountered then the error information is supplied in the reply as one of the chunks.

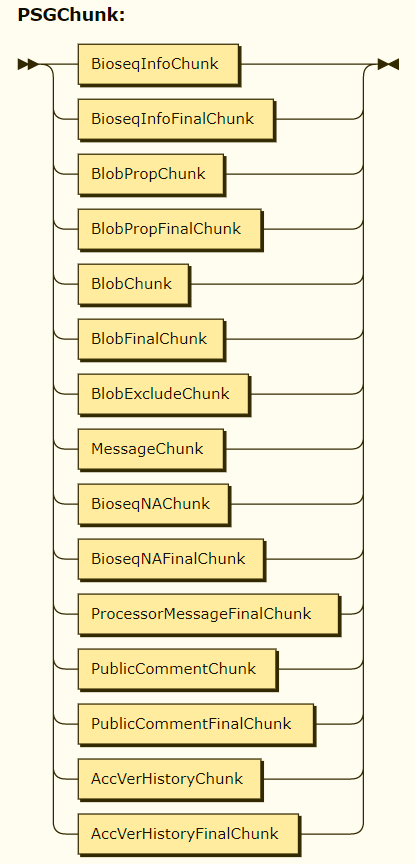
The “Content-type” header is set to “application/x-ncbi-psg”. The “Content-length” header is not set.



Sometimes it is needed to identify a blob. To do that a generic string is used. Various processors may use different formats for this string. For example, Cassandra/LMDB related processors use two integers divided by the ‘.’.



PSG server may have more than one processor to handle a request. So the reply to the request may contain items which were produced by more than one processor. So on many occaisions reply chunks will have an identification what processor produced a certain item. The identification is a non-structured string which in the PSG reply chunks is URL encoded.



Each PSG chunk uses a fixed prefix and then a set of URL-like paremeters which depends on a chunk type.



The BioseqInfoChunk is used to send bioseq info data. The item\_id parameter is a positive integer greater than zero which uniquely identifies the data item. Depending on the request the data are supplied in json or a protobuf format. So the data size is returned in the size parameter and the format is specified in the fmt parameter. The data follow the chunk and is a human readable string in case of json or a binary content in case of protobuf.



The BioseqInfoFinalChunk is used to send the information of how many chunks were sent about the blob bioseq info – see the n\_chunks parameter.



The BlobPropChunk is used to send the appropriate blob properties. The item\_id parameter is a positive integer greater than zero which uniquely identifies the data item. The provided data are always in json format and the size parameter tells the size of data.



The BlobPropFinalChunk is used to send the information of how many chunks were sent about the blob properties – see the n\_chunks parameter.



Blobs are stored in Cassandra in a form of chunks. A blob may have an arbitrary number of chunks and each of them could be of an arbitrary size. When a blob is sent to the client the chunks are transferred to the client exactly as they are stored in Cassandra. The blob chunks are numbered consequently starting from zero. So the item\_id parameter uniquely identifies the blob; it is greater than zero and stays the same for all the blob chunks. The size parameter tells the chunk size in bytes. The blob\_id parameter identifies the blob while the blob\_chunk tells the chunk sequential number.



When all blob chunks are sent to the client the server sends one more chunk with the blob finilizing information. The item\_id parameter value matches the BlobChunk chunk item\_id. The n\_chunks parameter value tells how many chunks were sent in total about the blob including this very chunk.



If the blob exclude cache feature is switched on, then the server may sent the BlobExcludeChunk chunk instead of the BlobChunks and the BlobFinalChunk chunks. The reason parameter in this case provides the exact reason why the blob was not sent.





In case of warnings, errors etc the server sends the MessageChunk. The message is linked to an appropriate item\_id as well as to the item type. If appropriate, the blob id is also supplied. The rest of the parameters describe a message similar to the C++ toolkit log messages.



The BioseqNAChunk is sent when the server responses with a named annotation information. The seq\_acc, seq\_ver and seq\_type provide accession, version and type respectively.



Each BioseqNAChunk is followed by a BioseqNAFinalChunk. The final chunk tells the total number of chunks sent about the item\_id.



The AccVerHistoryChunk is sent when the server responses with an accession version history record.



Each AccVerHistoryChunk is followed by a AccVerHistoryFinalChunk. The final chunk tells the total number of chunks sent about the item\_id.



If a processor message is generated it accompanied with the ProcessorMessageFinalChunk.



When the server finishes response it sends the PSGFinalChunk. This chunk tells the total number of chunks in the response (including this very chunk).

The PSG protocol reserves the item\_id value zero for the cases when a chunk is related to the whole response. The examples of such chunks are error messages and a final response chunk.



The ID/get, ID/getblob and ID/get\_tse\_chunk requests may result in blobs which have public comments. In those cases the PublicCommentChunk will appear in the replies.



If a PublicCommentChunk appeared in the reply then it is followed with the PublicCommentFinalChunk which shares the item\_id value with the PublicCommentChunk.

# Exclude Blob Cache

The PSG server supports blob requests based on seq\_id and seq\_id\_type. In this case there is a procedure of the provided identification resolution into a pair of sat and a sat\_key. This pair of values is used internally to retrieve and transfer the blob. It may happened that the client issues a massive number of blob retrieve requests using seq\_ids. In this scenario the resolution of many different seq\_ids may lead to the very same pair of sat and sat\_key. Consequently it will lead to transferring the same blob many times to the the very same client.

To address the problem – i.e. to avoid transferring the same blob more than one time to the same client – the PSG server introduces the exclude blob cache feature. It works as follows. When a client requests a blob with seq\_id/seq\_id\_type identification it also provides the client name as well as an optional list of the blob\_ids which the client already has. When the seq\_id/seq\_id\_type resolution procedure is finished the result sat/sat\_key is looked in the list provided by the client. If found then the blob is not sent. If not found then the exclude blob cache is looked up. If the blob\_id is found for the client then the blob is not sent. Otherwise a records about the blob is created in the cache.

The BlobExcludeChunk has the reason parameter which tells the following:

* excluded: the blob was found in the list supplied by the client.
* inprogress: the blob was found in the cache; the transfer of the blob chunks is in progress. There is no guarantee that all the blob chunks will be transferred successfully.
* sent: the blob was found in the cache; the server has finished transferring the blob to the client before.

The cache supports automatic garbage collection. It is provided basing on the timeout when the client communicated with the server last time as well as on the maximum number of the most recent blob records per client.

# Requests

The server accepts HTTP 1.1 and HTTP/2 GET requests. The section describes the requests and the server responces.

The requests are split into three cathegories:

* Data requests
* Administrative requests
* Test requests

The distinguish between the request cathegory is the first path element in the request URL.

## Common ID/... Request Parameters

All requests which URL starts with http://<host:port>/ID/ have common parameters. The table below describes them.

|  |  |
| --- | --- |
| Parameter | Description |
| trace=<trace> | The option to include trace messages to the server output. Acceptable values: yes and no.  Default: no |
| hops=<N> | Number of hops, integer greater than zero.  If the number of hops is greater than the configured value of [SERVER]/max\_hops (default: 2) then the request is rejected and the server replies in PSG protocol with 400 error code.  Note: individual processors may reject a request basing on a specific logic around the number of hops.  Optional parameter.  Default: 0 |
| enable\_processor=<identifier> | The parameter tells what processor is enabled.  There could be many enable\_processor parameters with different string identifiers. It works in pair with the disable\_processor parameters. Individual processors may consult to the enabled and disabled processors from the request and make a decision if they are enabled or not. The logic depends on a particular processor.  Optional parameter.  Default: empty string |
| disable\_processor=<identifier> | The parameter tells what processor is disabled.  There could be many disable\_processor parameters with different string identifiers. It works in pair with the enable\_processor parameters. Individual processors may consult to the enabled and disabled processors from the request and make a decision if they are enabled or not. The logic depends on a particular processor.  Optional parameter.  Default: empty string |

Cassandra/LMDB processors implement the logic for the enable\_processor and disable\_processor as follows:

* Check what the configuration file setting in [CASSANDRA\_PROCESSOR]/enabled (default: 1)
* If [CASSANDRA\_PROCESSOR]/enable is 1 then the disable\_processor list is checked. If there is (case insensitive) value “cassandra" in the list then the processor is disabled.
* If [CASSANDRA\_PROCESSOR]/enable is 0 then the enable\_processor list is checked. If there is (case insensitive) value “cassandra" in the list then the processor is enabled.

## ID/getblob Request

The format of the request:

http://<host:port>/ID/getblob

where (see the [Common ID/... Request Parameters](#_Common_ID/..._Request) chapter as well):

|  |  |
| --- | --- |
| Parameter | Description |
| blob\_id=<string> | The blob identifier.  Mandatory parameter  Processors may interpret the blob id in their own way.  Cassandra processors expect the following format: <sat>.<sat\_key> where both are integers. |
| tse=<tse\_opt> | TSE option.  Return the following blobs depending on the value:   |  |  |  | | --- | --- | --- | | Value | ID2 split available | ID2 split not available | | none | Nothing | Nothing | | whole | All split blobs | All Cassandra data chunks of the blob itself | | orig | All Cassandra data chunks of the blob itself | All Cassandra data chunks of the blob itself | | smart | Split INFO blob only | All Cassandra data chunks of the blob itself | | slim | Split INFO blob only | Nothing |   Optional parameter. Default value: orig |
| last\_modified=<last\_mod> | Last modified, integer.  If provided then the exact match will be requested with the Cassandra storage corresponding field value.  Optional parameter.  By default the most recent match will be provided. |
| use\_cache=<cache> | Allowed values:   * no: do not use LMDB cache (tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP) at all; go straight to Cassandra storage. * yes: do not use tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP from Cassandra storage at all. I.e., exclusively use the cache for all seq-id resolution steps. If the seq-id cannot be fully resolved through the cache alone, then code 404 must be returned.   Optional parameter.  By default (no use\_cache option specified), the behavior is to use the LMDB cache if at all possible; then, fallback to Cassandra storage. |
| client\_id=<client\_id> | The client identifier (string).  If provided then the exclude blob feature takes place.  Optional parameter.  **Note**: see the [Exclude Blob API](#_Exclude_Blob_API) for more information |
| send\_blob\_if\_small=<# bytes> | Integer >= 0  If [SERVER]/send\_blob\_if\_small config value is bigger of that then [SERVER]/send\_blob\_if\_small should be used.   * "tse" - value of {{tse}} URL parameter * "id2-split" -- whether the ID2-split version of the blob is available * "Small blob" -- size of the (compressed) blob data <= send\_blob\_if\_small * "Large blob" -- size of the (compressed) blob data > send\_blob\_if\_small  |  |  |  |  | | --- | --- | --- | --- | | tse | id2-split | Small blob | Large blob | | slim | no | Send original (non-split) blob data | Do not send original (non-split) blob data | | smart | no | Send original (non-split) blob data | Send original (non-split) blob data | | slim | yes | Send all ID2 chunks of the blob | Send only split-info chunk | | smart | yes | Send all ID2 chunks of the blob | Send only split-info chunk |   Optional parameter. Default: 0 |

The response uses the PSG protocol.

The HTTP header Content-Type is set to “application/x-ncbi-psg”.

The HTTP header Content-Length is not set.

The HTTP 1.1 or HTTP/2 status code is always 200.

In case of success the following PSG protocol chunks will appear:



The sequence of chunks is not guaranteed.

In case of errors a MessageChunk will appear accompanied by the PSGFinalChunk.

The id\_chunk=<int> and the id2\_info=<string> values will be added to the reply chunks if the following coditions are met:

* The originally requested blob has id2info not empty
* the tse request option is not orig

If the id2\_chunk value is going to be added and the chunk sat\_key is equal the the sat\_key from the original blob props id2info then the id2\_chunk value is reported as 999999999.

## ID/get Request

The format of the request:

http://<host:port>/ID/get?

where (see the [Common ID/... Request Parameters](#_Common_ID/..._Request) chapter as well):

|  |  |  |
| --- | --- | --- |
| Parameter | Description | |
| seq\_id=<seq\_id> | SeqId of the blob to be retrieved (string).  Mandatory parameter. | |
| seq\_id\_type=<seq\_id\_type> | SeqId type of the blob to be retrieved (integer > 0).  Optional parameter. | |
| use\_cache=<cache> | Allowed values:   * no: do not use LMDB cache (tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP) at all; go straight to Cassandra storage. * yes: do not use tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP from Cassandra storage at all. I.e., exclusively use the cache for all seq-id resolution steps. If the seq-id cannot be fully resolved through the cache alone, then code 404 must be returned.   Optional parameter.  By default (no use\_cache option specified), the behavior is to use the LMDB cache if at all possible; then, fallback to Cassandra storage. | |
| tse=<tse\_opt> | TSE option.  Return the following blobs depending on the value:   |  |  |  | | --- | --- | --- | | Value | ID2 split available | ID2 split not available | | none | Nothing | Nothing | | whole | All split blobs | All Cassandra data chunks of the blob itself | | orig | All Cassandra data chunks of the blob itself | All Cassandra data chunks of the blob itself | | smart | Split INFO blob only | All Cassandra data chunks of the blob itself | | slim | Split INFO blob only | Nothing |   Optional parameter. Default value: orig | |
| exclude\_blobs=<exclude\_list> | A comma separated list of BlobId which client already has. If provided then if the resolution od seq\_id/seq\_id\_type matches one of the blob id then the blob will not be sent.  Optional parameter. | |
| client\_id=<client\_id> | The client identifier (string).  If provided then the exclude blob feature takes place.  Optional parameter.  **Note**: see the [Exclude Blob API](#_Exclude_Blob_API) for more information | |
| acc\_substitution=<policy> | The option controls how the bioseq info accession substation is done.  The supported policy values are:   * default: substitute if version value (version <= 0) or seq\_id\_type is Gi(12) * limited: substitute only if the resolved record's seq\_id\_type is GI(12) * never: the accession substitution is never done   If the substitution is needed then the seq\_ids list is analyzed. If there is one with Gi then it is taken for substitution. Otherwise an arbitrary one is picked.  Optional parameter. | |
| auto\_blob\_skipping=<value> | The option to switch on/off automatic cache of already sent blobs to a particular client. If on then a blob will be skipped if it was already delivered to the client.  Acceptable values: yes and no.  Default: yes | |
| send\_blob\_if\_small=<# bytes> | | Integer >= 0  If [SERVER]/send\_blob\_if\_small config value is bigger of that then [SERVER]/send\_blob\_if\_small should be used.   * "tse" - value of {{tse}} URL parameter * "id2-split" -- whether the ID2-split version of the blob is available * "Small blob" -- size of the (compressed) blob data <= send\_blob\_if\_small * "Large blob" -- size of the (compressed) blob data > send\_blob\_if\_small  |  |  |  |  | | --- | --- | --- | --- | | tse | id2-split | Small blob | Large blob | | slim | no | Send original (non-split) blob data | Do not send original (non-split) blob data | | smart | no | Send original (non-split) blob data | Send original (non-split) blob data | | slim | yes | Send all ID2 chunks of the blob | Send only split-info chunk | | smart | yes | Send all ID2 chunks of the blob | Send only split-info chunk |   Optional parameter. Default: 0 |

The response uses the PSG protocol.

The HTTP header Content-Type is set to “application/x-ncbi-psg”.

The HTTP header Content-Length is not set.

The HTTP 1.1 or HTTP/2 status code is always 200.

In case of success the following PSG protocol chunks will appear:



The sequence of chunks is not guaranteed.

In case of errors a MessageChunk will appear accompanied by the PSGFinalChunk.

The id\_chunk=<int> and the id2\_info=<string> values will be added to the reply chunks if the following coditions are met:

* The originally requested blob has id2info not empty
* the tse request option is not orig

If the id2\_chunk value is going to be added and the chunk sat\_key is equal the the sat\_key from the original blob props id2info then the id2\_chunk value is reported as 999999999.

## ID/get\_tse\_chunk Request

The format of the request:

http://<host:port>/ID/get\_tse\_chunk

where (see the [Common ID/... Request Parameters](#_Common_ID/..._Request) chapter as well):

|  |  |
| --- | --- |
| Parameter | Description |
| id2\_chunk=<chunk number> | The requied TSE blob chunk number. It must be greater or equal than 0.  Mandatory parameter  Note: the Cassandra/LMDB processor needs the chunk number greater than 0.  Note: the Cassandra/LMDB processors recognize a special value of 999999999 for the id2\_chunk. In this case the effective id2\_chunk value will be taken from the id2\_info. |
| id2\_info=<string> | The Cassandra/LMDB processor recognizes two formats as follows:   * 3 or 4 integers separated by '.': <sat>.<info>.<chunks>[.<split version>] * psg~~tse\_id-<sat>.<sat key>[~~tse\_last\_modified-<int>[~~tse\_split\_version-<int>]   The other processors may recognize the following format:  id2~~tse\_id-<string>~~tse\_last\_modified-<int>~~tse\_split\_version-<int> |
| use\_cache=<cache> | Allowed values:   * no: do not use LMDB cache (BLOB\_PROP table) at all; go straight to Cassandra storage. * yes: do not use BLOB\_PROP table from Cassandra storage if the initial lookup of the TSE blob in the BLOB\_PROP cache succeeded and its split version matched the requsted one.   Optional parameter.  By default (no use\_cache option specified), the behavior is to use the LMDB cache if at all possible; then, fallback to Cassandra storage. |

The response uses the PSG protocol.

The HTTP header Content-Type is set to “application/x-ncbi-psg”.

The HTTP header Content-Length is not set.

The HTTP 1.1 or HTTP/2 status code is always 200.

In case of success the following PSG protocol chunks will appear:



The Cassandra/LMDB processor extends all the chunks (except the final one) with 2 more values:

* id2\_chunk=<value from the request>
* id2\_info=<value from the request>

The Cassandra/LMDB processor message chunks in case of errors or warnings will also have id2\_chunk and id2\_info items.

## ID/resolve Request

The format of the request:

http://<host:port>/ID/resolve

where (see the [Common ID/... Request Parameters](#_Common_ID/..._Request) chapter as well):

|  |  |
| --- | --- |
| Parameter | Description |
| seq\_id=<seq\_id> | SeqId of the bioseq info to be retrieved (string).  Mandatory parameter. |
| seq\_id\_type=<seq\_id\_type> | SeqId type of the bioseq info to be retrieved (integer > 0).  Optional parameter. |
| use\_cache=<cache> | Allowed values:   * no: do not use LMDB cache (tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP) at all; go straight to Cassandra storage. * yes: do not use tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP from Cassandra storage at all. I.e., exclusively use the cache for all seq-id resolution steps. If the seq-id cannot be fully resolved through the cache alone, then code 404 must be returned.   Optional parameter.  By default (no use\_cache option specified), the behavior is to use the LMDB cache if at all possible; then, fallback to Cassandra storage. |
| fmt=<format> | The bioseq info data format (string).  Accepted values:   |  |  | | --- | --- | | protobuf | Bioseq info will be sent as a protobuf binary data  The protobuf format description can be found here: <https://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/objtools/pubseq_gateway/protobuf/psg_protobuf.proto> | | json | Bioseq info will be sent as a serialized JSON dictionary | | native | The PSG server will decide what format to use: protobuf or json. |   Optional parameter.  Default: native |
| all\_info=<bool\_val>  canon\_id=<bool\_val>  seq\_ids=<bool\_val>  mol\_type=<bool\_val>  length=<bool\_val>  state=<bool\_val>  blob\_id=<bool\_val>  tax\_id=<bool\_val>  hash=<bool\_val>  date\_changed=<bool\_val>  gi=<bool\_val>  name=<bool\_val>  seq\_state=<bool\_val> | It is used to specify explicitly what values to include/exclude from the provided bioseq info. The accepted values are yes and no.  It could be used e.g. as follows:  …&all\_info=yes&length=no  In this case all the fields will be supplied except of the length.  Optional parameters.  Default: do not include anything.  The parameters are taken into consideration only if the effective data format is JSON. |
| acc\_substitution=<policy> | The option controls how the bioseq info accession substation is done.  The supported policy values are:   * default: substitute if version value (version <= 0) or seq\_id\_type is Gi(12) * limited: substitute only if the resolved record's seq\_id\_type is GI(12) * never: the accession substitution is never done   If the substitution is needed then the seq\_ids list is analyzed. If there is one with Gi then it is taken for substitution. Otherwise an arbitrary one is picked.  Optional parameter. |

The following PSG protocol chunks will appear:



The sequence of chunks is not guaranteed.

In case of errors a MessageChunk will appear accompanied by the PSGFinalChunk.

## ID/get\_na Request

The format of the request:

http://<host:port>/ID/get\_na

where (see the[Common ID/... Request Parameters](#_Common_ID/..._Request) chapter as well):

|  |  |  |
| --- | --- | --- |
| Parameter | Description | |
| seq\_id=<seq\_id> | SeqId of the bioseq info to be retrieved (string).  Mandatory parameter. | |
| seq\_id\_type=<seq\_id\_type> | SeqId type of the bioseq info to be retrieved (integer > 0).  Optional parameter. | |
| names=<names> | A comma separated list of named annotations to be retrieved.  Mandatory parameter. | |
| use\_cache=<cache> | Allowed values:   * no: do not use LMDB cache (tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP) at all; go straight to Cassandra storage. * yes: do not use tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP from Cassandra storage at all. I.e., exclusively use the cache for all seq-id resolution steps. If the seq-id cannot be fully resolved through the cache alone, then code 404 must be returned.   Optional parameter.  By default (no use\_cache option specified), the behavior is to use the LMDB cache if at all possible; then, fallback to Cassandra storage. | |
| fmt=<format> | The format of the data sent to the client.  Supported values: json and native.  Optional parameter.  Default is json.  Note: at the moment JSON format is always used. | |
| tse=<tse\_opt> | | TSE option.  Return the following blobs depending on the value:   |  |  |  | | --- | --- | --- | | Value | ID2 split available | ID2 split not available | | none | Nothing | Nothing | | whole | All split blobs | All Cassandra data chunks of the blob itself | | orig | All Cassandra data chunks of the blob itself | All Cassandra data chunks of the blob itself | | smart | Split INFO blob only | All Cassandra data chunks of the blob itself | | slim | Split INFO blob only | Nothing |   Processors may also ignore the actual value of this option and treat it as “none”.  Optional parameter. Default value: none |
| client\_id=<client\_id> | | The client identifier (string).  If provided then the exclude blob feature takes place.  Optional parameter.  **Note**: see the [Exclude Blob API](#_Exclude_Blob_API) for more information |
| send\_blob\_if\_small=<# bytes> | | Integer >= 0  If [SERVER]/send\_blob\_if\_small config value is bigger of that then [SERVER]/send\_blob\_if\_small should be used.   * "tse" - value of {{tse}} URL parameter * "id2-split" -- whether the ID2-split version of the blob is available * "Small blob" -- size of the (compressed) blob data <= send\_blob\_if\_small * "Large blob" -- size of the (compressed) blob data > send\_blob\_if\_small  |  |  |  |  | | --- | --- | --- | --- | | tse | id2-split | Small blob | Large blob | | slim | no | Send original (non-split) blob data | Do not send original (non-split) blob data | | smart | no | Send original (non-split) blob data | Send original (non-split) blob data | | slim | yes | Send all ID2 chunks of the blob | Send only split-info chunk | | smart | yes | Send all ID2 chunks of the blob | Send only split-info chunk |   Optional parameter. Default: 0 |
| auto\_blob\_skipping=<value> | | The option to switch on/off automatic cache of already sent blobs to a particular client. If on then a blob will be skipped if it was already delivered to the client.  Acceptable values: yes and no.  Default: yes |

The response uses the PSG protocol.

The HTTP header Content-Type is set to “application/x-ncbi-psg”.

The HTTP header Content-Length is not set.

The HTTP 1.1 or HTTP/2 status code is always 200.



The sequence of chunks is not guaranteed. If

In case of errors a MessageChunk will appear accompanied by the PSGFinalChunk.

**NOTE**: A full bioseq info is sent to the client. The accession substitution will be done in accordance to the default substitution policy, see the ID/resolve request acc\_substitution parameter description.

## ID/accession\_version\_history Request

The format of the request:

http://<host:port>/ID/accession\_version\_history

where (see the[Common ID/... Request Parameters](#_Common_ID/..._Request) chapter as well):

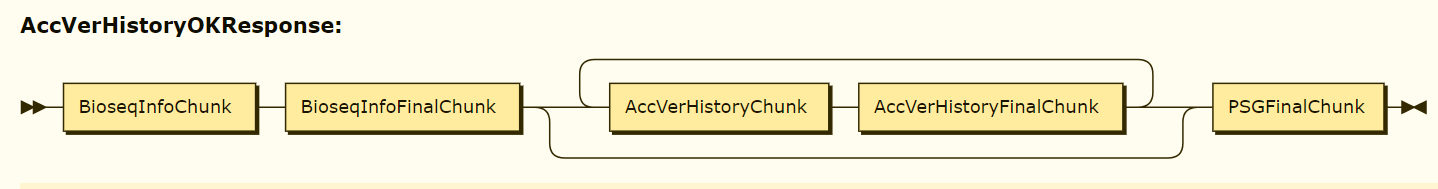
|  |  |
| --- | --- |
| Parameter | Description |
| seq\_id=<seq\_id> | SeqId of the bioseq info to be retrieved (string).  Mandatory parameter. |
| seq\_id\_type=<seq\_id\_type> | SeqId type of the bioseq info to be retrieved (integer > 0).  Optional parameter. |
| use\_cache=<cache> | Allowed values:   * no: do not use LMDB cache (tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP) at all; go straight to Cassandra storage. * yes: do not use tables SI2CSI, BIOSEQ\_INFO and BLOB\_PROP from Cassandra storage at all. I.e., exclusively use the cache for all seq-id resolution steps. If the seq-id cannot be fully resolved through the cache alone, then code 404 must be returned.   Optional parameter.  By default (no use\_cache option specified), the behavior is to use the LMDB cache if at all possible; then, fallback to Cassandra storage. |

The response uses the PSG protocol.

The HTTP header Content-Type is set to “application/x-ncbi-psg”.

The HTTP header Content-Length is not set.

The HTTP 1.1 or HTTP/2 status code is always 200.



The sequence of chunks is not guaranteed. If

In case of errors a MessageChunk will appear accompanied by the PSGFinalChunk.

**NOTE**: A full bioseq info is sent to the client. The accession substitution will be done in accordance to the default substitution policy, see the ID/resolve request acc\_substitution parameter description.

## ADMIN/config Request

The format of the request:

http://<host:port>/ADMIN/config

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If non-error reply then:

* The standard HTTP 1.1 or HTTP/2 protocol is used.
* The HTTP header Content-Type is set to “application/json”
* The HTTP header Content-Length is set approprietely
* The content is formed as a JSON dictionary with the following items:

|  |  |  |
| --- | --- | --- |
| Key | Value Type | Description |
| ConfigurationFilePath | String | Full path on the server local file system to the configuration file |
| Configuration | String | The full content of the configuration file the server started with |

## ADMIN/info Request

The format of the request:

http://<host:port>/ADMIN/info

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If non-error reply then:

* The standard HTTP 1.1 or HTTP/2 protocol is used.
* The HTTP header Content-Type is set to “application/json”
* The HTTP header Content-Length is set approprietely
* The content is formed as a JSON dictionary with the following items:

|  |  |  |
| --- | --- | --- |
| Key | Value Type | Description |
| PID | Integer | Server process PID |
| ExecutablePath | String | Full local file system path to the server executable |
| CommandLineArguments | String | Command line arguments exactly as the server was started including the binary name. |
| RealTime | Double or string | If succeeded then the process real time consumed as a double.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| UserTime | Double or string | If succeeded then the process user time consumed as a double.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| SystemTime | Double or string | If succeeded then the process system time consumed as a double.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| PhysicalMemory | Integer or string | If succeeded then the number of physical memory bytes available on the host as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedTotal | Integer or string | If succeeded then the number of total used memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedTotalPeak | Integer or string | If succeeded then the peak number of total used memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedResident | Integer or string | If succeeded then the number of resident memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedResidentPeak | Integer or string | If succeeded then the peak number of resident memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedShared | Integer or string | If succeeded then the number of used shared memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedData | Integer or string | If succeeded then the number of used data memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedStack | Integer or string | If succeeded then the number of used stack memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedText | Integer or string | If succeeded then the number of used text memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedLib | Integer or string | If succeeded then the number of used library memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| MemoryUsedSwap | Integer or string | If succeeded then the number of used swap memory bytes as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| ProcFDSoftLimit | Integer or string | If succeeded then the process file descriptor soft limit as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| ProcFDHardLimit | Integer or string | If succeeded then the process file descriptor hard limit as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| ProcFDUsed | Integer or string | If succeeded then the number of used file descriptors as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| CPUCount | Integer | The number of CPUs on the host. |
| ProcThreadCount | Integer or string | If succeeded then the number of threads the process uses as an integer.  In case of an error getting the value from the OS then a string with a fixed value “n/a”. |
| Version | String | Package version X.Y.Z  0.0.0 if built outside of the prepare\_release framework. |
| BuildDate | String | Build timestamp. Format:  MMM DD YYYY HH:mm:SS |
| StartedAt | String | Local time when the server started. Format:  MM/DD/YYYY HH:mm:SS |
| ExcludeBlobCacheUserCount | Integer | The number of users which have records in the exclude blob cache |

## ADMIN/status Request

The format of the request:

http://<host:port>/ADMIN/status

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If non-error reply then:

* The HTTP header Content-Type is set to “application/json”.
* The HTTP header Content-Length is set appropriately.
* All the event counters (errors, request counters etc.) are monotonically growing and are set to 0 at the server instance startup.
* The content is formed as a JSON dictionary with the following items:

|  |  |  |
| --- | --- | --- |
| Key | Value Type | Description |
| CassandraActiveStatementsCount | Integer | The current number of active Cassandra statements. |
| NumberOfConnections | Integer | The current number of connections to the server. |
| ActiveRequestCount | Integer | The number of active pending requests (does not include admin or test requests). |
| ShutdownRequested | Boolean | true if a graceful shutdown was requested |
| GracefulShutdownExpiredInSec | Integer | If ShutdownRequested is true tehn the number of seconds left till the graceful shutdown timeout is over. |
| BadUrlPathCount | Integer | The total number of bad URL path requests. |
| InsufficientArgumentsCount | Integer | The total number of requests with insufficient argumens. |
| MalformedArgumentsCount | Integer | The total number of requests with malformed arguments. |
| GetBlobNotFoundCount | Integer | The total number of requests when a requested blob is not found. |
| UnknownErrorCount | Integer | The total number of requests when an unknown error was encountered. |
| ClientSatToSatNameErrorCount | Integer | The number of errors when the client supplied a blob sat however the sat could not be mapped to a Cassandra keyspace. |
| ServerSatToSatNameErrorCount | Integer | The number of errors when the server data are referring to a sat which could not be resolved to a Cassandra keyspace. |
| BioseqID2InfoErrorCount | Integer | The number of errors when ID2 info data field is invalid. |
| BlobPropsNotFoundErrorCount | Integer | The number of errors when blob properties were not found. |
| LMDBErrorCount | Integer | The number of errors when the LMDB lookup failed. |
| CassQueryTimeoutErrorCount | Integer | The number of errors when there was a Cassandra request execution timeout. |
| TotalErrorCount | Integer | The total number of requests with any kind of error encountered. |
| InputSeqIdNotResolved | Integer | The number of times the user requested SeqId was not resolved. |
| AdminRequestCount | Integer | The total number of successful requests for and administrative information. |
| ResolveRequestCount | Integer | The total number of successful requests to resolve an accession. |
| GetBlobBySeqIdRequestCount | Integer | The total number of successful requests to get a blob by SeqId. |
| GetBlobBySatSatKeyRequestCount | Integer | The total number of successful requests to get a blob by sat and sat key. |
| GetNamedAnnotationsCount | Integer | The total number of the successful get\_na requests. |
| TestIORequestCount | Integer | The total number of the io requests. |
| TotalRequestCount | Integer | The total number of requests. |
| Si2csiCacheHit | Integer | The total number of times when the si2csi cache had the required data. |
| Si2csiCacheMiss | Integer | The total number of times when the si2csi cache didn’t have the required data. |
| BioseqInfoCacheHit | Integer | The total number of times when the bioseq info cache had the required data. |
| BioseqInfoCacheMiss | Integer | The total number of times when the bioseq info cache didn’t have the required data. |
| BlobPropCacheHit | Integer | The total number of times when the blob properties cache had the required data. |
| BlobPropCacheMiss | Integer | The total number of times when the blob properties cache didn’t have the required data. |
| Si2csiNotFound | Integer | The total number of times when no data were found when the si2csi Cassandra table was looked through. |
| Si2csiFoundOne | Integer | The total number of times when exactly one record was found when the si2csi Cassandra table was looked through. |
| Si2csiFoundMany | Integer | The total number of times when more than one records ware found when the si2csi Cassandra table was looked through. |
| BioseqInfoNotFound | Integer | The total number of times when no data were found when the bioseq\_info Cassandra table was looked through. |
| BioseqInfoFoundOne | Integer | The total number of times when exactly one record was found when the bioseq\_info Cassandra table was looked through. |
| BioseqInfoFoundMany | Integer | The total number of times when more than one records ware found when the bioseq\_info Cassandra table was looked through. |
| Si2csiError | Integer | The total number of errors when the si2csi Cassandra table was looked through. |
| BioseqInfoError | Integer | The total number of errors when the bioseq\_info Cassandra table was looked through. |

## ADMIN/shutdown Request

The format of the request:

http://<host:port>/ADMIN/shutdown?

where

|  |  |
| --- | --- |
| Parameter | Description |
| username=<username> | The user name who wanted to do the shutdown (string).  At the moment the parameter is used only for logging.  Optional parameter.  Default: empty string. |
| auth\_token=<token> | Authorization token (string).  If the configuration [ADMIN]/auth\_token value is provided then the request must have the token value matching the configured to be granted.  Optional parameter.  Default: empty string. |
| timeout=<timeout> | The timeout in seconds within which the shutdown must be performed (integer).  If 0 then it leads to an immediate shutdown.  If 1 or more seconds then the server will reject all new requests and waits till the timeout is over or all the pending requests are completed and then do the shutdown.  Optional parameter.  Default: 10 (seconds) |

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used. In case of errors:

|  |  |
| --- | --- |
| PSG protocol status code | Description |
| 409 | The previous shutdown request is shorter |
| 400 | Invalid timeout |
| 401 | Unauthorized |
| 500 | Internal error |

If it is a non-error reply then:

* The standard HTTP 1.1 or HTTP/2 protocol is used.
* The HTTP header Content-Type is set to “text/plain”.
* The HTTP header Content-Length is set appropriately.

|  |  |
| --- | --- |
| HTTP 1.1 or HTTP/2 status code | Description |
| 202 | Shutdown request has been successfully accepted |

The content may have the corresponding message.

## ADMIN/get\_alerts Request

The format of the request:

http://<host:port>/ADMIN/get\_alerts

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If it is a non-error reply then:

* The HTTP header Content-Type is set to “application/json”.
* The HTTP header Content-Length is set approprietly.
* The content has a JSON dictionary which describes the current alerts, both acknowledged and not.

## ADMIN/ack\_alert Request

The format of the request:

http://<host:port>/ADMIN/ack\_alert?

where

|  |  |
| --- | --- |
| Parameter | Description |
| alert=<alert\_id> | The alert identifier to acknowledge (string)  Mandatory parameter |
| username=<username> | The user name who acknowledges the alert (string).  The parameter is used only for logging.  Mandatory parameter |

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If it is a non-error reply then:

* The HTTP header Content-Type is set to “text/plain”.
* The HTTP header Content-Length is set appropriately.

|  |  |
| --- | --- |
| HTTP 1.1 or HTTP/2 status code | Description |
| 200 | The alert has been acknowledged or had already been acknowledged before. |

* The content may have the corresponding message.

## ADMIN/statistics Request

The format of the request:

http://<host:port>/ADMIN/statistics?

where

|  |  |
| --- | --- |
| Parameter | Description |
| reset=<yes or no> | If provided as tes then the collected statistics is rest. Otherwise the collected statistics is sent to the client.  Default: no  Optional parameter |
| most\_recent\_time=<time> | Number of seconds for the most recent time range limit.  See more below |
| most\_ancient\_time=<time> | Number of seconds for the most ancient time range limit.  See more below |
| histogram\_names | Comma separated list of the histogram names.  If provided then the server returns all existing histograms (listed in histogram\_names) which intersect with the specified time period. |

Time limits mean "so many seconds ago from the current time".

If the histogram names are not provided then the server sums up the histograms which intersect the specified time range. Otherwise all the intersected histograms will be sent for the provided histogram names.

If both most\_recent\_time and most\_ancient\_time are specified, then their order is not important (will be auto-reordered if needed)

If only most\_recent\_time is specified, then assume most\_ancient\_time as infinite.

If only most\_ancient\_time is specified, then assume most\_recent\_time equal to 0.

The actual time period for which the data is returned is passed back to the client.

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If it is a non-error reply then:

* The HTTP header Content-Type is set to “application/json”.
* The HTTP header Content-Length is set approprietly.
* The content has a JSON dictionary which stores all the collected statistics since the server restart or the last reset.

## TEST/io Request

The server responses to this request only if the configuration file has the [DEBUG]/psg\_allow\_io\_test value set to true.

The format of the request:

http://<host:port>/TEST/io?

where

|  |  |
| --- | --- |
| Parameter | Description |
| return\_data\_size=<data\_size> | Size in bytes (positive integer up to 1000000000) which should be sent to the client. The data are random.  Mandatory parameter. |
| log=<log> | Boolean parameter which tells if the logging of the request is done or not.  Accepted values are yes and no.  Optional parameter.  Default: no |

Response:

In case of errors a PSG protocol reply is sent otherwise the standard HTTP 1.1 or HTTP/2 protocol is used.

If it is a non-error reply then:

* The HTTP header Content-Type is set to “application/octet-stream”.
* The HTTP header Content-Length is set to data\_size.
* The content will have the data\_size of randomly generated bytes.

## health Request

The format of the request:

http://<host:port>/health

http://<host:port>/deep-health

The URL is for regular checks various monitoring tools may want to perform. The server will do the following upon receiving the request:

* Check the Cassandra DB status. If the DB is not in operational state then the server sends an HTTP reply with status 500. The message explains the reason in more details.
* Checks if the cache files are configured. If not then the server sends an HTTP reply with status 200.
* Resolves the configured ([HEALTH]/test\_seq\_id) seq\_id in cache only.
  + If the resolution succeeded then the server sends an HTTP reply with status 200
  + If the resolution failed by any reason then the server checks the configured option ([HEALTH]/test\_seq\_id\_ignore\_error) if the error should be ignored

If the error is to be ignored then the server sends an HTTP reply with status 200. Otherwise it sends an HTTP reply with status 500.

## deep-health Request

At the moment the deep-health request is an equivalent of the health request (see description above).

## favicon.ico Request

The format of the request:

http://<host:port>/favicon.ico

The request is often sent by various browsers so the PSG server implements a response to it.

Response:

The standard HTTP 1.1 or HTTP/2 protocol is used and an image is sent to the client.

## Unknown URL Request

Response:

The PSG protocol is used. The status is 200.

The HTTP header Content-Type is set to “application/x-ncbi-psg”.

The HTTP header Content-Length is not set.

The response body has two chunks:



# Cassandra Database

The database structure is described here:

<https://confluence.ncbi.nlm.nih.gov/display/CT/Design+extended+schemata+for+PubSeq+data+in+Cassandra+and+LMDB>

# Monitoring and Maintenance

The server code uses the standard C++ Toolkit logging. So logging configuration is the same as for all other NCBI C++ written applications.

The server also exposes some internal events monotonically growing counters, status and configuration information. This information is available via /ADMIN/<item> requests (see the request description above). It is also possible to shutdown the server using a URL request.

At the moment the alerts infrastructure is not supported.

# Files Architecture

The diagram below shows the files used by Pubseq Gateway server (LMDB cache files are not shown as they are not read directly).



Pubseq Gateway reads its configuration file (usually named pubseq\_gateway.ini) and configures data access and internal structures correspondingly.

Due to a predicted high load on the server there will be no extensive logging on every event. The only warnings and errors are going to be logged.

# Client API

The client API is available here:

<https://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/include/objtools/pubseq_gateway/client/psg_client.hpp>

# Command Line Arguments

The table below describes the server command line arguments.

|  |  |
| --- | --- |
| Argument | Description |
| -help | Prints the help message and exits. |
| -daemonize | If given then the server does the daemonization. By default the server does not daemonize. |
| -version | Prints the server version and exits. |
| -version-full | Prints the server version, the storage version and the protocol version and then exits. |
| -logfile | The file to which the server log should be redirected. |
| -conffile | The file from which the server should read the configuration. |

# Signal Handling

The table below describes the signal handling in the server.

|  |  |
| --- | --- |
| Signal | Description |
| INT | Immediate shutdown which will interrupt all currently executed requests. |
| TERM | Gracefull shutdown.  The server will wait till all the currently executed requests are finished. During that time all new requests are rejected. When all requests are finished, the server will shut down.  Will be called e.g. on the host reboot. |
| QUIT | immediate coredump ( most convenient for debugging, Ctrl-\ |
| HUP USR1 USR2 WINCH | The signals are intercepted and logged with no other actions. |

# Configuration Parameters

Pubseq Gateway reads the configuration from a file. The default name of the server is pubseq\_gateway so (if the –conffile command line argument is not provided) the default configuration file name will be pubseq\_gateway.ini.

The configuration file uses the NCBI standard ini file format with sections and values within sections. The sections below describe each section of the configuration file separately.

## [LMDB\_CACHE] Section

|  |  |
| --- | --- |
| Value | Description |
| dbfile\_si2csi | Path to the file where an LMDB si2csi cache file is located.  If not provided then no cache will be used. |
| dbfile\_bioseq\_info | Path to the file where an LMDB bioseq\_info cache file is located.  If not provided then no cache will be used. |
| dbfile\_blob\_prop | Path to the file where an LMDB blob\_prop cache file is located.  If not provided then no cache will be used. |

## [SERVER] Section

|  |  |
| --- | --- |
| Value | Description |
| port | HTTP port (1...65534)  No default. If port is not specified or is out of range the server will not start |
| workers | Number of HTTP workers (1...100)  Default: 32 |
| backlog | Listener backlog (5...2048)  Default: 256 |
| maxconn | Max number of connections (5...65000)  Default: 4096 |
| optimeout | Operation timeout in milliseconds  Default: 30000 |
| maxretries | Max Cassandra operation retries  Default: 1 |
| log | If set to true then request contexts will be created for each request.  It does not affect logging macros. |
| root\_keyspace | Cassandra root keyspace which is used for discovering the sat to keyspace mapping as well as the location of the SI2CSI and BIOSEQ\_INFO tables  Default: sat\_info |
| send\_blob\_if\_small | In most cases the blobs are not split because they are... just too small to be split. So, in the spirit of the "slim/smart" purpose such original blobs should be sent to the client.  Default: 10KB |
| max\_hops | The number of maximum allowed hops for the ID requests. If exceeded then the request is rejected.  The accepted value is integer and must be greater than zero.  Default: 2 |

## [AUTO\_EXCLUDE] Section

The section describes settings for the exclude blob cache feature.

|  |  |
| --- | --- |
| Value | Description |
| max\_cache\_size | Cache size per client.  0 - means it is disabled.  Default: 1000 |
| purge\_percentage | The percentage of the records to purge (of max\_cache\_size; 0 <= int <= 100)  Default: 20 |
| inactivity\_purge\_timeout | Client inactivity purge timeout, seconds, integer.  Used for garbage collecting  Default: 60 |

## [ADMIN] Section

|  |  |
| --- | --- |
| Value | Description |
| auth\_token | Authorization token for the shutdown request.  If provided then the URL shutdown request must have the corresponding auth\_token parameter to be authorized.  If not provided then any URL shutdown request will be authorized. |

## [STATISTICS] Section

|  |  |
| --- | --- |
| Value | Description |
| small\_blob\_size | The statistics for the blob retrieving timing is collected depending on the blob sizes in separate bins. The first bin covers the range of sizes from 0 till small\_blob\_size inclusive. Then the ranges will start from the power of 2.  Default: 16 |
| min | Min time value (microseconds)  Default: 1 |
| max | Max time value (microseconds)  Default: 16 \* 1024 \* 1024 |
| n\_bins | Number of bins  Default: 24 |
| type | Scale type  Can be "log" or "linear"  Default: log |

## [DEBUG] Section

|  |  |
| --- | --- |
| Value | Description |
| diag\_post\_level | The level of messages which will be in the log file  Note: the value “Trace” will have no effect; in this case the value will taken as “Info”. The trace is controlled independently in the diag\_trace option |
| psg\_allow\_io\_test | If set to true then psg will respond to TEST/io URL sending back up to 1000000000 bytes |
| diag\_trace | If present in the config file with any value (the datatype is string for this option) then PSG\_TRACE and ERR\_POST(Trace …) will be populated in the log file |

## [CASSANDRA\_DB] Section

|  |  |
| --- | --- |
| Value | Description |
| Ctimeout | Connection timeout in ms  Default: 30000 |
| qtimeout | Query timeout in ms  Default: 5000 |
| namespace | Data namespace  Default: empty string |
| fallbackrdconsistency | Fallback read consistency  Default: false |
| fallbackwriteconsistency | Lower down consistency of BD write operations if local quorum can't be achieved.  0 - default cassandra driver behavior  Default: 0 |
| loadbalancing | Load balancing policy. Accepted values are: DCAware, RoundRobin  Default: DCAware |
| tokenaware | Enables TokenAware routing  Default: true |
| latencyaware | Enables LatencyAware routing  Default: true |
| numthreadsio | Number of io threads to async processing (1...32)  Basically it is how many blob chunks are requested simultaneously per request. For example, if a blob with 100 chunks is requested then 4 select statements will be created and maintained simultaneously.  Default: 4 |
| numconnperhost | Number of connections per node (1...8)  Default: 2 |
| maxconnperhost | Maximum count of connections per node (1...8)  Default: 4 |
| keepalive | TCP keep-alive the initial delay in seconds  Default: 0 |
| password\_file | Cassandra password file and a section where credentials are stored. If a password\_file is not provided then password\_section value is ignored.  Default: empty, i.e. no user/password combination is used. |
| password\_section |
| service | The value is a load balancer name or a list of host[:port] items (‘,’ or ‘ ‘ separated).  If the value has neither ‘ ‘, nor ‘,’ nor ‘:’ then it is treated as a load balancer name. The load balancer resolved host ports are are sorted in accordance with their rates.  The list of host[:port] items, regardless where it came from – directly from a parameter value or from a load balancer – is analyzed further. The analizis checks that if ports are provided then they are the same. If the port is provided then it is used for the Cassandra cluster. If no ports are provided then the Cassandra driver uses its default one. |
| cassandralog | Tells if the Cassandra drivers should produce log records.  If switched on then the records go into the same destination as the rest of the server logging. The logging level matches the one configured for the application.  Default: false |

## [CASSANDRA\_PROCESSOR] Section

|  |  |
| --- | --- |
| Value | Description |
| enabled | Tells if the Cassandra/LMDB processors are enabled if a request URL does not specify it explicitly.  Default: 1 |

## [COUNTERS] Section

The section lets to configure what name and what description the server will send to the client for each statistics counter and histogram in the corresponding requests.

All the values are optional and have the following format:

<Item ID> = <Name>:::<Description>

Where:

* Item ID is an identifier of a counter or a histogram the server sends to the client
* Name is a string which is used by GRID Dashboard to display the item
* Description is a string which is used by GRID Dashboard to show a tooltip for the corresponding item

## [HEALTH] Section

The section configures the server behavior for the /health and /deep-health URLs.

|  |  |
| --- | --- |
| Value | Description |
| test\_seq\_id | The seq\_id to be resolved within the procedure of checking the server health.  Default: gi|2 |
| test\_seq\_id\_ignore\_error | Specifies what to do if the resolution of the [HEALTH]/test\_seq\_id failed by any reason. Supported values are: true and false.  If the value is true then the server will reply with status 200 even if there was a resolution error.  Default: true |

## [SSL] Section

The section configures the server https configuration.

|  |  |
| --- | --- |
| Value | Description |
| ssl\_enable | Lets to switch on/off the https support  Note: certificate and private key files must be provided if switched on  Default: false |
| ssl\_cert\_file | Path to the certificate file. Must be provided for https.  Default: no default |
| ssl\_key\_file | Path to the private key file. Must be provided for https.  Default: no default |
| ssl\_ciphers | SSL ciphers. Optional  Default: TLSv1.2 |

# Appendix

## PSG Processors

The PSG server internal infrastructure is built around a few major entities: PSG requests, PSG replies and PSG processors. An instance of a PSG request (CPSGS\_Request class) and a PSG reply (CPSGS\_Reply class) are created for each incoming request. These objects are shared between all the PSG processors which are supposed to handle the requests.

A PSG processor is a class which implements an interface defined in the IPSGS\_Processor class. Each processor class must register itself in the PSG infrastructure, see the method void CPubseqGatewayApp::x\_RegisterProcessors(void). An important detail here is the sequence of registering a processor. The earlier a processor is registered the higher priority is assigned to it. The priorities are unique so there are no processors with the same priority. Basically the priority is an integer value and the greater value means higher priority.

The IPSGS\_Processor interface is a self factory. When a processor is registered its default constructor is used. So the default constructor is expected to be lightweight. Later, for each incoming user request the PSG server infrastructure will do the following:

* Do the incoming request sanity check. In case of problems finishes the request. If everything looks fine then the PSG request and PSG reply objects are initialized and ready for use.
* For each registered processor the factory method IPSGS\_Processor\* CreateProcessor(shared\_ptr<CPSGS\_Request>, shared\_ptr<CPSGS\_Reply>, TProcessorPriority) const will be called. The agreement is that if a processor cannot handle the request it should return nullptr. Otherwise a processor instance capable of handling the request is returned. If an instance is returned then the infrastructure will take case about the instance lifetime.
* At this moment the infrastructure will have an arbitrary long list of processors which can process a request. For each of them the Process() method will be called in parallel threads.

After calling the prcessors Process() method the communacations between the infrastructure and each of the running processor is asynchronous. The infrastructure can call the following methods:

* GetStatus().  
  In response to this call a processor should tell its current status which can be one of the following:

|  |  |
| --- | --- |
| ePSGS\_InProgress | Processor is still working |
| ePSGS\_Found | Processor finished and found what needed |
| ePSGS\_NotFound | Processor finished and did not find anything |
| ePSGS\_Error | Processor finished and there was an error |
| ePSGS\_Cancelled | Processor finished because earlier it received the Cancel() call |

* Cancel()  
  It is a request to cancel processing. It is not required to stop processing right away because a processor may have ongoing asynchronous communications with other services. After requesting a cancellation the infrastructure will periodically call GetStatus() to understand when the processor finished dealing with the request.
* GetName()  
  In response to this call a human readable processor name should be provided for the purpose of logging and tracing
* ProcessEvent()  
  The infrastructure may do this call when an event happened which may require to have some processing. It also can be called on a timer event. This method is not mandatory and by default does nothing.

A processor, in turn, can use the following IPSPS\_Processor methods to communicate with the infrastructure:

* SignalStartProcessing()  
  This method informs the infrastructure that the processor started to receive data and most probably will be able to complete the request. Upon receiving this call the infrastructure will cancel all the other processors.  
  Due to a parallel nature of handling the requests it is possible that two processors will start to receive data at about the same time. So the SignalStartProcessing() call returns a flag weather the processor should continue or stop.
* SignalFinishProcessing()  
  This method should be called when the processor decides tha there is nothing else to do.

There are a few assumptions about the processors’ behavior:

* The server replies use the PSG chunk protocol and many processors may want to send something to the client simultaneously. To handle that a raw access to the connection should not be used. Instead the instance of the CPSGS\_Reply (saved in the m\_Reply member) should be used.
* When a processor finishes it should call SignalProcessorFinished()
* If a processor needs to do anything with logging then the processor needs to do the following:
  + Set request context for the current thread
  + Use one of the macro PSG\_TRACE, PSG\_INFO, PSG\_WARNING, PSG\_ERROR, PSG\_CRITICAL, PSG\_MESSAGE (see pubseq\_gateway\_loggig.hpp)
  + Reset request context

This could be done like this:  
{  
 CRequestContextResetter context\_resetter;  
 m\_Request->SetRequestContext();  
 . . .  
 PSG\_WARNING("Something");  
}

* A request may have a parameter which requests tracing. In this case it is a good idea to send to the user the trace PSG chunks. This could be done as follows:  
  if (m\_Request->NeedTrace()) {  
   m\_Reply->SendTrace(“A message”,  
   m\_Request->GetStartTimestamp());  
  }  
  The timestamp parameter is required so that the client receives the information when the even has happened relative to the beginning of the request handling.

## PSG Requests

The PSG processors deal with a high level object (CPSGS\_Request class) which wraps the incoming user request. The object stores all the incoming parameters and sometimes a bit more. The PSG processors should not generally access the lower level connection and should rely on CPSGS\_Request instead.

The request object can tell the type of a particular request it wraps and then the PSG prcessor can request a conversion to a particular request (see the CPSGS\_Request::GetRequest() method).

All but annotation requests are essentially a collection of incoming url parameters. The annotation request add some more functionality. To explain the added functionality let’s consider an example. Suppose that the user requested 5 named annotations. The infrastructure visited all the registered processors and two of them reported that they can hanle the request. Suppoose that the first two annotations can be retrieved only by the first processor while the last two annotations can be retrieved only by the second processor. The third annotation however can be retrieved by both processors. Basically the distribution of what can be retrieved by what processors can be arbitrary.

The infrastructure does not have information of what processor can potentially retrieve what. The processors neither have info of how many parallel processors are handling the same request and what they are going to retrieve. The common ground is the annotation request which holds the full list of all the user requested annotations. So the annotation request object provides an interface for the processors to deal with the retrieved annotations. When a processor receives an annotation from a data source and before sending it the processor should do the following:

* Call the SPSGS\_AnnotRequest:: RegisterProcessedName() providing the annotation name and the processor own priority
* The return of the call will report the priority of the competitive parallel processor
  + If the other processors have not sent this annotation yet then the kUnknownPriority constant will be returned. The processor should proceed with sending the annotation
  + Otherwise the reported priority of a competitive processor should be compared with the processor own priority. Only if the processor own priority is higher the annotation should be sent to the user.
* When a processor knows that there will be no more annotations retrieved from the backend it should call the processor base class SignalFinishProcessing() method

In the case of annotation processors they should not call the processor base class SignalStartProcessing() method. This method will lead to cancelling all the parallel processors while the cooperative execution is expected via the annotation request object.

In case of all the other requests the processors should call SignalStartProcessing() method because:

* There is only one piece of data to be delivered (unlike named annotations where there is a list of names which may come from different backends)
* The data are assumed to be the same regardless of the backend
* It is assumed that if a processor started to receive data earlier than the other it will also finish earlier

## PSG Timing Collecting

Sometimes it is convenient to see the timing of a certain operation (e.g. successful lookup in a cache of secondary seq id, or the same lookup in a database) on GUI in a form of a chart. PSG server has in infrastructure to collect such data and then report them to the client via certain requests so that the charts are shown. The collected data from the server are transferred to the client in response to the /ADMIN/statistics requests as a JSON dictionary. The GRID dashboard implements a web interface to request the data and then show it. It may look as follows:



Inside the server all timing is collected in integer microseconds and two corelib classes are used as the actual storage: CHistogram and CHistogramTimeSeries. This is done to support average timing in the past in growing intervals.

To add a new timing the following changes need to be done:

* Introduce a new operation enumeration member in enum EPSGOperation (timing.hpp). Typically that would be an id for an operation which can complete successfully or fail. For this example let’s call it eMyLookup
* Introduce a new timing class which derives from CPSGTimingBase. This will let you to have a customized initialization of the timing histogram storage. Some timing parameters are coming from the configuration file so the class constructor should accept them. Also, the config file may have the timing parameters misconfigured so the class should be able to reset the settings to default values.  
  So the example class may closely follow the implementation of e.g. CCassResolutionTiming (timing.hpp, timing.cpp).  
  Let’s call the example class CMyLookupTiming
* Create a new member in the COperationTiming class (timing.hpp) where the values are going to be stored, e.g.:  
  vector<unique\_ptr<CMyLookupTiming>> m\_MyLookupTiming;  
  The vector will have two items. One is for the successful operation (found) and one is for unsucessfull (not found).
* Initialize the m\_MyLookupTiming vector in the COperationTiming constructor (timing.cpp)
* The UI on the client side needs to display the data with proper labels and also needs to refer to a particular timing info when some detailed information is requested. So a correspondence between an identifier and its storage plus human readable description should be created. It is also done in the COperationTiming constructor (timing.cpp), see the m\_NamesMap member initialization. For our example two items need to be inserted into that map, e.g.:  
  . . .  
  { “MyLookupFound”,  
   SInfo(m\_MyLookupTiming[0].get(), “My lookup found”, “The timing of that lookup when a record is found”) },  
  { “MyLookupNotFound”,  
   SInfo(m\_MyLookupTiming[1].get(), “My lookup not found”, “The timing of that lookup when a record is not found”) },  
  . . .
* The code which actually does the lookup needs to register the corresponding timing. To do so there is the COperationTiming::Register(. . .) method which will register the timing in the necessary storage depending on the operation and its outcome (found or not found). So the method needs to be adjusted (timing.cpp, see the switch statement).
* To support the collected data displayed on the time auxis a few COperationTiming methods needs to be trivially extended (timing.cpp):  
  COperationTiming::Rotate() needs to pass the Rotate() call to m\_MyLookupTiming  
  COperationTiming::Reset() needs to pass the Reset() call to m\_MyLookupTiming

With these changes in place the server processor will be able to make calls like

TPSGS\_HighResolutionTimePoint start = chrono::high\_resolution\_clock::now();  
. . .  
auto app = CPubseqGatewayApp::GetInstance();  
app->GetTiming().Register(eMyLookup, eOpStatusFound, start);

At this moment the server side functionality is completed. It will collect the new timing and properly serializes it for the client in response to the requests. The serialization is done in JSON.

It is up to the client how to display the data. One of the options is to use the GRID dashboard. If so it would require some changes in the CGI to display the charts. Please create a JIRA ticket to add this functionality.

## Exclude Blob API

The exclude blob API is supposed to be used by the PSG processor developers to implement the exclude blob cache feature (see the description in [Exclude Blob Cache](#_Exclude_Blob_Cache)).

The class which implements the building blocks is called CExcludeBlobCache and located in exclude\_blob\_cache.hpp and exclude\_blob\_cache.cpp files. An instance of it is created on the application level so it can be shared between the processors. The following is the way to get access to the CExcludeBlobCache instance:

auto \* app = CPubseqGatewayApp::GetInstance();  
app->GetExcludeBlobCache()->...

The cache operates on a basis of a client\_id. Thus if the request comes with an empty client\_id then all the interactions with the cache should be omitted. Also, if a blob is requested by sat/sat\_key it means that the client knows for sure that it needs the blob regardless if it was already sent to the client. So the sat/sat\_key requested must be sent regardless of what cache reports.

Before sending a blob to the client a processor may want to call AddBlobId(...) method which can:

* tell if the blob is already in cache; if so then if the blob is in process of transferring to the client or completely transferred to the client
* if not in cache then register the blob as the one which is in the process of transferring to the client

So if the blob is already in cache then the processor should not send the blob to the client. Instead the processor should send a message reporting why the blob is not sent: in progress of sending by another processor or has already been transferred.

If a processor decides to send the blob to the client then it should inform the cache about the blob status when the blob processing is completed. If the blob has been successfully transferred to the client then the processor should call the SetCompleted(…) method. If there was an error while the blob was transferred then the processor should call the Remove(…) method.

## Alerts API

The PSG processors can use alerts API to signal certain conditions. The PSG processors can set the alert on and it will stay on till the user (supposedly an administrator) acknowledges it. Also the last alert timestamp and the total counter are memorized together with an alert.

To introduce a new alert the following would need to be done:

* Add a new member into the EPSGS\_AlertType enumeration to uniquely identify the new alert (alerts.hpp), e.g. ePSGS\_ItWentWrong.
* Make a correspondace between the ePSGS\_ItWentWrong and the alert string identifier. This should be done in alerts.cpp via adding another member into kAlertToIdMap, e.g.  
  . . .  
  { ePSGS\_ItWentWrong, "ItWentWrong" },  
  . . .

After that the PSG processor will be able to register a new alert using the following call:

auto \* app = CPubseqGatewayApp::GetInstance();  
app->GetAlerts().Register(ePSGS\_ItWentWrong, “My alert message”);

The server should have one alert storage per instance so the actual storage is in the application instance class.

The GRID dashboard supports alerts so they can be visible in the web UI. GRID dashboard also supports alert acknowledging. The user who acknowledged an alert and the time of acknowledging will also be recorded.

## Event Counter API

The PSG processors may also want to support counters of some events, e.g. the number of successfully retrieved annotations. There server offers a generic API to expose such counters for monitoring purposes.

There are a few different kind of counters:

* Monotonically incremented counters. These are for events like a number of incoming requests.
* Summarizing counters like a sum of all requests regardless of the request type
* Incremented and decremented counters like the number of currently active database connections.

The server offers an easy way to add monotonically incremented counters so that they are automatically exposed to an external user interface like GRID Dashboard. To introduce a new counter the following needs to be done:

* Add a new member of an enumeration CPSGSCounters::EPSGS\_CounterType in pubseq\_gateway\_stat.hpp to identify the new counter, e.g. ePSGS\_MyCounter.
* Add a description of the counter in the CPSGSCounters::CPSGSCounters() constructor in pubseq\_gateway\_stat.cpp

After that the following calls can be used to increment the counter:

auto \* app = CPubseqGatewayApp::GetInstance();  
app->GetCounters().Increment(CPSGSCounters:: ePSGS\_MyCounter);

## Asynchronous socket polling API

PSG uses libuv loop to handle asynchronous events. There is an API which processors can use to poll a socket asynchronously using the libuv facilities. The behavior is mostly defined by how libuv is implemented.

The implementation of the API is in psgs\_io\_callbacks.\*pp files. The basic idea behind the API is three callbacks which are initialized in a callback object constructor:

* The constructor of the callback object:  
  CPSGS\_SocketIOCallback(int fd, EPSGS\_Event event,  
   uint64\_t timeout\_millisec,  
   void \* user\_data,  
   TEventCB event\_cb,  
   TTimeoutCB timeout\_cb,  
   TErrorCB error\_cb);  
  Where  
  fd is a socket file descriptor which will be polled  
  event is one of the following: read, write, disconnect, prioritized  
  timeout\_millisec is a timeout for waiting the event  
  user\_data is an arbitrary data a processor wants to get when a callback is called  
  the last three items are the callbacks
* The event callback  
  function<EPSGS\_PollContinue(void \* user\_data)>  
  It will be called when the requested event happened.  
  The user\_data argument is what was supplied in the constructor.  
  The return value of the callback tells what to do with polling. There are two options: to continue or to stop polling. If the user chooses to continue polling then the timer is restarted and polling continues. Otherwise both the timer and polling are stopped and no more callbacks will be invoked.
* The error callback  
  function<EPSGS\_PollContinue(const string & message, void \* user\_data)>  
  It will be called in case libuv detects an error.  
  The user\_data argument is what was supplied in the constructor. The message argument is a string associated with the particular error.  
  The return value of the callback tells what to do with polling. There are two options: to continue or to stop polling. If the user chooses to continue polling then the timer is restarted and polling continues. Otherwise both the timer and polling are stopped and no more callbacks will be invoked.
* The timeout callback  
  function<EPSGS\_PollContinue(void \* user\_data)>;  
  It will be called when timeout of waiting for the requested event occurred.  
  The user\_data argument is what was supplied in the constructor.  
  The return value of the callback tells what to do with polling. There are two options: to continue or to stop polling. If the user chooses to continue polling then the timer is restarted and polling continues. Otherwise both the timer and polling are stopped and no more callbacks will be invoked.

There is no limit on how many callback objects a processor can create.

It is a PSG processor responsibility to take care of the callback object lifetime. The callback object should not live longer than the callback functions, otherwise a callback will be invoked when the corresponding object has potentially already been deleted.

It is safe to keep an instance of a callback object after returning ePSGS\_StopPolling in any of the callbacks. Also the callback object destructor takes care of removing a timer and stopping polling in libuv.

## Handling Non-libuv Events Asynchronously

In some cases a processor may want to wait for the events which cannot be incorporated to the libuv event loop. At the same time a processor should meet two conditions:

* The control flow in the processing method must be released to the infrastructure while waiting for the required events. This ensures that the processor does not block the other processors from working and does not hold the resources which can be used for the other requests (like worker threads)
* When the required event has come the processing should be done from a libuv call. Otherwise an unpredicted behavior may be encountered when a processor would start using shared objects like CPSGS\_Reply.

To illustrate a solution let’s consider a test processor.

class TestProc : public IPSGS\_Processor

{

public:

virtual IPSGS\_Processor\*

CreateProcessor(shared\_ptr<CPSGS\_Request> request,

shared\_ptr<CPSGS\_Reply> reply,

TProcessorPriority priority) const;

virtual void Process(void);

virtual void Cancel(void) {}

virtual EPSGS\_Status GetStatus(void);

virtual string GetName(void) const { return "TestProc"; }

virtual void ProcessEvent(void);

TestProc() {}

TestProc(shared\_ptr<CPSGS\_Request> request,

shared\_ptr<CPSGS\_Reply> reply,

TProcessorPriority priority);

virtual ~TestProc() {}

// This method will be called from the libuv event loop

void OnMyCustomEvent(void) {}

int c;

std::thread \* t;

};

Let’s make the processor handling the annotation requests:

IPSGS\_Processor\*

TestProc::CreateProcessor(shared\_ptr<CPSGS\_Request> request,

shared\_ptr<CPSGS\_Reply> reply,

TProcessorPriority priority) const

{

if (request->GetRequestType() != CPSGS\_Request::ePSGS\_AnnotationRequest)

return nullptr;

return new TestProc(request, reply, priority);

}

TestProc::TestProc(shared\_ptr<CPSGS\_Request> request,

shared\_ptr<CPSGS\_Reply> reply,

TProcessorPriority priority) :

c(0), t(nullptr)

{

IPSGS\_Processor::m\_Request = request;

IPSGS\_Processor::m\_Reply = reply;

IPSGS\_Processor::m\_Priority = priority;

}

The framework invokes the processor Process() method at the beginning of handling the request. This is done in a loop for all the processors which are able to handle the request. So it is important not to block inside the Process() method but wait the required events asynchronously. The test processor will wait for the event in a separate thread and the function f(…) will be the one executed by the thread.

void f(void \* proc)

{

// To illustrate the binding the thread will sleep.

// The real processor may wait for any kind of event,

// e.g. availability of the data on its backend

this\_thread::sleep\_for(chrono::milliseconds(1500));

// This is the moment when a non-libuv event has come.

// So libuv should be asked to do a callback from its event loop

// Note the last argument of the PostponeInvoke(): it is a pointer to

// instance of the TestProc

auto \* app = CPubseqGatewayApp::GetInstance();

app->GetUvLoopBinder().PostponeInvoke(libuv\_cb, proc);

}

void TestProc::Process(void)

{

t = new thread(f, (void \*)(this));

// Note: the Process() method does not block. It just returns

// while the events are waited in a separate thread.

}

One missed item in the fragment above is a libuv\_cb() function which libuv will call from its event loop. The implementation of the function can be quite simple:

void libuv\_cb(void \* user\_data)

{

// A drop of glue to deliver the call to the instance of the TestProc

// Now the TestProc::OnMyCustomEvent() is called

// from the libuv event loop

(TestProc \*)(user\_data)->OnMyCustomEvent();

}

The framework will regularly call the processor ProcessEvent() method (typically once per second). So the TestProc will use a counter and informs the framework about the completion when the counter reaches a certain value.

void TestProc::ProcessEvent(void)

{

++c;

if (c > 25)

SignalFinishProcessing();

}

Also the framework will ask a processor about its status using the GetStatus() method. The framework will not destroy the process while GetStatus() returns InProgress.

IPSGS\_Processor::EPSGS\_Status TestProc::GetStatus(void)

{

if (c < 25)

return IPSGS\_Processor::ePSGS\_InProgress;

t->join();

delete t;

return IPSGS\_Processor::ePSGS\_Found;

}

Please note that the framework may want to cancel the processor handling. To do so the framework will call the Cancel() method. So it is a responsibility of the processor to make sure that all the resources are properly cleared before the processor returns a non InProgress status in the GetStatus() method. In case of waiting events in a separate thread a processor must:

* shutdown the thread
* make sure that there are no postponed invokes which have not been delivered to the processor yet

before returning non InProgress status. Otherwise the framework may destroy the processor before already postponed callback triggered or before the waiting thread is shutdown correctly.

## Protocol Diagrams

The diagrams were generated using an online tool at <https://bottlecaps.de/rr/ui>

Here is the grammar text:

PSGProtocol ::= PSGchunk+ PSGFinalChunk

ChunkPrefix ::= '\n\nPSG-Reply-Chunk: '

BlobId ::= String

ProcessorId ::= URL-Encoded-String

PSGChunk ::= (BioseqInfoChunk | BioseqInfoFinalChunk |

BlobPropChunk | BlobPropFinalChunk |

BlobChunk | BlobFinalChunk |

BlobExcludeChunk | MessageChunk |

BioseqNAChunk | BioseqNAFinalChunk |

ProcessorMessageFinalChunk |

PublicCommentChunk | PublicCommentFinalChunk |

AccVerHistoryChunk | AccVerHistoryFinalChunk)

BioseqInfoChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=bioseq\_info' '&chunk\_type=data' '&size=' Integer '&fmt='

('json' | 'protobuf') '\n' Data '\n'

BioseqInfoFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=bioseq\_info' '&chunk\_type=meta'

'&n\_chunks=' Integer '\n'

BlobPropChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=blob\_prop' '&chunk\_type=data'

'&size=' Integer

('&blob\_id=' BlobId '&last\_modified=' Integer |

'&id2\_chunk=' Integer '&id2\_info=' String) '\n'

Data '\n'

BlobPropFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=blob\_prop' '&chunk\_type=meta'

'&n\_chunks=' Integer '\n'

BlobChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=blob'

'&chunk\_type=data' '&size=' Integer

('&blob\_id=' BlobId '&last\_modified=' Integer |

'&id2\_chunk=' Integer '&id2\_info=' String)

'&blob\_chunk=' Integer '\n' Data '\n'

BlobFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=blob'

'&chunk\_type=meta' '&n\_chunks='

Integer '\n'

BlobExcludeChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=blob'

'&chunk\_type=meta'

('&blob\_id=' BlobId '&last\_modified=' Integer |

'&id2\_chunk=' Integer '&id2\_info=' String)

'&n\_chunks='

Integer '&reason='

('excluded' | 'inprogress' | 'sent') '\n'

MessageChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type='

('blob' | 'bioseq\_info' | 'blob\_prop' | 'reply' |

'bioseq\_na' | 'processor')

'&chunk\_type=message' '&size=' Integer ('&blob\_id=' BlobId)?

'&status=' Integer '&code=' Integer '&severity='

('trace' | 'info' | 'warning' | 'error' | 'critical' |

'fatal') '\n' Message '\n'

BioseqNAChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=bioseq\_na'

'&chunk\_type=data' '&size=' Integer '&na=' String

'\n' Data '\n'

BioseqNAFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=bioseq\_na'

'&chunk\_type=meta' '&n\_chunks=' Integer '\n'

AccVerHistoryChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=acc\_ver\_history'

'&chunk\_type=data' '&size=' Integer

'\n' Data '\n'

AccVerHistoryFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=acc\_ver\_history'

'&chunk\_type=meta' '&n\_chunks=' Integer '\n'

PSGFinalChunk ::= ChunkPrefix 'item\_id=0' '&item\_type=reply'

'&chunk\_type=meta' '&n\_chunks=' Integer '\n'

ProcessorMessageFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=processor' '&chunk\_type=meta' '&n\_chunks=' Integer '\n'

PublicCommentChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=public\_comment'

('&blob\_id=' BlobId '&last\_modified=' Integer |

'&id2\_chunk=' Integer '&id2\_info=' String)

'&size=' Integer

'\n' Message '\n'

PublicCommentFinalChunk ::= ChunkPrefix 'item\_id=' Integer '&processor\_id=' ProcessorId

'&item\_type=public\_comment' '&chunk\_type=meta' '&n\_chunks=' Integer '\n'

GetBlobOKResponse ::= BlobPropChunk BlobPropFinalChunk

(PublicCommentChunk PublicCommentFinalChunk)?

(BlobChunk+ BlobFinalChunk)+ PSGFinalChunk

GetOKResponse ::= BioseqInfoChunk BioseqInfoFinalChunk BlobPropChunk

BlobPropFinalChunk

(PublicCommentChunk PublicCommentFinalChunk)?

(BlobChunk+ BlobFinalChunk)+

PSGFinalChunk

ResolveOKResponse ::= BioseqInfoChunk BioseqInfoFinalChunk PSGFinalChunk

GetNAOKResponse ::= BioseqInfoChunk BioseqInfoFinalChunk (BioseqNAChunk BioseqNAFinalChunk)\* PSGFinalChunk

GetTSEChunkOKResponse ::= BlobPropChunk BlobPropFinalChunk

(PublicCommentChunk PublicCommentFinalChunk)?

BlobChunk BlobFinalChunk PSGFinalChunk

AccVerHistoryOKResponse ::= BioseqInfoChunk BioseqInfoFinalChunk (AccVerHistoryChunk AccVerHistoryFinalChunk)\* PSGFinalChunk

UnknownURLResponse ::= MessageChunk PSGFinalChunk

## GetBlob Diagram



## General Server Structure

There are two major transport layers in the PSG Server:

* TCP layer. It is comprised of:
  + TCP daemon (CTcpDaemon template class)
  + Worker threads (CTcpWorker template class)
  + Worker thread container (CTcpWorkerList template class)
* HTTP layer. It is comprised of:
  + HTTP connection (CHttpConnection template class)
  + HTTP protocol handler (CHttpProto template class)
  + HTTP request (CHttpRequest template class)
  + HTTP reply (CHttpReply template class)
  + HTTP daemon (CHttpDaemon template class)
  + various wrappers

Each layer is in its own namespace. All classes related to HTTP layer are in HST namespace, while TSL namespace is intended for components running on TCP layer.

### Startup

The working loop starts in CTcpDaemon::Run() method. This method performs the following:

* creates CTcpWorkersList container class
* installs signal hooks for SIG\_INT, SIG\_TERM, SIG\_USR1, SIG\_USR2 and SIG\_WINCH
* opens listening socket
* distributes listening socket among worker threads using uv\_import\_start() / uv\_import() helper functions
* enters main loop uv\_run()

Container creates threads and maintains them during their lifetime. Worker threads are represented by CTcpWorker context struct.

### CTcpWorker

This template structure represents worker thread that can work with generic P (protocol class), U (connection abstraction class) and D (outer daemon abstraction class, singleton), so the core can work with any classes implemented a certain API. Worker thread runs its main loop uv\_run() invoked in CTcpWorker::Execute() method.

### CTcpWorkersList

This template class is a container that creates, holds and "joins" worker threads.

### CTcpDaemon

This template class counts the number of connections, the number of requests. When a request comes it finds a worker associated with the current thread. Its main method Run() enrolls all the workers, then enters the loop and waits there until SIG\_INT or SIG\_TERM is delivered. If it was SIG\_INT then the server exits immediately. If it was SIG\_TERM then the server performs a gracefull shutdown letting all the currently executed requests finish.

### CHttpRequest

This class holds all parameters associated with a particular HTTP request. It runs GET or POST parser to parse the incoming buffer into a list of parameters.

### CPSGS\_Request

Higher level wrapper of a request. It holds all the request parameters parsed and sanitized. The processors deal with an instance of this class.

### CHttpReply

This template class provides end-point code with an interface to send a reply – the response status and the data. CHttpReply instance is created or picked from a buffer for each incoming HTTP request. This class maintains the state, the data availability, the IO availability etc. It knows what CHttpProto and CHttpConnection the request is associated with. It also holds a reference to the associated user data of the generic class P.

### CPSGS\_Reply

This is a high level wrapper around CHttpReply. The class takes care of the chunks of the PSG protocol and provides facilities to form PSG protocol chunks. The processors should deal with an instance of this class and should not use the lower level class directly.

### CHttpConnection

This template class holds 3 lists of requests CHttpReply – cache (m\_Finished), pending (m\_Pending) and active (m\_Backlog). Active requests are the requests that end-point is being invoked or will be invoked immediately or was invoked previously but corresponding work is not yet complete. Pending list contains postponed requests – their end-point will be called later. List of CHttpConnection connections is maintained by CTcpWorker.

### CHttpProto

This template class is an API provider for various events such as ThreadStart and ThreadStop invoked by a worker thread when it's about to start and stop, OnNewConnection and OnClosedConnection are invoked when a new client is connected or an existing disconnected, OnTimer is invoked by 1sec timer, OnAsyncWork is invoked by waken a worker to allow to finish active requests in a worker thread context, DaemonStarted and DaemonStopped are invoked by daemon when it's about to start and stop, OnHttpRequest is invoked by an HTTP library when it extracts a new HTTP request. There is only one incstance of CHttpProto per worker.

### CHttpDaemon

This template class represents an application-level daemon. It knows of all registered end-points (m\_Handlers), it initializes and finalizes underlying HTTP library, it creates, holds and invokes instance of CTcpDaemon

### CPendingOperation

The class wraps one processor which is capable to handle a request. It is responsible for calling the processor ProcessEvent() when a libuv event has come.

Basically when a request comes a dispatcher tries to create all the registered processors. Then for each successfully created processor an instance of the CPendingOperation is created. The list of pending operations is then associated with the incoming HTTP connection.

### CPSGS\_Dispatcher

For each incoming request the dispatcher tries to create all the registered processors. Then the class keeps an association between the request and the processors which are handling it. The class also lets the processors to communicate, e.g. a processor may inform all peers that it started to receive data from the backend or to inform the dispatcher that it finished processing.

### New Connection Flow



### Request Flow



### Handling Request In Synchronous Manner



### Handling Request In Asynchronous Manner

