QGIS Server and startup time in a cloud environment

www.QCooperative.net

Alessandro Pasotti and Paul Blottiere

21-12-2020

Table of Contents

Introduction

QGIS Server is being used increasingly in production with numerous online instances. Such infrastructures, typically deployed in cloud environment with a huge number of clients, generally require a very high level of availability. In this respect, the startup time of the individual instances becomes a determining factor.

A large infrastructure is usually synonymous with large and complex QGIS projects: high number of layers, complex symbology, … and the startup time of QGIS Server may become an issue in this kind of situation.

The classic solution to this kind of problems is a shared cache. This way, every QGIS Server instance could possibly retrieve information directly from the cache instead of reloading the whole execution environment from scratch.

So the scope of this research work is to identify what information could be cached (typically the most time-consuming steps in the server startup process) as well as discussing the shared cache solution and proposing technical implementations in a multi-machines environment.

Methodology

Considering that we would want to cache the most time-consuming information to retrieve, we have to profile the startup process of QGIS Server to determine the bottlenecks. But first, the *startup phase* needs to be defined.

The overall objective is to reduce the time to be up and running for QGIS Server, meaning the period between the launch of the FastCGI process and the ability to efficiently respond to a client with actual information. In practice, the startup phase involves three stages:

1. Stage 1: startup of the runtime environment (loading of libraries, plugins, …)

2. Stage 2: reading a project
3. Stage 3: generating the response (for example the *GetCapabilities* document)

To profile these stages, we are going to use several ways from the very general to the very specific against a very large project (about 800 PostGIS layers) and based on the *master* branch of QGIS:

1. The *time* command line tool
2. QGIS profiler (specifically implemented for this study #40399)
3. Hotspot and flamegraph analysis

Then, after analysing the results, we'll report upstream improvements in QGIS source code (Pull Requests already merged or in progress). Indeed, some optimizations were already possible to reduce the startup time without further investigations. We'll also quickly compare these results with a large project based on Shapefile and Geopackage layers as well as with the QGIS 2.18 release.

Finally, we'll have the information necessary to address the shared cache mechanism topic.

Profiling

On large project we have been testing, starting the FastCGI process (stage 1) takes about *600 ms* while the stage 2 and 3 take about *26 seconds*. Thanks to the Hotspot tool, we can have a clearer idea of the situation for the two last steps:

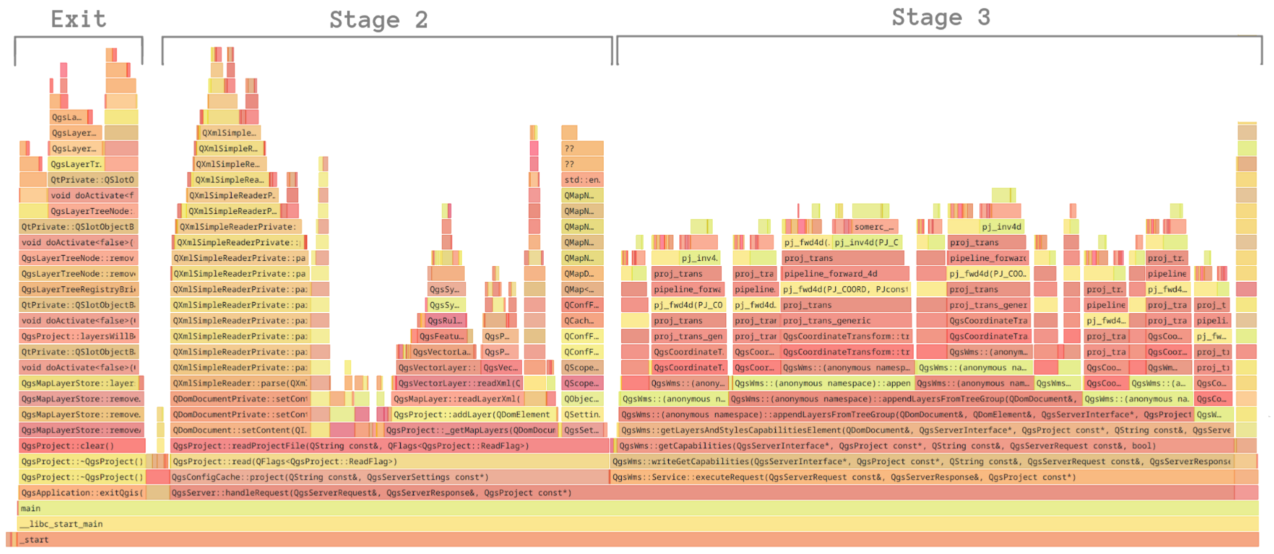


Figure 1: Flamegraph

Considering that we're not interested by the *Exit* stage, the situation can be summarised with the chart below. From now on, we are going to focus specifically on every stages.

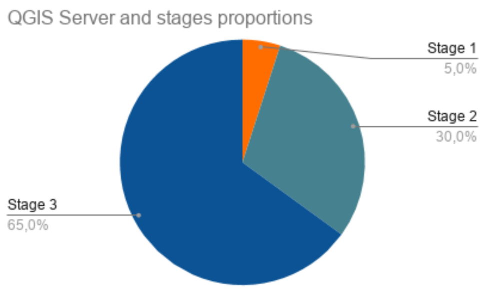


Figure 2: Stages

Stage 1: Startup of the process

The startup stage of QGIS Server consists in:

1. The OS loading the process into memory, loading and resolving shared libraries
2. QGIS Server searching and loading its library modules:
    - Data providers
    - Authentication methods
    - Server modules (WMS, WFS, WCS, WFS3, LandingPage etc.)

Considering that the whole stage 1 represents only about 5% of the total time, the potential gain is small. Moreover, there is not much room for substantial speed improvements in this area but a few tiny speed improvements are still possible.

Below an interesting result showing the proportion of time spent in loading modules.

Scenario	Time in ms	Penalty
Default behavior	603	12%
No modules	534	-

This shows how the loading of QGIS library modules takes a small but significant time of stage 1.

DLL loading The first optimization would be to move all types of DLLs to dedicated directories to avoid scanning the directory and trying to load an incompatible DLL (i.e. an authentication method while loading data providers and vice-versa).

Data provider loading The second optimization is about swapping *multipleProviderMetadataFactory* and *providerMetadataFactory* in the provider registry. Indeed the latter is used by all providers but a single one (WFS).

A dedicated PR #40317 has already been merged. The gain in speed is a few milliseconds.

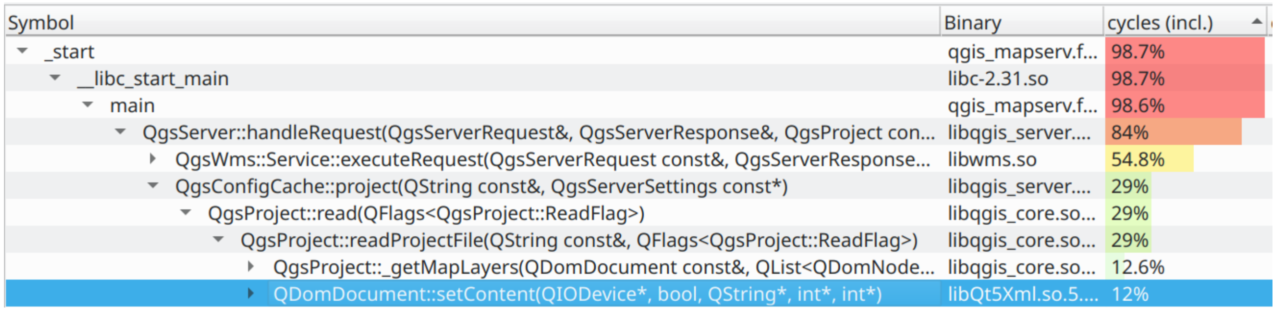
Authentication methods loading A third optimization would be to add a specific option to skip the loading of authentication methods altogether in case the projects do not need any particular authentication method. This could be implemented with a server configuration setting.

Finally another optimization, already implemented with #40315 is to avoid the load of authentication methods as providers. The gain in speed is a few milliseconds.

Stage 2: Time to load a project

The time to load a project consists in:

1. Parsing the XML document
2. Creating QGIS internal objects (map layers, project, . . .)



Symbol	Binary	cycles (incl.)
_start	qgis_mapserv.f...	98.7%
__libc_start_main	libc-2.31.so	98.7%
main	qgis_mapserv.f...	98.6%
QgsServer::handleRequest(QgsServerRequest&, QgsServerResponse&, QgsProject con...	libqgis_server....	84%
QgsWms::Service::executeRequest(QgsServerRequest const&, QgsServerResponse...	libwms.so	54.8%
QgsConfigCache::project(QString const&, QgsServerSettings const*)	libqgis_server....	29%
QgsProject::read(QFlags<QgsProject::ReadFlag>)	libqgis_core.so...	29%
QgsProject::readProjectFile(QString const&, QFlags<QgsProject::ReadFlag>)	libqgis_core.so...	29%
QgsProject::_getMapLayers(QDomDocument const&, QList<QDomNode...	libqgis_core.so...	12.6%
QDomDocument::setContent(QIODevice*, bool, QString*, int*, int*)	libQt5Xml.so.5....	12%

Figure 3: Time to load a project

Considering the above results from Hotspot, we can say that the XML parsing (*QDomDocument::setContent()*) and the creation of map layers (*QgsProject::_getMapLayers()*) are approximately even with ~12% of the total time.

In the immediate outlook, nothing can be done on the XML parsing mechanism to improve the situation because it's an internal method of the Qt library. However we'll discuss later in this document potential solutions.

On the contrary, the creation of QGIS internal objects is subject to prompt improvements as described below.

Storage format: qgz vs qgs The qgz file format is a zip file containing a QGS project as well as a QGD database if the auxiliary storage mechanism is used on one (or several) of the vector layers. While using a qgz project in desktop mode has basically no effect for end-users, the unzip action necessary to extract the project file implies an extra cost. This extra cost may be an issue on server side so we took a quick look on the *QgsArchive::unzip()* method. The time to unzip a 26M XML file from a qgz archive takes about 45ms. The total time to extract and parse/read the project is about 3300ms in this case, meaning ~1%.

So if someone really wants to improve the loading time in the order of magnitude of the milliseconds, using a qgs project is the good way to go.

Avoid unnecessary detach An under optimized loop, spotted thanks to Hotspot, was slowing down the project loading. By fixing that part of the code with #40420, we gained a small speed improvement (~150ms).

After the a.m. PR a few other similar optimizations have been applied to other parts of the project loading process with #40542, leading to an overall speed improvement of ~450ms.

Skip desktop specific initialization code There are many areas of QGIS core library that were developed with focus on the Desktop application and that are not optimized at all for non-interactive use (QGIS Server is just one of these use cases). As far as the project loading is concerned, a common approach is to pass a flag to the project in order to skip certain parts of the code that are not relevant for the server.

This approach was applied to the original style properties stored in bad layers handling and to restore the symbology and other layer properties when a broken layer is saved back into the project.

This optimization led to a huge speed improvement (the overall time for the test project cold start to *GetCapabilities* went down to 15 seconds from 26 seconds).

By implementing a read flag for the projects that allows the server to skip the initial storage of layer styles (see #40360), we measured the impact of the cold startup times with a single project having about 800 layers.

Scenario	Time in s	Penalty
Default behavior	26	73%
Skip storage	15	-

A similar approach was taken for the snapping configuration step with #40418 (snapping is not used in non-interactive QGIS applications) and with the optimization of this process we have been able to spare a small amount of time (~140ms) during the project startup

Avoid multiple QgsSettings instances Another small improvement (~150ms) was implemented in #40498 by storing a *QgsSettings* instance in the project class instead of using several throw-away instances.

Qt DOM manipulation The Qt DOM API may be pernicious and lead to bad performances according to the methods being used. For example the *QDomNodeList::count()* and *QDomNodeList::size()* methods are very time consuming and should be avoided. Such a work has been done in #40542 and #40500 and saved ~500ms.

Stage 3: GetCapabilities document

Thanks to the flamegraph, it can be observed that during the *GetCapabilities* document generation, a lot of time is spent in PROJ to calculate extents, transforming bounding boxes, . . .

WGS84 extent When building the *GetCapabilities* document, the WGS84 extent is necessary for every layer. However, the computation may be very time consuming, especially due to the *transformBoundingBox* call. To improve performances, the WGS84 extent may be stored in the qgs project and used as soon as the *trust* option is activated. The gain speed is about 6% for ~800 layers. Concretely, we're saving a few milliseconds per layer so it's negligible for projects with a small number of layers.

The implementation is available in #40444.

Data provider influence and comparison with QGIS 2.18

The previous profiling analysis for the QGIS *master* branch is based on a large project with PostGIS layers. But extracting information from the datasource in order to construct internal QGIS objects (layers, . . .) take some time and may differ from one data provider to another. Moreover, the QGIS Server implementation has changed significantly since the QGIS 2.18 release. Considering these aspects, some comparisons tests have been performed. The results were as follow:

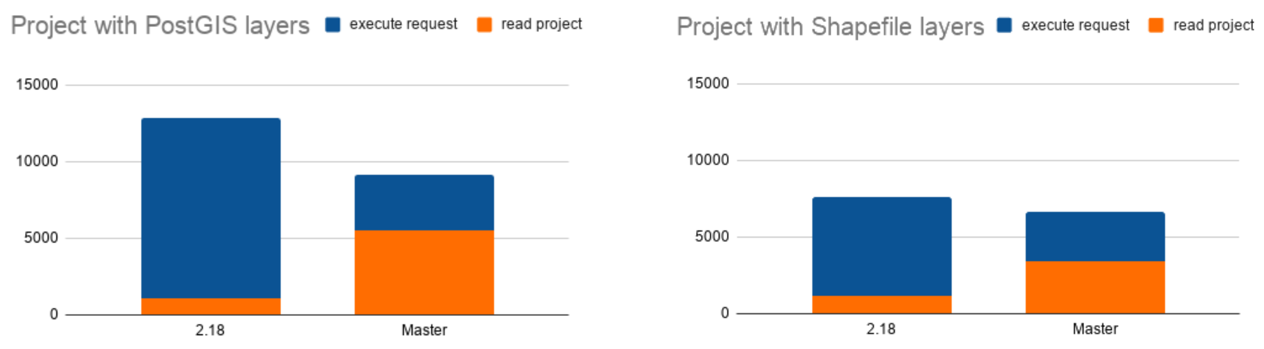


Figure 4: QGIS 2.18 vs master branch

The main conclusion to be drawn from this experience is that the 2.18 release is more efficient to read the project but slower to build the *GetCapabilities* document than the *master* branch (once optimizations PR merged). This may be explained by the fact that QGIS Server 3 uses the generic *QgsProject* class whereas the 2.18 release uses a custom *QgsServerProjectParser*. The latter is very minimal and loads only what is necessary for QGIS Server. However, such a mechanism raises some fundamental questions about long term maintenance. So, it's probably important to stress that there is no perfect solution in this case.

The second conclusion is that a project based on Shapefiles is more quickly loaded than a project based on PostGIS layers, regardless of the version of QGIS Server.

Shared cache

The profiling analysis realized thanks to the Hotspot tool and its flamegraph gave us precise information about the timing of a cold server startup and led us to a consistent number of optimizations in QGIS core and server source code. Due to the very high number of layers in the project, we have been able to improve the global reading time of the project as well as the *GetCapabilities* document generation by saving a few milliseconds per layer. With that said, we don't see a lot of remaining improvements for simple optimizations.

In case of the *GetCapabilities* document, some information are cached directly in the project (layer extent and wgs84 extent) and read when needed if the *trust* option is activated. However, it does not appear opportune to keep more information in cache, and even if it was necessary, the project should/could probably be used. So a broader cache mechanism wouldn't be useful to optimize the stage 3, especially given that a *GetCapabilities* document can already be cached as a whole.

In the same way, a cache mechanism doesn't seem relevant for the stage 1. Thus, only the reading project phase could benefit from a shared cache. Indeed, the XML project parsing into QT DOM objects and the transformation of the QT DOM objects into QGIS internal structures (map layers etc.) is one of the most time consuming steps and lead to several questions:

1. What can be done to improve the loading time of a qgs project?
2. How could we efficiently cache a project?

A binary project storage

Possible solutions To improve the parsing time of a qgs project, there are mainly two possible approaches:

1. Using a faster library to parse the XML project documents
2. Using a faster format for storing and retrieving the projects

Regarding the first approach, we may note that the XML parser of Qt is famous for being very slow so we tried another XML library - named RapidXML - to read and parse the qgs document. The Qt parsing takes about 950ms whereas the RapidXML takes less than 200ms. In this case, the time spent for the whole operation is reduced by a factor of 5.

Indeed, using another library to parse and read the XML document may lead to better results, but a binary format instead of XML to store a project will necessarily lead to even better performances. This would reduce to a minimum the two steps implied in the project loading phase by drastically reducing the parsing phase time and leaving only the QGIS internal data structures creation phase.

Implementation The low-level implementation of a binary storage format could be effectively done using *QDataStream*: this QT class allows to store the primitive QT types into a *QIODevice* (which include files and *QTcpSocket*) and it is extendable to custom classes by the implementation of the insertion/extraction operators.

At a higher level, the cleanest and most powerful solution is to use a visitor pattern, not only because of the multiple use cases but also because it leaves the doors open for additional storage formats that we might want to add in the future.

By implementing a generic visitor pattern for QGIS project components we could use the visitors not only for marshalling/unmarshalling but also for doing other operations on the project components hierarchy, for example to collect information about the different QGIS project components (such as collect all expressions from styles and widgets, collect all SVGs etc.).

The proposed implementation implies the creation of an abstract interface for project components (all QGIS classes that currently have a *writeXML/readXML* member), the interface would expose a pure virtual accept method that takes the base visitor class (or a lambda overload) as an argument.

The concrete visitor implementations will provide the functionality to marshall/unmarshall the different components into/from the different supported storage formats (XML and binary as a start) and to extract/modify other pieces of information (for example retrieve all the expressions or the SVGs from the different components).

Of course this API would not be QGIS Server specific, any QGIS application would directly benefit from it.

A shared cache for projects

Current implementation of QgsConfigCache The current implementation of QGIS Server keeps an internal cache of QGIS projects in the *QgsConfigCache* class. This cache keeps in RAM an instance of the QGIS project allowing for an immediate retrieval of the project when a request for that project hits the server.

QgsConfigCache is a simple cache and it works in the most traditional way: it maintains a hash of QGIS projects (*QgsProject* objects) and whenever the server needs a particular project it checks if the project is already present in the cache and reads the project from the disk if it is not, finally the cache returns the project retrieved from the cache.

QgsConfigCache is the ideal place to hook up a secondary shared cache from which the different QGIS Server instances can retrieve the cached projects instead of loading them directly from the disk in case of cache miss. Note that secondary cache would not replace the existing internal cache in any way, because querying and retrieving project information from the secondary cache would imply the re-creation of the project on each and every request, leading to an unacceptable slowdown.

Constraints Considering that this shared cache could be used across hosts and virtual machines to centralize the project storage, a mechanism based on a *QSharedMemory* is not an option (a common RAM segment is necessary).

Moreover, the shared cache could be designed in a few different ways, to achieve different degrees of control. For example:

1. Decentralized solution: every first time a project is required from any of the QGIS Server instances the project is loaded from the disk, the project is then sent to the cache from the QGIS Server instance that loaded it, all subsequent project requests from other QGIS Server instances will retrieve the project from the shared cache.

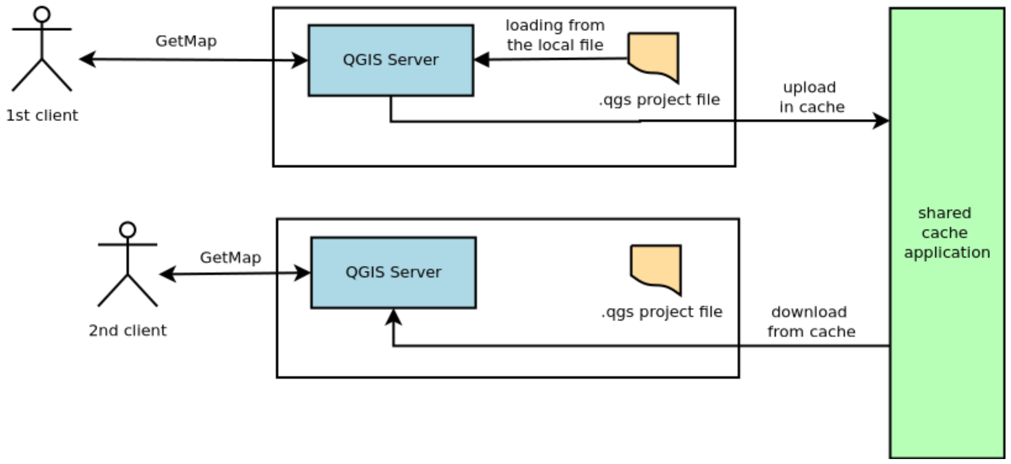


Figure 5: Decentralized solution

2. Centralized solution: projects may be stored on a single location/machine. In this case, the shared cache may be preloaded with all projects that will be served by the QGIS Server instances

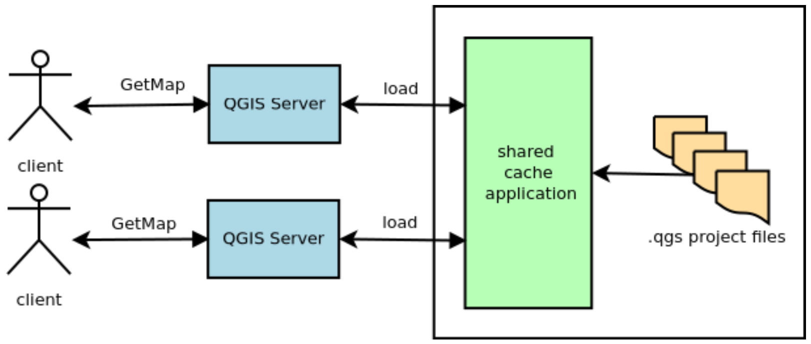


Figure 6: Centralized solution

Communication This design implements a unidirectional communication (from QGIS Server instances to the shared cache) and it would not be possible for the shared cache to notify the QGIS Server instances about cache invalidation of a particular project.

A bi-directional communications channel where the shared cache would be able to send messages directly to the QGIS Server instances presents some technical challenges if implemented as a QGIS Server API as discussed in detail in the recent QEP about the server monitoring (see the discussion about QGIS Server behind reverse proxies and load balancers).

Also from the a.m. QEP, one of the most promising implementations to overcome that limitation is to use a poll mechanism where the QGIS Server instances periodically access a QGIS Server controller application and gather information related to the cache validity. The polling logic must reside in its own thread in order to not slow down the ordinary QGIS Server operations (i.e. serving requests). Following this approach, the QGIS Server instances query the QGIS Server controller application for an expiration tag or a last-modified timestamp related to a cached project and invalidate or refresh the cached object accordingly.

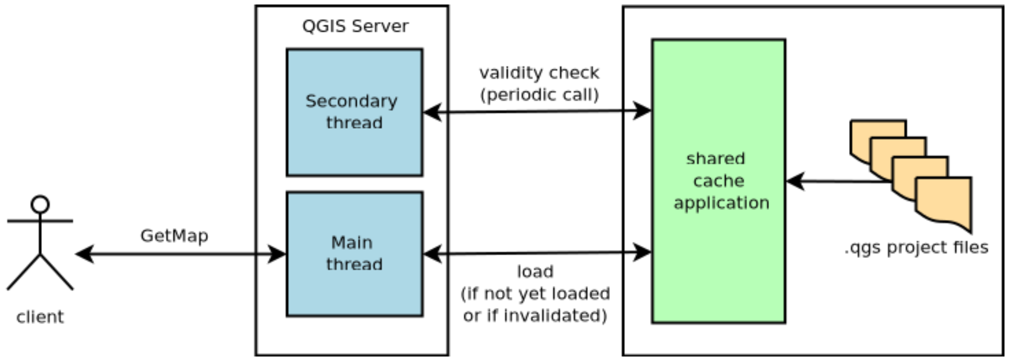


Figure 7: Cache invalidation

Implementation discussion Without entering into details of the possible implementations of the shared cache application, the choice largely depends on whether it needs to offer the logic to pre-load the QGIS projects from a list of file system locations (or QGIS project stores like PG or GPKG) and the automatic cache invalidation or if this functionality is delegated to a different independent application.

It is possible to think at a generic cache (memcached or redis for example) that has no knowledge about QGIS internals and delegate the cache control functionality to the dedicated QGIS Server controller application that would take care of cache preloading and cache invalidation, the latter will require the usage of the QGIS core libraries to load the project into the generic cache and to manage their lifetime.

To summarize this up: the use of a QGIS Server controller application appears to be the most promising solution, this application could integrate the shared cache or rely on a generic shared cache application (memcached or redis for example), in this case the QGIS Server instances would query the QGIS Server controller application to know the type of cache they need to use and the connections parameters. In case we wanted to support more than a single cache engine, it would be necessary to abstract the shared cache interface in order to implement different cache connectors (memcached or redis for example). Note that a generic cache connector for QGIS server could also be used for the other cache which is currently implemented in QGIS server: the get capabilities response cache and possibly also for the caching plugin filters (available in the Python server interface).

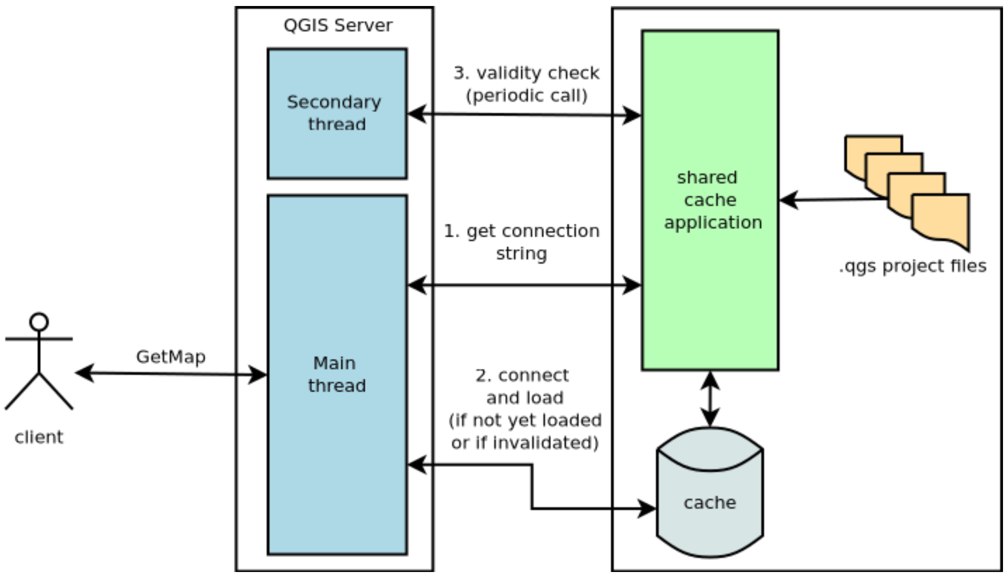


Figure 8: Cache connection

Conclusion

Thanks to this study, we've reached the conclusion that improving the *QgsProject* management is crucial if we want to improve the startup time of QGIS Server. This may de bone by two complementary solutions:

1. implementing a more efficient format to store project
2. developing a server cache for projects

A QEP will be drafted soon to discuss these possibilities with the community.

About performances, we may also note that before this study, the 2.18 release was faster than QGIS 3 (see the 3.10 release below) to build the *GetCapabilities* document based on the large project with PostGIS layers. But thanks to the bunch of optimizations made in *master* and backported to the 3.16 release (for some of them at least), QGIS 3 is now faster than the 2.18 release.

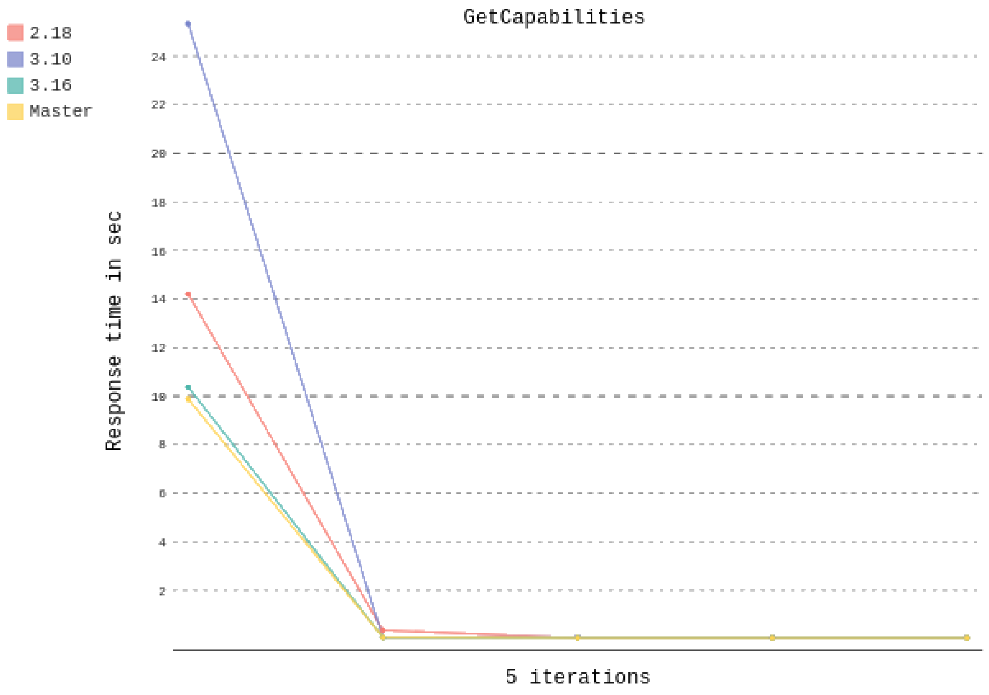


Figure 9: Project with PostGIS layers