Improving the Latency of Python-based Web Applications

António Esteves[a] and João Fernandes
Centro ALGORITMI, School of Engineering, University of Minho, Campus de Gualtar, Braga, Portugal
esteves@di.uminho.pt, jmiguelgfernandes@gmail.com

Keywords: Web Performance Optimization, Latency, Web Application, Django, Python.

Abstract: This paper describes the process of optimizing the latency of Python-based Web applications. The case study used to validate the optimizations is an article sharing system, which was developed in `Django`. `Memcached`, `Celery` and `Varnish` enabled the implementation of additional performance optimizations. The latency of operations was measured, before and after the application of the optimization techniques. The optimization of the application was performed at various levels, including the transfer of content across the network and the back-end services. HTTP caching, data compression and minification techniques, as well as static content replication using Content Delivery Networks, were used. Partial update of the application's pages on the front-end and asynchronous processing techniques were applied. The database utilization was optimized by creating indexes and by taking advantage of a NoSQL solution. Memory caching strategies, with distinct granularities, were implemented to store templates and application objects. Furthermore, asynchronous task queues were used to perform some costly operations. All of the aforementioned techniques favorably contributed to the Web application's latency decrease. Since `Django` operates on the back-end, and optimizations must be implemented at various levels, it was necessary to use other tools.

1 INTRODUCTION

According to published studies, regarding the effect a Web application's response time has on the user's experience, 47% of the users expect the pages to be loaded in two seconds or less and 40% will abandon the site if it takes more than three seconds to load (Anastasios, 2016). With the increasing complexity of Web applications, optimizing its response time (or latency) is crucial in many cases.

The concept of performance is wide. According to (Meier et al., 2004), performance can be expressed in terms of latency, throughput, scalability, and resource utilization. **Latency** quantifies the time required to perform a particular operation. In Web applications, the latency associated with a page request includes both server latency, which is the time required to process the client's request, and the latency of the network, which is the time spent in transferring the request and response messages of the page. The **throughput** describes the number of transactions performed per unit time. Due to multithreading strategies, this metric is not the inverse of latency. **Resource utilization** refers to the consumption of services, or infrastructures, such as memory, processor, disk, and network. **Scalability** is the ability of a system to process higher loads, allowing an increase in the number of users or operations without significantly degrading performance.

In this paper, we discuss several latency optimization techniques that were applied on a Web application development. Optimization of the application performance took place at several levels, including the transfer of content across the network and the back-end services. The optimization of the content transfer involved the use of techniques aimed at reducing the number or size of HTTP messages, such as HTTP caching (Grigorik, 2017), information compression and minification (Google, 2017), as well as static content replication using Content Delivery Networks (CDNs) (Fielding et al., 1999). In addition, asynchronous processing techniques and partial updating of application pages on the front-end were applied. On the server side, the database was optimized by creating indexes (Connolly and Begg, 2005) and taking advantage of a NoSQL solution (Smith, 2013). Authors in (Holovaty and Kaplan-Moss, 2009) recommend the use of in-memory caching strategies, which have been implemented with distinct granularities to store templates and objects generated by the application. In addition,

[a] https://orcid.org/0000-0003-3694-820X

we used asynchronous task queues to improve the user experience in some scenarios (Greenfeld and Greenfeld, 2017) (Arcos, 2016).

The paper is organized as follows. Section 2 presents the related work and section 3 refers the technologies used to the develop the Web application that will be presented later on. In section 4 is introduced our case study: the `Sharticle` Web application. Section 5 details the performance optimization techniques, which allowed decreasing the latency of several `Sharticle` operations. Section 6 presents the results obtained with the optimizations made on `Sharticle`. Finally, in section 7 are pointed out some conclusions and topics for future developments.

2 RELATED WORK

According to (Subrayen et al., 2013) the major goal of improving Web pages' performance is to minimize the delay, perceived by the user, between the moment he/she clicks on a link and the page is displayed. This is a user-centric approach to Web performance optimization (WPO), the same followed by our work. To reduce the perceived delay, they reduced the time the browser takes to fetch a given resource, decreased the number of requests, optimized the rendering speed, and made the loading time appear shorter. The paper presents the optimization techniques but not the gain obtained with their application.

In (Manchanda, 2013) the author analyzed seven techniques to reduce the average user response time for the Moodle Learning Management System. A performance analysis of Moodle was performed using Apache JMeter. Among the front-end optimizations documented are reducing the number and size of HTTP requests, minimizing DNS lookups (Grigorik, 2013), using HTTP compression, reducing the response header size by deactivating the ETags, and reducing the response time with AJAX. The maximum reduction in user response time was 98%, achieved with hardware optimizations.

Authors of (Horat and Arencibia, 2009) present Web optimizations such as reducing the number of HTTP requests, using CDNs, using expires header, compressing plain text resources, deactivating ETags, avoiding HTTP redirections, sending partially generated pages, reducing the number of DOM elements and `iframes`, reducing the cookies size, using domains without cookies for static components, placing styles and scripts correctly, avoiding CSS expressions, using GET requests whenever possible, externalizing all static content, minifying styles and scripts, downloading components before they are requested, and selecting the most compact image formats. The work doesn't report the application of the techniques, just analyzes Moodle to verify if it is "compliant" with them.

In (Ossa et al., 2012), latency is considered the main Web performance metric. This work is grounded on studies that demonstrate that Web prefetching is an effective technique to reduce latency. Web prefetching was split in two main components: a predictor engine and a prefetching engine. The predictor is located at the Web server and the prefetcher is at the client. Prefetching improved page latency from 39% to 52%.

In (Shivakumar and Suresh, 2017) it is discussed the key aspects of pro-active quality in Web applications, including identification of the key indicators of quality problems, methods to achieve the quality at the source, tools for continuous integration, and monitoring of QoS. WPO techniques were classified in 8 categories. The application of the techniques was not validated with quantitative measures.

(Anastasios, 2016) evaluates how WebPageTest, GTmetrix, PageSpeed Insights, Blackfire, and Google Analytics can be used to identify which parts of a Web application must be improved to accelerate page rendering. The author also analyzed a significant set of front-end and back-end optimization techniques. Despite the detailed and fairly comprehensive explanation of the different optimizations, the author presents few results.

Since front-end tasks contribute on average for 80-90 % of the time it takes to get a response from a Website, work in (Cao et al., 2017) focuses on five front-end optimizations. Authors present the optimizations but don't validate their implementation.

In (Cerny and Donahoo, 2010a; Cerny and Donahoo, 2010b) it is used a debugging Web proxy to evaluate and improve page load times, identifying the load time and characteristics of the various page resources. After identifying the resources with the largest impact on page load time, several optimizations can be applied. The number and coverage of the applied optimization strategies are very incomplete.

3 TECHNOLOGIES

This section summarizes the main technologies that supported the Web application development and optimization. **Django** is a full-stack Python-based Web development framework that provides solutions

for all levels of the application stack. `Django` uses the MTV architectural pattern, which comprises models, templates, and views. The **models** form the data access layer, containing its representation and validation rules. The **views** make up the business logic layer, being responsible for defining which data will be presented to the users, rely on templates for rendering the content. **Templates** establish the presentation layer, since they define the visual structure of the content, receiving the data from the views and rendering it according to established rules.

In contrast to traditional Web servers, which use several threads to serve customer requests, **NGINX Web server** has an event-driven architecture that is more scalable than alternative solutions, enabling the simultaneous processing of thousands of requests. This Web server also has a set of characteristics that allow to improve the performance of applications, namely load balancing, fault tolerance, caching and resources compression.

NoSQL solutions are flexible since they allow variable-structure data. The best cache solutions utilize key-value NoSQL databases to store the data in memory, as in the `Memcached` and `Redis` solutions. The **Memcached** technology was used in order to implement caching strategies at various levels on the server side. `Redis` is a more flexible alternative that can also be used as broker in the implementation of asynchronous task queues.

Varnish is a reverse proxy, placed between the Web server and the network, that caches the responses returned by the application. In most cases, reverse proxies are used to decrease the latency associated with the retrieval of Web content. If `Varnish` has an appropriate response in its cache, it returns it directly to the client. Otherwise, it contacts the Web server in order to obtain a response to the client. `Varnish` is an alternative to using `Django` and `Memcached` for caching templates on the server side.

The **Celery** asynchronous task queue technology was selected in the present work. Other alternatives were Huey and `Django Q`. Commonly, an asynchronous task queue requires a broker to store the tasks. Brokers may be implemented with in-memory key-value databases. A broker can be seen as a simple message queue without consumers. Examples of available brokers are `Redis`, `Disque`, and `RabbitMQ`. `Celery` supports task scheduling, the possibility of working synchronously, flexibility, and can be integrated with brokers.

4 SHARTICLE APPLICATION

The Web application developed to validate the optimization techniques is an article sharing system, similar to `Medium`. Users can register themselves in the application, and gain access to wider functionalities. `Sharticle` includes features such as writing, publishing, consulting, and commenting articles, as well as searching articles by topic or title. An article consists of a title, a topic, some tags, and content that may contain text and images. The application was developed with `Django` and the other technologies introduced in the previous section. Figure 1 presents the architecture of `Sharticle`. On the server side, the architecture uses in-memory caching technologies (`Memcached`), asynchronous task queues (`Celery`), and a reverse proxy (`Varnish`). At the network level, a CDN is used to serve static content.

5 IMPROVING SHARTICLE LATENCY

In order to reduce the latency of the operations, several optimization techniques were applied, including database indexes, a NoSQL solution, in-memory caching, asynchronous task queues, multi-level paging, HTTP compression and caching, minification, combination and correct placement of scripts and styles, dynamic inlining, post-onload download, data validation and rendering on the front-end, and CDNs (Souders, 2007).

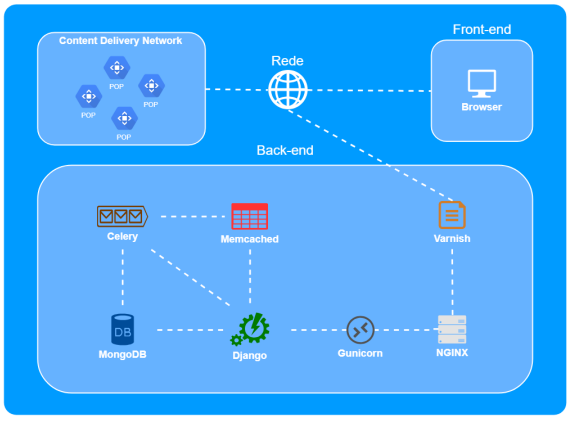

Figure 1: Architecture of `Sharticle` Web application.

5.1 Cached Sessions

To speedup reading and writing the users session data, it was decided to keep such data in memory, therefore

eliminating unnecessary accesses to the database and increasing the efficiency of login and logout operations. To achieve this goal, `Memcached` was used and it was necessary to appropriately configure the `Django` project so that the application could interact with `Memcached`.

5.2 HTTP Caching

Two HTTP caching strategies were used, namely caching with explicit time and conditional caching with `Last-Modified` headers. Conditional caching allows us to significantly reduce the size of a response if the content that is stored in an HTTP cache is still valid. In `Sharticle`, the pages used to view and edit the user profile are dynamic. Therefore, these pages are served along with `Last-Modified` headers, to enable the clients reuse their copies. Similarly, pages containing the lists of articles belonging to a particular user are also subject to conditional caching, varying according to the actions of creating, deleting and publishing articles. Additionally, pages used to edit the drafts of articles, which vary with the update frequency, also have `Last-Modified` headers. This strategy only proves to be favorable if calculating the last modification date of the resource is faster than generating a response that contains the resource itself.

Regarding HTTP caching with explicit time, it is important to select a suitable time for the validity of the resources. Although this technique is better suited for static resources, it can be used in dynamic pages. The published articles view pages are dynamic because they contain, in addition to the content of the corresponding article, the information of the respective author. However, one can perform HTTP caching of these pages for some time, eventually reaching consistency after expiring the resource copies. In addition, such pages also include user session data in the navigation bar. In order to reuse the copies of these pages among clients, and adapt them to each user session, it was necessary to render the session data in the application front-end. For this purpose, cookies that include such session data were used by a script that populates the navigation bar after loading the page from the server or a cache. Similarly, HTTP caching with explicit validity was also applied to the pages used to search articles by topic, which are also rendered progressively with cookies.

5.3 Database Indexes

In order to speedup database selection operations indexes should be used. Selecting user information, based on its user name, integrates the process of constructing several pages of the application. This way an index was created on the attribute that corresponds to the user name, which contributed for decreasing the latency of such operation. Similarly, indexes were also created on the attributes that correspond to the identifier and author of an article. These attributes are used by several operations to select articles. To accelerate searching articles by keyword, a textual index was created on the attributes that correspond to the title and description of the articles.

5.4 Bulk Operations

Database bulk operations are more performant than a sequence of several independent operations. Publishing an article involves the association of a variable number of tags to the corresponding article. To improve the performance of this operation, one can perform a bulk operation that adds all tags in a single query. Although the latency of this operation depends on the number of tags, it increases much less when we execute a bulk operation than in the alternative scenario.

5.5 NoSQL Database

To publish an article, we have to associate a set of tags with this article. In a relational paradigm, one would have to resort to an additional table to store such tags, which would be separated into different registers. Alternatively, by using the MongoDB NoSQL database one can define the article data model as a container of its own tags. Updating a single database record is significantly faster than inserting multiple records. In contrast to what happens in a relational paradigm, the execution time associated with the tag insertion remains constant and independent on the number tags.

5.6 Dynamic Inlining and Post-onload Download

Even though scripts and styles are external resources to Web pages, obtaining such pages can benefit from using inline scripts and styles. This accelerates the transfer and rendering of such content. The dynamic inlining technique allows to decide whether (i) the responses returned to the clients should include the scripts and styles inlined on the pages, or (ii) they should contain only references to external resources, which should be served separately.

We implemented dynamic inlining on the registration page. It was necessary to select the styles

for the correct presentation of the page, to extract them from the respective files and to place them in an HTML `<style>` tag. In addition, it was necessary to implement application code for the view that serves this page, in order to determine if the client request has a cookie and to decide the appropriate response to return to the client, i.e., whether to use internal style sheets or references to external style sheets.

Post-onload Download allows the reuse of resources that were transferred inline, along with the required page, thus not being cached. After rendering the page on the client side, in subsequent requests the external resources can be downloaded and cached. To implement Post-onload Download, it was used a script that, after loading the page, transfers the scripts and styles present in external files, so that they can be placed in the browser cache and reused in the future.

5.7 Validating Forms in the Front-end

For data consistency and security reasons, form validation is typically performed on the back-end. To eliminate the computational burden on the server and to accelerate form validation, it must also be performed on the front-end. This technique was implemented on the registration page, by adding a script that verifies if both passwords in a registration form are equal. Validation on the front-end does not offer any performance advantage when the submitted forms are valid. However, when there is an irregularity in the form, it is advantageous to report such error to the user when he tries to submit the form. This eliminates a server request.

5.8 Asynchronous Operations and Partial Update of Web Pages

Operations such as deleting drafts or published articles do not require reloading the Web page completely. The page that contains the user published and draft articles is dynamic. The server returns a page with a common skeleton, which includes only the required list. Later, if the user wants to view the other list of articles, it will be transferred from the server, using AJAX technology, and rendered in the client browser using JavaScript. This eliminates reloading the page completely and rendering the base skeleton again, when the client switches between lists. Any operation that does not require a complete page reload can be performed asynchronously. So the user can continue interacting with the page while the required operation is being processed. Therefore, deleting an article was implemented asynchronously, with an AJAX request to the server followed by the page update on the client side.

5.9 Pagination

Sometimes the number of items that result from processing the user request is so extensive that it is impractical to build and return a response that includes the complete set. In these cases, the technique of data pagination must be used. Pagination operates on several levels, from selecting the information from the database, through building the corresponding response and transferring it over the network, to rendering the data on the client device.

5.10 Asynchronous Task Queues

Asynchronous tasks queues allow to perform certain operations asynchronously, without the application being blocked waiting for the execution result. Registering a new user in the application involves sending a confirmation e-mail, which is a relatively time-consuming process. To mitigate performance issues associated with the synchronous execution of e-mail delivery, and to be able to return the registration confirmation page to the user as quickly as possible, the asynchronous processing mechanism was used. Upon submitting a registration form, the application creates and orders the execution of an asynchronous task that will be responsible for sending the confirmation e-mail, returning immediately the register confirmation page to the user. The `delay` function of `Celery` was used to run tasks asynchronously.

5.11 In-memory Caching

There are several situations where the content displayed on a page results from the execution of actions that produce dynamic results, such as retrieving information from a database. To reduce the time required to get this information, it is convenient to store it temporarily in a cache memory to speed up the processing of subsequent requests. However, it is necessary to define a maximum time during which the data can be reused, since its state can vary, generating inconsistencies between the source and the cache.

In certain situations, the cache must be populated proactively and not as a result of client requests, since the first request will be penalized by the need to populate the cache. The solution is to use asynchronous task queues, executing a task at a given frequency to populate the in-memory cache with information from database. In-memory caching, with

a periodic task, allowed us to decrease the latency of the process that generates the pages with articles searched by topic.

5.12 Combining and Placing Scripts and Styles

It is often preferable to combine resources of the same type, because so clients only have to transfer once each type of resource. Using external and combined scripts and styles, the clients can reuse them among distinct pages. The scripts and styles used in the application are provided as external resources and include `Cache-Control` or `Expires` headers, to allow them to be stored in the proxy cache or in client browser. When the styles used in the different pages of the application are combined in a single file conflicts arise, which need to be eliminated. Thus, to suppress internal stylesheets it was necessary to perform its re-factorization. Similarly, the scripts defined in HTML pages were removed and replaced in external and reusable files.

To render the application pages in the client's browser progressively, it is convenient to refer the external stylesheets at the top of the pages (`<head>` section). Instead, the scripts must be referenced at the end of HTML pages (`<body>` section), to allow faster rendering. Placing scripts in other parts of the page could cause them to be transferred early, temporarily blocking the renderization.

5.13 Scripts and Styles Minification

All scripts and styles were minified with the `Minify` tool. Removing unnecessary characters, as well as the comments on the original code, contributes to decrease the latency associated with their transfer. Minification must occur before the files are deployed to the server. Minification can be more or less rigid. It is possible to change the names of the functions and variables, using names with fewer characters.

5.14 HTTP Compression

To further reduce the latency of HTTP responses, we have used the compression technique, which allows us to reduce the size of the transferred content. Compression can be performed at the application level, using the `Django` compression tool. In this case, it is only necessary to include a reference to the `GZipMiddleware` class in the `MIDDLEWARE` variable of the `Django` project configuration file. Another alternative is to use the capabilities of the `Nginx` Web server, through the presence of the directive `gzip on;` in its configuration file. In any case, HTTP responses will now include the `Content-Encoding` HTTP header, parameterized with the value `gzip`. This technique proves to be advantageous since the time involved in compression and decompression is lower than the reduction in latency pages' transfer.

5.15 Server-side Template Caching

In some cases, server-side template caching completely eliminates operations such as database accesses and template rendering. All pages that are targeted for HTTP caching must also be stored on the server side, to avoid they need to be recalculated. One of the main solutions for template caching on the server is a reverse proxy, such as `Varnish`. All requests made to the application go first through `Varnish` and, if the appropriate response in its cache, `Varnish` returns it directly to the client. Otherwise, `Varnish` contacts the Web server to obtain a response for the client.

By default, `Varnish` does not cache pages whose requests or responses have cookies. To enable the return of the pages stored in cache, it was necessary to remove all the cookies present in the client requests. For this purpose, code was added to the `vcl_recv` routine, which is executed by `Varnish` immediately after parsing each request and before checking the existence of a cached response. This code removes the cookies from all page requests whose URL is not declared in the test of the `if` statement included in the `vcl_recv` routine. Deleting cookies allows `Varnish` to return the responses, rather than contacting the application server. When the `Varnish` receives a request whose URL is declared in the `if` statement test, cookies remain unchanged and the request is forwarded to the application server in order to get the corresponding response.

5.16 Content Delivery Networks

CDNs create replicas of the Web applications static content in different locations, putting the content closer to the target users and decreasing its transference time. We used the `Hostry` CDN service, which configuration includes information about the address and URL of the static resources server. On the origin server side, it is also necessary to rename the references to static resources in the HTML pages. The new location will be a URL provided by the CDN service, followed by the resource name. In Django, the definition of the location of static resources is done through the variable `STATIC_ROOT`.

Only the styles and scripts created specifically

for the developed application, as well as the images submitted by the users, are served by the configured CDN. Furthermore, the styles and scripts of third parties, namely `bootstrap.min.js`, `bootstrap.min.css` and `jquery.min.js`, are served by specific CDNs.

6 RESULTS

Table 1 shows that the pagination technique when applied to database accesses allows a significant decrease of the data retrieval latency. Additionally, it highlights the benefits of storing data in memory, in which case the latency can be reduced almost 50 times, in contrast with accessing database. Table 2 documents the difference in latency verified with the application of the pagination technique at the network level. When we reduce the content size 10 times, we get a 5-fold reduction in the latency of the download.

Table 1: Memory vs. database access time.

Source	Operation	Latency (ms)
Database	Retrieve 500 articles	51.77
	Retrieve 50 articles	14.78
Memory	Retrieve 50 articles	0.3

Table 2: Influence of network pagination on latency (s).

	Network	
Operation	Slow 3G	Fast 3G
Download a 1.2 MB page	26	7
Download a 120 KB page	5	1.2

Table 3 shows the latency values associated with the insertion in the database of a variable number of article tags. Regarding the relational paradigm, it can be verified that bulk operations provide much faster response times than individual insertions. The latency increases with the number of tags, but this trend is much more pronounced when inserted one by one. Furthermore, the table shows that NoSQL allows to maintain the response time independent of the number of records inserted in the database.

Table 3: Latency of database operations (ms).

	Number of tags			
	1	3	6	12
Individual insertion	5.2	29.6	33.1	63.9
Bulk insertion	5.2	7.9	8.3	10.6
NoSQL	5.3	5.5	5.4	5.5

Table 4 presents the latency associated with content retrieval, highlighting the benefits of using HTTP caching. While the transfer time of several application pages depends on its size, obtaining any page from the browser cache takes only 5 ms approximately, which is much faster than any download made from the application server.

Table 4: Influence of HTTP caching on latency.

Source	Page	Size	Latency
Network	Articles by topic	122 KB	5.03 s
	User profile page	13 KB	2.54 s
	Article page	11 KB	2.29 s
	Search page	3.5 KB	2.08 s
B. cache	Any page	Variable	5 ms

Table 5 contains the latency for five operations, highlighting the advantages of applying other performance optimization techniques. Storing session data in-memory allows to reduce 40 times the latency of the logout operation. Sending registration confirmation emails asynchronously reduces latency 1700 times. The time required to select a user profile data from the database, based on his/her username, is 14 times lower when an index is used on the corresponding attribute. HTTP compression allowed us to reduce the latency associated with downloading a 122 KB page to half of the original time. Finally, the CDN made it possible to reduce the download time of a 217 KB image by 10 times.

Table 5: Latency of diverse operations.

		Latency (ms)	
Technique	**Operation**	**NonOpt**	**Opt**
In-memory sessions	Logout	123	3
Async task	Send an e-mail	1700	1
DB indexes	Get user data	55	4
HTTP compression	Download a 122K page	4490 (122K)	2120 (5.2K)
CDN	Get a 217K image from Washington	1110	106

Tests were carried out with the GTmetrix performance analysis tool, which grades the website pages according to their load speed and performance best practices, having obtained high grades for several application pages. The performance score provided by GTmetrix is determined by running both Google PageSpeed Insights tool and Yahoo YSlow. The homepage, which lists the articles of a specific topic, got 95% and 96% scores (figure 2). The article page got 87% and 96% scores (figure 3). A score of 90 and above is considered fast, 50 to 90 is considered average, and below 50 is considered to be slow.

GTmetrix also provides recommendations regarding performance optimization. For example, the images served by the application should be scaled, in order to reduce the network traffic associated with its transfer. Furthermore, the static components of the application, such as scripts, styles and images, should also be subject to HTTP compression and caching. Beyond these aspects, we can confirm in figures 2 and 3 that the application is highly optimized.

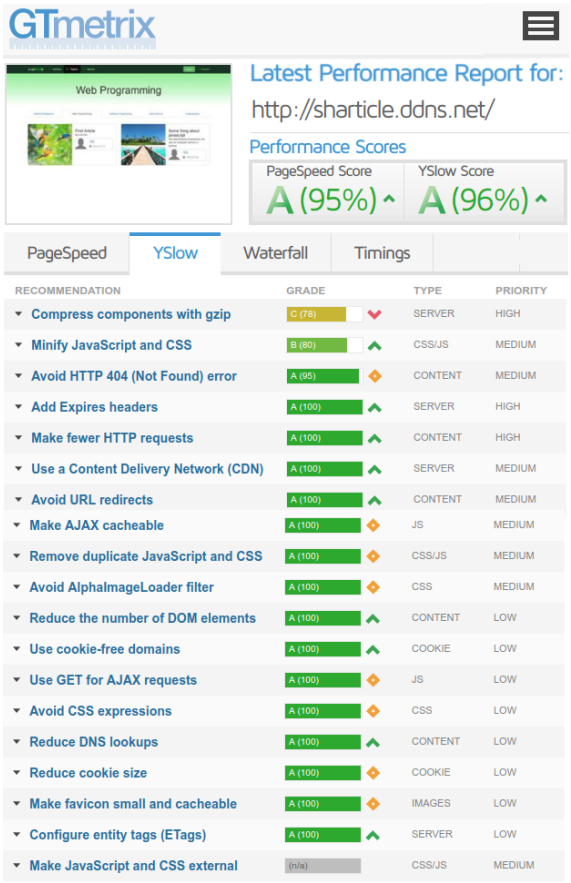

Figure 2: YSlow and PageSpeed scores for the homepage.

Figure 3: PageSpeed and YSlow scores for article page.

7 CONCLUSIONS

There is a conflict among the quality attributes considered during Web applications development. For example, database indexes have a positive impact on the latency of reading the associated data, but it influences negatively the insert and update operations. Therefore, a higher priority should be given to the operation that is intended to perform best.

The `Django` framework supports building high-performance applications, providing a number of tools to accelerate the execution of operations. Some examples are HTTP compression of the pages returned by the application, handling HTTP caching, caching user session data in memory, and caching the generated pages. Since it is a full-stack framework, `Django` is not globally as fast as other Python frameworks. NoSQL solutions are not natively supported by the framework. An alternative is the `Djongo` technology, which has limitations in performing certain operations. For example, it is not possible to execute textual search queries with a `OR`-clause, i.e., a logical disjunction.

The HTTP protocol has evolved over time. The latest HTTP/2 version exhibits several optimizations, making it unnecessary to implement some of the

described optimizations. Server push is an HTTP/2 mechanism that enables sending proactively Web resources without requiring an explicit request. This mechanism eliminates the necessity to implement certain performance enhancement techniques, such as combining features of the same type and inlining styles in the home page. The remaining optimizations that were discussed are not affected by the new version of the HTTP protocol and should continue to be applied.

In the future, more performance optimization techniques can be explored, as well as other performance metrics besides latency, such as throughput and resource utilization. Regarding databases, there are mechanisms, such as sharding, whose objective is sharing the information among multiple partitions, as well as data replication, which implies the existence of several copies of the same information. These techniques allow to spread the database access load across several processing nodes, reducing resource utilization per node and possibly increasing throughput and decreasing the system's response time. It is also possible to replicate the application servers, taking advantage of load balancing mechanisms, which aim to distribute client requests across multiple server nodes. On the client side, there are technologies such as `Service Worker`, which is a service placed in the browser and acts as a proxy, intercepting HTTP requests, and can serve responses from its cache. This technology gives users the ability to use certain parts of the application while being offline, eliminating the need to contact the server and decreasing the latency associated with obtaining certain content.

ACKNOWLEDGMENT

This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2019.

REFERENCES

Anastasios, D. (2016). Website Performance Analysis and Optimization. Master's thesis, Harokopio University, Greece.

Arcos, D. (2016). Efficient django. EuroPython.

Cao, B., Shi, M., and Li, C. (2017). The solution of web font-end performance optimization. In *10th International Congress on Image and Signal Processing*. IEEE.

Cerny, T. and Donahoo, M. (2010a). Evaluation and optimization of web application performance under varying network conditions. In *Modelling and Simulation of Systems conference*.

Cerny, T. and Donahoo, M. (2010b). Performance optimization for enterprise web applications through remote client simulation. In *7th EUROSIM Congress on Modelling and Simulation*.

Connolly, T. and Begg, C. (2005). *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison Wesley.

Fielding, R., Gettys, J., and Berners-Lee, T. (1999). Hypertext transfer protocol - http/1.1. RFC 2616.

Google (2017). Google page speed insights and rules. developers.google.com/speed/docs/insights/rules. Consultado em 19/10/2017.

Greenfeld, D. R. and Greenfeld, A. R. (2017). *Two Scoops of Django 1.11*. Two Scoops Press.

Grigorik, I. (2013). *High Performance Browser Networking*. O'Reilly.

Grigorik, I. (2017). Web fundamentals: Optimizing content efficiency. https://developers.google.com/web/fundamentals/-performance/optimizing-content-efficiency/. Consultado em 19/10/2017.

Holovaty, A. and Kaplan-Moss, J. (2009). *The Definitive Guide to Django: Web Development Done Right*. Apress.

Horat, D. and Arencibia, A. (2009). Web applications: A proposal to improve response time and its application to moodle. In *Computer Aided Systems Theory (EUROCAST)*, pages 218–225. Springer.

Manchanda, P. (2013). Analysis of Optimization Techniques to Improve User Response Time of Web Applications and Their Implementation for MOODLE. In *International Conference on Advances in Information Technology*, pages 150–161. Springer.

Meier, J., Vasireddy, S., Babbar, A., and Mackman, A. (2004). Improving .net application performance and scalability. https://msdn.microsoft.com/en-us/library/ff647781.aspx (accessed in 12/oct/2017).

Ossa, B., Sahuquillo, J., Pont, A., and Gil, J. (2012). Key factors in web latency savings in an experimental prefetching system. *Journal of Intelligent Information Systems*, 39:187–207.

Shivakumar, S. and Suresh, P. (2017). An analysis of techniques and quality assessment for Web performance optimization. *Indian Journal of Computer Science and Engineering*, 8(2):61–69.

Smith, P. (2013). *Professional Website Performance: Optimizing the Front End and the Back End*. John Wiley and Sons, Inc.

Souders, S. (2007). *High Performance Websites*. O'Reilly.

Subrayen, B., Elangovan, G., Muthusamy, V., and Anantharajan, A. (2013). A Case Study for Improving the Performance of Web Application. *International Journal of Web Technology*, 02(01):17–20.