Content Delivery Network integration for resource caching and video streaming

Stefan Grigore

Padoq Ltd.

September 2019

Fourth Year MEng Report
MEng Computer Science
School of Computer Science - University of Manchester

Contents

Abstract — **To improve the access time and availability of media resources, a Content Delivery Network (CDN) is required. This report describes the integration of such a network with the back-end infrastructure of the mobile apps of Padoq, a startup developing a community network platform. The implementation has been extended to enable the hosting and delivery of video streams allowing the introduction of video posts on the apps. The performance of the system is analysed, and testing and continuous integration methods are explained.**

Contents

1 Introduction

1.1 Padoq

Padoq is a Manchester-based startup developing a community network platform designed to allow organisers to better manage and engage with their communities (Crunchbase, 2019). The Padoq app for Android and iOS enables users to request and collect payments, chat with other users, vote in polls, share photos and files or plan and promote events (Padoq, 2019).

1.2 Project goals

Significant infrastructure is required to manage a large volume of resources and traffic while maintaining security. A Content Delivery Network (CDN) was introduced during this project to mitigate the load involved in authenticating users and serving resources.

One significant aspect of interactions on social media platforms is the ability to share video posts. The integration with the CDN also provided a reliable way of serving these videos, enabling the introduction of video posts on the app while minimising the load on the infrastructure.

The scope of this project is to integrate a highly scalable CDN with the back-end of Padoq. This integration would support the caching and serving of resources. The implementation would require to maintain the existing security and enable the introduction of new features.

The report will detail the different approaches for increasing the availability, performance and scalability of the Padoq app, the different techniques used to evaluate and ensure the reliability and stability of the system, and how these new capabilities have been used to develop new features such as video streaming on the app.

2 Background

2.1 Padoq posts

Central to the Padoq app are the feeds of the different groups (padoqs) that users are part of. Figure 1 shows posts in a padoq specialised in movies.

Posts can contain attachments such as images or documents. As the user base of Padoq grows, serving all these attachments to clients across different regions could pose problems to the existing infrastructure.

Figure 1: Posts in a padoq

2.2 The Cloudflare CDN

'Cloudflare is a company that provides content delivery network (CDN) and distributed DNS services by acting as a reverse proxy for websites' (Anicas, 2015). The Cloudflare CDN consists of a global network of data centres that cache and serve content at nodes close to the users (Cloudflare, 2019). A significant component of the CDN is the Cloudflare Workers feature. The workers provide the capability of JavaScript execution at the nodes of the CDN. The system enables developers to support existing applications or create new serverless ones without configuring or maintaining infrastructure (Cloudflare, 2019).

2.2.1 Cloudflare Workers

Cloudflare Workers provide a serverless execution of scripts at the edges of the Cloudflare CDN, which can support the offloading of processing and functionality of applications to them. Different to web workers, which are scripts executed from an HTML page in the background of an individual machine (i.e. browser application) *'independently of any user interface scripts'* (Web Hypertext Application Technology Working Group, 2019), Cloudflare Workers run on Cloudflare's Edge Network, *'a growing global network of thousands of machines distributed across hundreds of locations'* (Cloudflare, 2019). Each of those hosts an instance of the Workers runtime, which uses Google's V8 engine (Cloudlfare, 2019).

Worker scripts can be configured to listen to incoming requests to the domain proxied by Cloudflare. An event listener then intercepts the request, runs the worker script and then produces a response back to the client. The worker can choose to pass the request forward to the server of the domain and then retrieve the response to the client, or to resolve the request at the edge of the network, without the need to make a request to the server, thus minimising the load. Figure 2 illustrates this workflow.

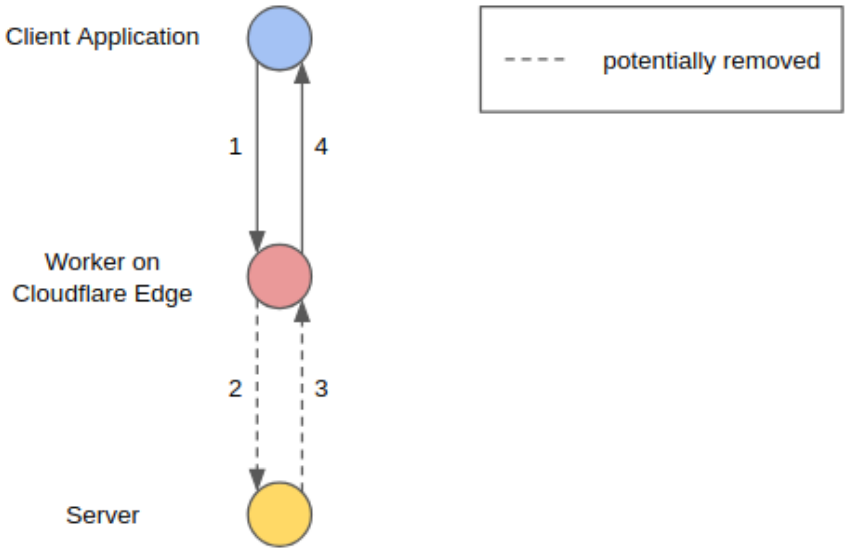

Figure 2: The proxying done by the worker

While proxying a request, the worker can perform subsequent requests to external services or other endpoints of the server before choosing to pass the request through or respond to it.

3 Design

3.1 Choosing the CDN

The various features of CDN providers and their performance have been taken into consideration when choosing one. Among other providers such as Amazon, Cloudflare seemed to be the easiest to configure, while providing features such as video streaming or the proxying of requests. These features and the technologies used would enable easy integration with the existing Padoq stack, all at a competitive price.

A Python script has been implemented to test the latency of Cloudflare by pinging the Padoq domains which were set up to be proxied by it. For 500 requests at 0.5s intervals, the total time added to the 99th percentile by the proxying of Cloudflare was 233ms, which was an acceptable value.

Web Application Firewall capabilities have been assessed, and Cloudflare was found to have a wide variety of firewall rules developed both by them and by the Open Web Application Security Project (OWASP) community.

3.2 Caching media resources

Cloudflare Workers can use Cloudflare's Cache API to cache responses based on requests. The cache works as a map where the key is the request, and the value is the response. For security, however, authentication handling had to be implemented over the course of this project. The worker was configured to hold in its runtime memory tokens that have been previously verified by the Padoq authentication server. When subsequent requests to access resources using the same credentials are made, they would be searched in memory and approved directly by the worker, which will then serve the cached response without making a request to the server.

The Padoq API consists of several modules. Relevant for this section is the authentication module called Gatekeeper, and the resource module. The Gatekeeper server is used, among other things, to issue and verify authentication tokens. Previously, when a client would want to access a resource from the resource server, the resource server would need to verify the token via a request to the authentication server. Figure 3 illustrates this workflow.

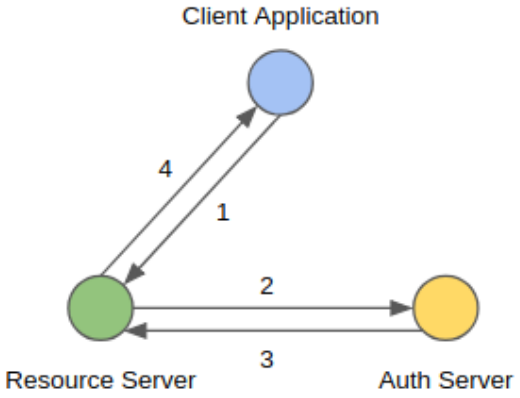

Figure 3: The initial workflow

With workers in place, the worker now makes requests to the authentication server to verify tokens, stores them and caches the resource responses, being able to eventually fulfil resource requests itself, greatly reducing the load on the resource server. Figure 4 shows this new method.

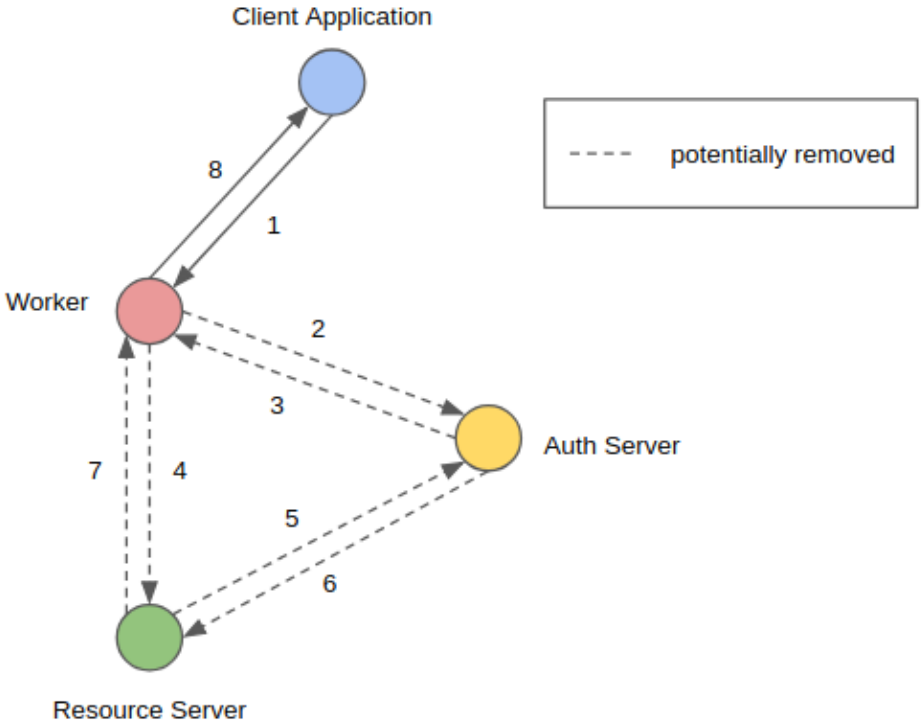

Figure 4: Proxying the workflow via the worker

In the case of responses that have been cached by the worker, the number of requests needed to fulfil the query goes down from 2 in the previous workflow (i.e. from client to resource server and from resource server to the Gatekeeper), to only 1 (from client to worker). In the case where an uncached token is used, but it is invalid, the total requests is 2 (from client to worker and then from worker to Gatekeeper), and the resource server is never called, which is an improvement. In the case where a valid token that has not been cached is used, however, the total requests it is 4 (from client to worker,

worker to Gatekeeper, worker to the resource server and resource server to Gatekeeper). This tradeoff is advantageous, as most resource requests are made with the same token, which will be cached by the worker. Furthermore, this method removes the risk of a denial of service attack on the resource server, as requests with invalid tokens will never pass through to it.

The memory constraints for worker instances imposed by Cloudflare are 128MB of memory, including the source code of the deployed script (Cloudflare, 2019). At the time of implementation, the CPU time limit for processing requests, excluding subrequests made, was 10ms. Cloudflare recently increased this limit to 50ms.

A limit has been imposed on how many tokens an instance of the worker can hold at any time to comply with the memory and CPU time constraints.

3.2.1 Maintaining the runtime memory of the worker

All tokens are evicted from the memory of the worker whenever the maximum number allowed is reached. Different versions of the token cache eviction such as periodic evictions based on token expiration timestamps were implemented and tested but were not kept as to not consume resources under the 10ms CPU time constraint put in place by Cloudflare at the time. Tokens are stored in memory every time the Gatekeeper server successfully authenticates them. On subsequent requests with that token, if it is found locally and it did not expire, the query is fulfilled without an authentication request to the server. If the token expired, however, it would be evicted from memory.

3.2.2 Verifying the membership of users

The Padoq user base is split into groups (padoqs), and not all padoqs are public. Even though a user might request a resource from a padoq with a valid token, if the user is not part of the padoq that that resource belongs to, the request should fail. The worker has to handle this when trying to fulfil queries without subsequent requests to the server.

The worker was modified to also store in memory the resource ids along with the membership associated with the used tokens and requested resources. To get access to the membership details of users, the worker makes subsequent requests to the Padoq API to obtain an authentication token of its own using a secret stored as an environment variable. Modifications to the Padoq API were also made to allow this.

The logic of the worker now included checking whether the padoq membership of the user matches the padoq that the resource requested belongs to. The token that the worker uses to get the membership information is refreshed periodically by the worker. A limit on how many resources the worker can hold information about has also been imposed to comply with memory constraints.

3.3 Streaming videos

Previously, the Padoq app would not support video attachments to posts in padoqs. Both the Padoq API, Padoq app and the worker would be changed over the course of this project to implement this functionality.

Cloudflare enables the storage, encoding and distribution of videos from their platform at a competitive price. To integrate this, the client requests to the Padoq API the creation of a video attachment location for a post, and the Padoq server makes a subsequent request to the worker that holds the necessary credentials to create a video upload location on the Cloudflare platform. The client then uploads the video at this upload location which is proxied by the worker to handle authentication. Figure 5 illustrates this workflow.

Figure 5: The Padoq video upload workflow

After the video is uploaded, a stream link and thumbnail URL are provided by Cloudflare, which are stored and exposed by the Padoq API for the clients to use to display videos in posts.

3.3.1 Signing Stream URLs

Posts can be copied across Padoq. We would not want to duplicate the video attachments of these posts on Cloudflare, however, to decrease the hosting costs. We would want to have separate references (i.e. stream links) across the app that point to the same video hosted on Cloudflare. When all these references are removed from the database, then the video would also be deleted from Cloudflare.

Cloudflare provides the ability to create stream links by signing a JSON Web Token (JWT) which can be used instead of the video ID to access the video and its thumbnail.

The videos cannot be signed, however, until they have been fully processed by Cloudflare after uploading them. Cloudflare also provides a webhook service which can notify servers (or workers) that the videos have been processed and are ready to stream or be signed. An endpoint was implemented on the worker to receive this webhook and then sign the video URLs, and then update the details of this post attachment via the Padoq API. The API was also modified to support this functionality.

Now copying a post with a video attachment is only a matter of specifying a reference in the form of a video ID to the worker, and the worker would sign the necessary URLs and provide them to the Padoq back-end without creating a new video on Cloudflare.

4 Implementation

4.1 Building the Worker script

The Cloudflare Worker scripts are written in JavaScript to run on the JavaScript V8 engine that the CDN nodes use. The scripts can be deployed by introducing the code into the editor available on the Cloudflare account dashboard, or they can be uploaded via a PUT HTTP request at the Cloudflare REST API. The latter has been implemented to facilitate continuous integration.

4.1.1 Organising the Worker as a Node.js project

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine (Node.js Foundation, 2019). It follows the "JavaScript everywhere" paradigm, where the same language is used for client and server-side scripts. This environment enables the development of server-side scripting, which would facilitate running the script locally for testing, continuous integration and development.

Npm (short for Node Package Manager), is a package manager for JavaScript Node modules. It is the default package manager for the Node.js runtime environment. Its command-line interface can be used to install modules from its online database of packages, and it can be used to create and run scripts for testing, building and deploying Node.js applications.

Over the course of the development of the Worker script, external packages would be integrated via npm to facilitate testing, continuous integration and the use of other utilities for developing the features.

4.1.2 Configuring the Worker

The worker scripts need to be associated with Padoq API paths so that they proxy the connection to them. The documentation of the Padoq API routes can be found at https://app.dev.padoq.com/swagger-ui.

The media resources of Padoq can be found under the /resrc/media/* path of the API. The worker was configured to listen to the traffic of routes matching the pattern of the /resrc/media/* path and to apply the script while proxying this traffic. An event listener applies a function to the event before sending a response. The event object that is received contains the request and the details associated with it such as the URL that the request was sent to, the payload and any headers or authorisation.

4.1.3 Caching media resources

To cache the response, Cloudflare workers use a cache API strongly influenced by the implementation of the caches of the Web API. The cache is a global memory that can be accessed by any instance of the workers which maps request objects to response objects. Cloudflare handles and optimises the eviction of this global cache. To implement the caching of responses, when a request is intercepted from the event listener, this request is searched in the global cache that all workers have access to, and if it is found then the resulting response is sent back to the client without the need of a call to the Padoq server.

4.1.4 Caching authentication tokens and memberships

Padoq uses the OAuth 2.0 protocol for authorisation. To access media resources and perform other actions, clients need to gain authorisation via the Gatekeeper server. The token consists of a universally unique identifier (UUID), which is a 36 character string.

The worker was modified to hold in its local runtime memory tokens that have been verified by the authorisation server. For each of them, we store an object containing their value, the expiration timestamp and a list of the IDs of the padoqs that the user associated with that token is a member of. At the same time, the worker also keeps in memory a map of resource IDs pointing to the IDs of the padoqs that they belong to. When the token is not expired, and its membership matches the membership of the resource that is requested, then the response is given from the cache without the need of a request to any of the Padoq servers.

An npm library called ”moment” is used to parse and check the expiration of tokens. To integrate the code of this library into the final script ready to be uploaded via the Cloudflare API, another npm library called ”webpack” is used. Webpack is a module bundler used to bundle JavaScript resources into a single file, ready to be deployed in a browser application, or this case, as the worker script at the edges of the CDN. The webpack command-line interface is used to specify the files to be bundled, resulting in a command that can be used to produce a single file containing all the dependencies of the worker script, including external libraries, credentials and variables from environment files.

To access the padoq memberships of users and resources, the worker needs to access the Padoq REST API routes exposing these details. Previously no endpoints were exposing the metadata of post attachments or the membership of users. These endpoints were implemented as part of this project. A client id and secret specific to the worker have been added to the database of the API so that they can be used by the worker to obtain an authentication token with the scope of reading user memberships and resource metadata.

The Padoq API uses the Vert.x toolkit for building reactive applications on the JVM. It is an event-driven, non-blocking framework enabling Java applications to handle large amounts of concurrency using a small number of kernel threads (Eclipse Vert.x, 2019).

To implement the new API endpoints using this framework, new /token/{token}/member-padoq and /media/{id}/metadata routes have been added, producing JSON responses containing the details of the

padoqs that the user associated with the specified token belongs to, and the ID of the padoq that the media resource belongs to respectively.

4.2 Introducing video attachments to posts

For the introduction of video attachments in posts, a new type of media resource would need to be supported. Within the worker, a new route would be created, /video, which would act as a proxy to the Cloudflare Stream API for managing videos, converting the Padoq authentication that the Padoq apps use into the credentials that the Cloudflare API requires to create and accept video uploads. These credentials would be stored as environment variables bundled into the script of the worker via webpack.

Cloudflare uses the TUS resumable file upload protocol, which is an open-source, HTTP based protocol (Tus.io, 2019). Clients for both Android and iOS are available. For this project, a test Android application has been forked from the example application provided by Tus.io and modified to act the way a Padoq mobile client application would when authenticating, creating and uploading video post attachments.

The workflow to create post attachments via the Padoq API is that the client creates attachment metadata using a POST request at the /padoq/{name}/post/{postId}/version/{postVersion}/media/ endpoint which will return the location where the media needs to be uploaded.

On the Padoq API side, to persist the attachment details and structure the interactions with Cloudflare, the Data Access Object (DAO) structural pattern has been implemented. This pattern aims to isolate the application/business layer from the persistence layer (i.e. relational databases or any other persistence mechanism) (Baeldung, 2019). The workflow is that the client makes a request at an endpoint exposed in the handler layer, the handler layer then uses the service layer responsible for coordinating the logic of persistence operations, and the persistence layer exposes these operations via Data Access Objects to be used by the service layer. Figure 6 illustrates this workflow.

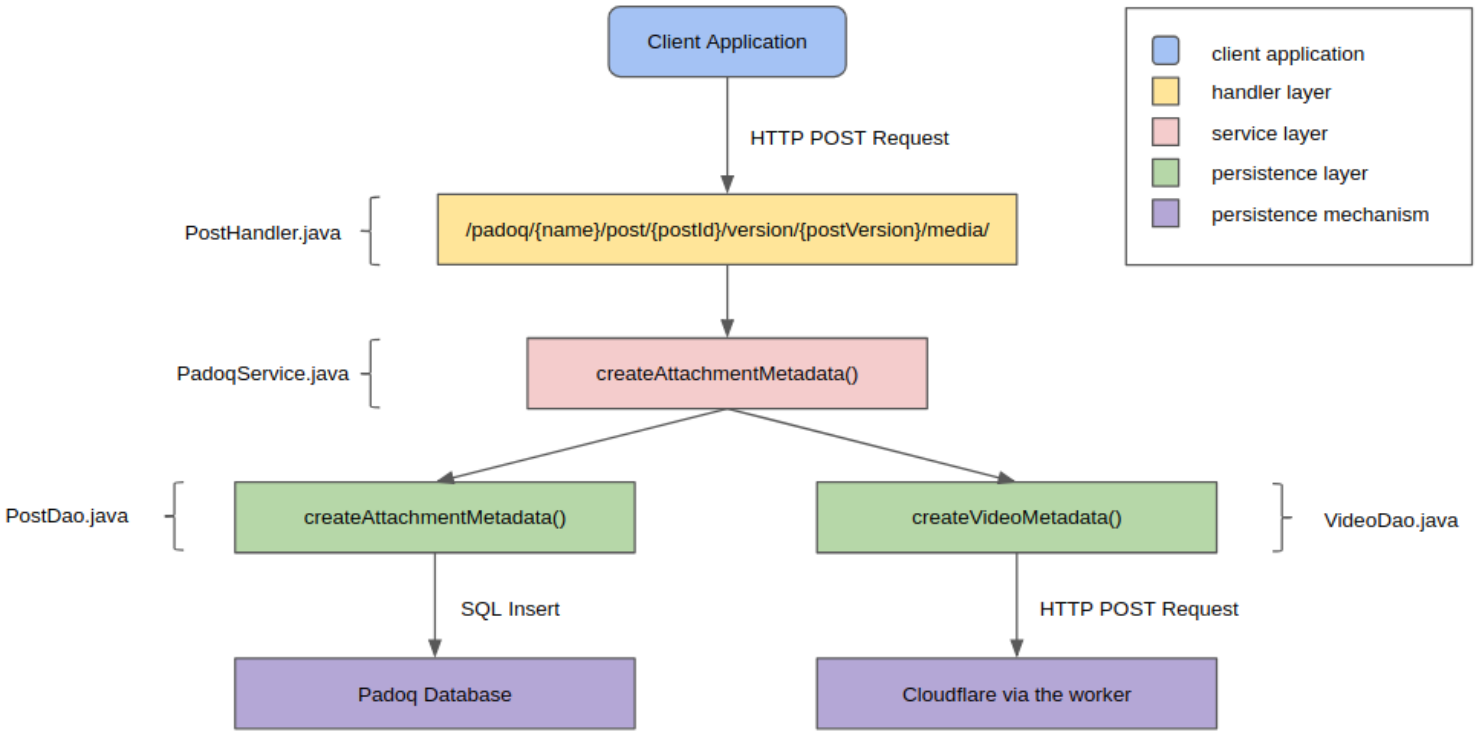

Figure 6: The workflow for creating video attachments in posts

The PadoqService which manages and provisions padoqs makes a call to the VideoDao responsible for creating videos hosted on Cloudflare. The VideoDao achieves this via an HTTP POST request to the worker, which verifies the user credentials and then uses the stored credentials of the Padoq Cloudflare account to create a video via the Cloudflare Stream API. The Stream API then passes the ID of

the resulting video to the worker, and the worker responds to the PadoqService via the VideoDao with this ID. The requests are handled with the use of the Reactive Extensions (Rx) in Java, which function in a ”publish-subscribe” manner. When the HTTP call from the VideoDao is made, the PadoqService is subscribing to an Observable of the future result of the operation. The asynchronicity of this technique enables the non-blocking implementation of I/O, which is appropriate compared to other blocking alternatives such as an HttpUrlConnection.

After the PadoqService obtains the ID of the video, it uses the PostDao to persist data about the attachment of the post in the Padoq database. It achieves this by using an Object Relational Mapping from methods mapped to SQL queries working with objects such as posts and attachments represented in a PostgreSQL relational database. An attachment object is created and persisted by using the SQL commands mapped by the PostDao.

Transactional advice has been implemented by using the Spring Aspect-Oriented Programming (AOP) framework to maintain the integrity of database operations. For the transactions creating the video attachment metadata, the advice has been set so that those transactions never propagate, i.e. they execute non-transactionally and throw an exception if one exists. The technique is used because the createVideoMetadata method could potentially hold a database transaction open while a call to Cloudflare is made.

Among the details of the video attachment that are retrieved from Cloudflare and persisted in the Padoq database is the stream link to an M3U8 file, and the thumbnail URL which points to a PNG image. These details can be used by the client to display the videos. Figure 7 shows a post with a video attachment in the app.

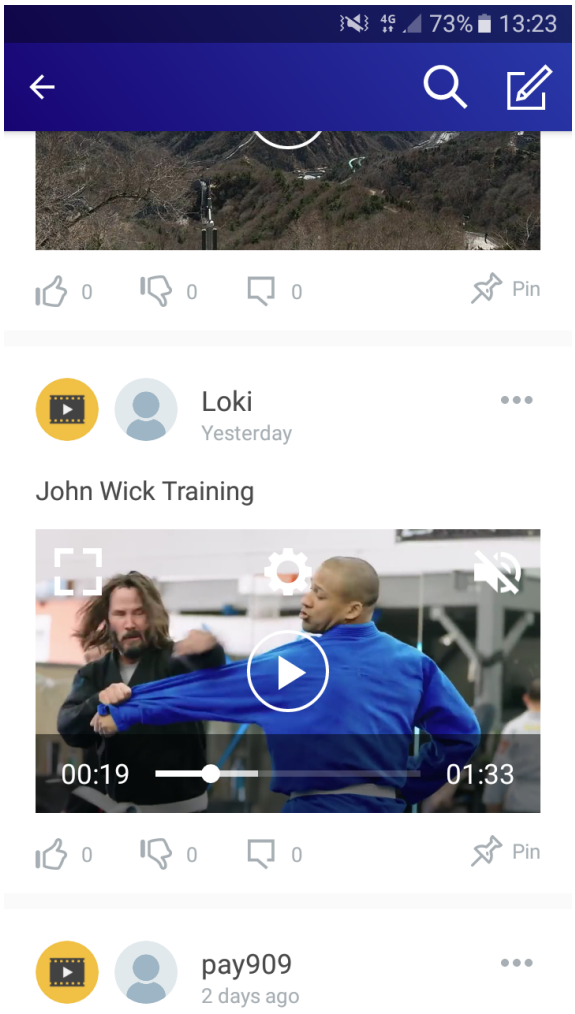

Figure 7: A video post in a padoq

4.2.1 Signing Stream URLs

After the client finishes the upload via the TUS protocol at the upload location retrieved after the attachment metadata has been created, the video needs to be processed by Cloudflare. After processing, Cloudflare has been configured to send a webhook at a new /video-status route of the worker signalling that a video has been processed. The webhook contains the metadata of the attachment, such as the IDs of the padoq and post that it belongs to. These details were sent to Cloudflare when the video was created initially.

Upon receiving such a webhook, the worker will produce links for the stream and thumbnail containing signed JWTs created with credentials stored by the worker. The process of signing the JWTs would take too many resources to be performed locally and still maintain the 10ms CPU time requirement imposed by Cloudflare at the time, so the signing process has been offloaded to an external service and retrieved via HTTP subrequests, as those do not count towards the CPU time limit. After producing these signed URLs, the worker calls a newly implemented /padoq/{name}/post/{postId}/version/{postVersion}/media/{id}/status path of the Padoq API. This new endpoint updates a hasContent flag that has been introduced to attachment objects, which indicates whether the content of the video has been processed and persisted on Cloudflare.

Now, when copying a post which contains a video attachment, the Padoq API makes an HTTP POST request to the worker by using the same publish-subscribe technique, specifies a video ID as a reference and receives an Observable of the future metadata of the video attachment copy containing the signed URLs. New uploads of the video are not made to Cloudflare this way, improving efficiency and decreasing hosting costs. When deleting a video post via the Padoq API, the implementation now searches all attachments associated with the corresponding video hosted on Cloudflare, and if there are no copies, the video is also deleted from Cloudflare via a request to the worker.

5 Testing

Several components were involved in the development of this project, specifically the Cloudflare Worker script, the Padoq API and the Padoq mobile client apps for Android and iOS.

Extensive testing has been implemented for the Cloudflare Worker script, in the form of both unit tests and integration tests. The Padoq API lacks unit and integration test implementations, possibly because of the ease with which it can be deployed and tested locally. The video integration has been tested manually on an app made specifically for this feature, and automated UI tests were not needed for this app.

The scope of this project would have been increased significantly with the introduction of the development of unit, integration and UI tests for the Padoq API and the Padoq apps for Android and iOS, which would become separate projects of their own given their size. Instead, the focus was kept on the Cloudflare Worker, which has been tested extensively, and the other components have been tested in-depth manually. Because the Cloudflare worker is so closely-coupled with the API, however, integration tests of the worker often cover the functionality of the API implementation as well.

5.1 Testing the Worker and improving the implementation

The Mocha JavaScript test framework running on Node.js has been used for the worker. Coupled with other libraries such as Chai for testing asynchronous code and evaluating assertions, the testing of the event handler of the worker has been split into several sections. These sections include tests for the caching and accessing of resources, creating videos and checking their status and other miscellaneous tests that would ensure that the worker does not interfere with the normal operations of the routes that it is proxying.

5.1.1 Unit Tests

For ease of testing and implementation, the worker has been split into two components: the layer of the worker describing the logic of the different routes, and the utility component containing the implementation of the numerous functions that the high-level layer references and uses.

A mock of the database has been implemented, storing some test tokens covering different scenarios. Mock utility functions have been created to perform requests matching different scenarios, such as makeResourceRequestWithWrongTokenType() or makeVideoStatusRequestWithWrongMetadata(). Test utilities have also been created to perform actions such as pushing random tokens onto the mock database, creating headers or logging. Mocks of the utility functions of the worker have also been created to reproduce the fetch API used to make requests. Careful consideration has been applied when structuring the testing to ensure that there are no circular dependencies among these testing utilities, mocks and files.

Among the tests implemented were tests ensuring the correct behaviour of the worker in the different token validity and cache hit scenarios described in the sections above. The worker needed make the appropriate requests and give the suitable responses depending on whether the token was present, was of the right type, with a valid expiration timestamp and found locally in the cache, tested for all the permutations of these possibilities. The worker was configured to raise errors from the utility functions which would be caught and handled at the top level of the call stack, in the high-level component of the worker, to organise and facilitate the implementation and testing of the handling of all these possibilities. Figure 8 shows some of the error messages that would be mapped to status codes and then handled by the worker to follow correct behaviour and produce an appropriate response to the client.

```
'Invalid request URL': 400,
'Missing video attachment data': 400,
'No auth token': 400,
'Wrong token type': 400,
'Not authorized': 401,
'Token expired': 401,
'Resource not found': 404,
'Could not get the padoq that this resource belongs to': 503,
'Could not get worker token': 503,
'Could not get user membership': 503,
'Service unavailable': 503,
'Subrequests to padoq server timed out': 503,
```

Figure 8: Worker error messages and codes

Tests were written to ensure that the worker would not exceed the memory and CPU time limits imposed by Cloudflare, together with tests that would ensure that the worker would clear its local runtime memory once it reached the maximum number of tokens and resources that it can store details about.

For these tests, a snapshot of the size of the heap used is recorded at the beginning of the test, and then from that, the amount of memory used at the end of the test is subtracted and checked to comply with the 128MB memory constraint. This way, the test only takes into consideration the memory used by the information stored about the tokens and resources, ignoring the memory impact that the testing framework and environment have. CPU time tests are also taken in the most stressful scenarios of the worker, such as searching for tokens and resources while the local cache that contains them is almost full, or during the local memory evictions that would happen regularly over the course of the worker's runtime. All these tests would require to take less than 9ms from the moment the handler is called to the moment the response is given.

With all the memory and CPU tests performed, the worker has been optimised to hold information about a maximum of 40,000 tokens and 4,000 resources at any time. Multiple instances of the worker script are deployed across multiple nodes, so these limits would not be global across the whole infrastructure. Other optimisations included representing tokens as a buffer of bytes rather than a string.

5.1.2 Integration Tests

While the unit tests ensure the functionality of isolated elements composing the worker, integration tests are the high level, final check for whether these components put together achieve the desired outcome of the implementation. Using the ”request” npm module, the integration tests make HTTP requests at the routes proxied by the worker, checking the responses, status codes, as well as the headers indicating whether the cache has been hit or not. The workflow of the integration tests obtains authentication, creates a post in a test padoq, creates an attachment for that post, uploads it, accesses it and checks to ensure that the cache was not hit, and then accesses it again ensuring that the cache has been hit for this second request.

5.1.3 Logging

To log the activity of the worker after it has been deployed to the edges of the Cloudflare CDN, the worker has been configured to append the logs that it gathers during the execution of the script when handling a request, and then send these to a logging server before finishing execution or when an error occurs. The logging server has been developed as a Node.js project as well and deployed as an application on the Heroku cloud hosting platform. The live log feed can be viewed via the Heroku command-line interface whenever needed.

5.2 Testing the implementation of the new Padoq API endpoints

Instances of the Padoq modules for authentication and serving resources can be run locally. While running locally, these servers can be accessed at different ports of the localhost. Using tools such as Postman, requests can be made and their responses analysed to test functionality. Padoq maintains and provides a series of Postman collections detailing the different endpoints of the Padoq API and the payloads that they consume. Figure 9 shows a request made from Postman used to test the successful creation of a post containing a video attachment.

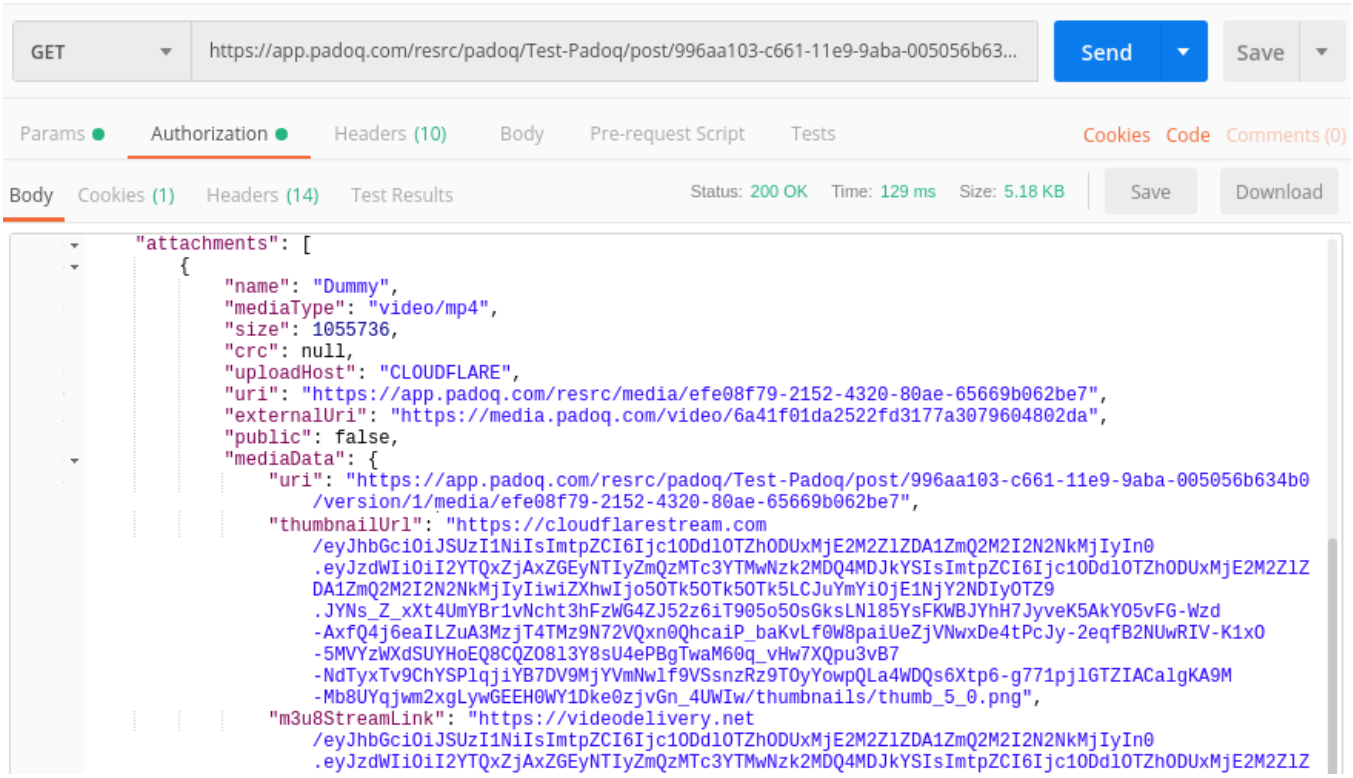

Figure 9: An HTTP GET request executed using Postman for a test post

The request retrieves a test post created in a test padoq, where a video attachment has been created and uploaded. In the JSON response, the details of the media data can be seen, including URLs that can be used to display the thumbnail of the video and stream the video.

Database playbooks are also available, which can be deployed using configuration management tools such as Ansible to create a local instance of the PostgreSQL database that the API uses.

These tools facilitate the manual testing of the new API functionality implemented, both locally and for the deployed versions on the live and development environments.

5.3 Testing the mobile client

To test the integration with the worker and the TUS protocol, the example Android app provided by Tus.io has been forked from GitHub and modified to use the Padoq authentication workflow and send the payloads required by the Padoq API to create video attachments in posts.

Using the TUS library, videos have been successfully uploaded, and the integration has been regularly tested over the course of development and after its deployment to the Android and iOS Padoq clients. To implement and test the new feature on the mobile client apps of Padoq, however, would be beyond the scope of this project, so it was left to the mobile developers while testing using the example app was sufficient for the back-end development.

6 Evaluation

6.1 Cloudflare analytics

Cloudflare provides analytics which give insight into the performance of the workers. Figure 10 shows the total amount of requests proxied by the worker over the course of the month of August 2019. We can observe a success rate of around 99.99%.

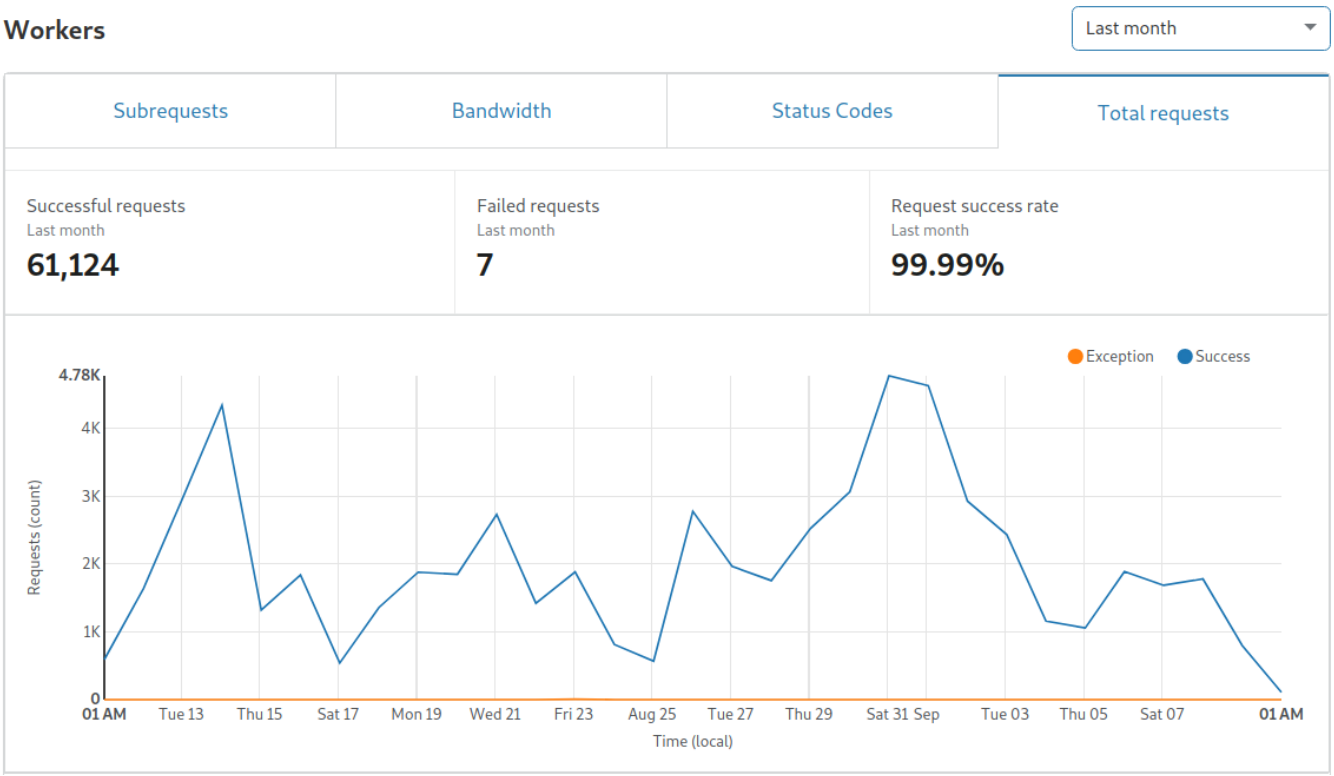

Figure 10: Requests proxied by the worker in August 2019

Figure 11 shows the status codes of the requests. A large proportion has been cached, reducing the load on the Padoq servers.

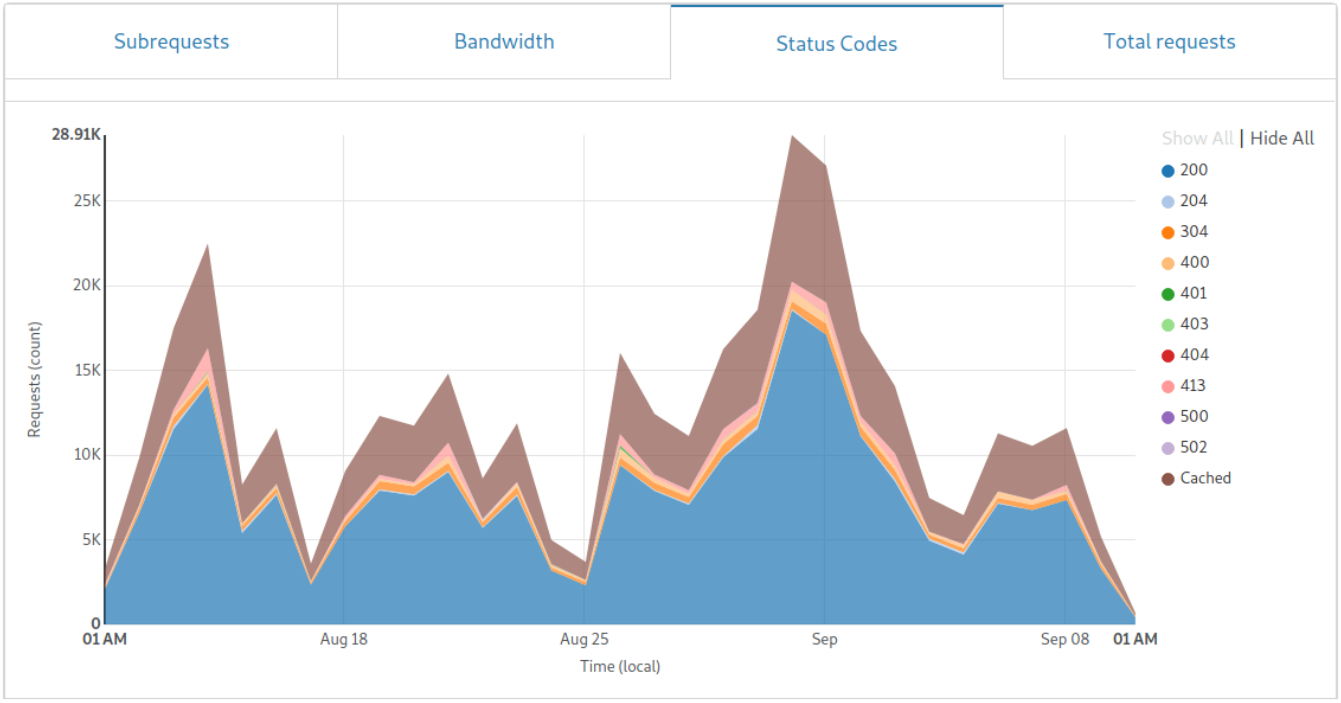

Figure 11: Status codes of requests proxied by the worker in August 2019

To evaluate the impact of the worker system on the server workload, however, analysis on bandwidth is required. Among the cached requests are the resource requests which consume large amounts of bandwidth. Figure 12 presents the amount of bandwidth cached by the worker, which underlines the effectiveness of the system.

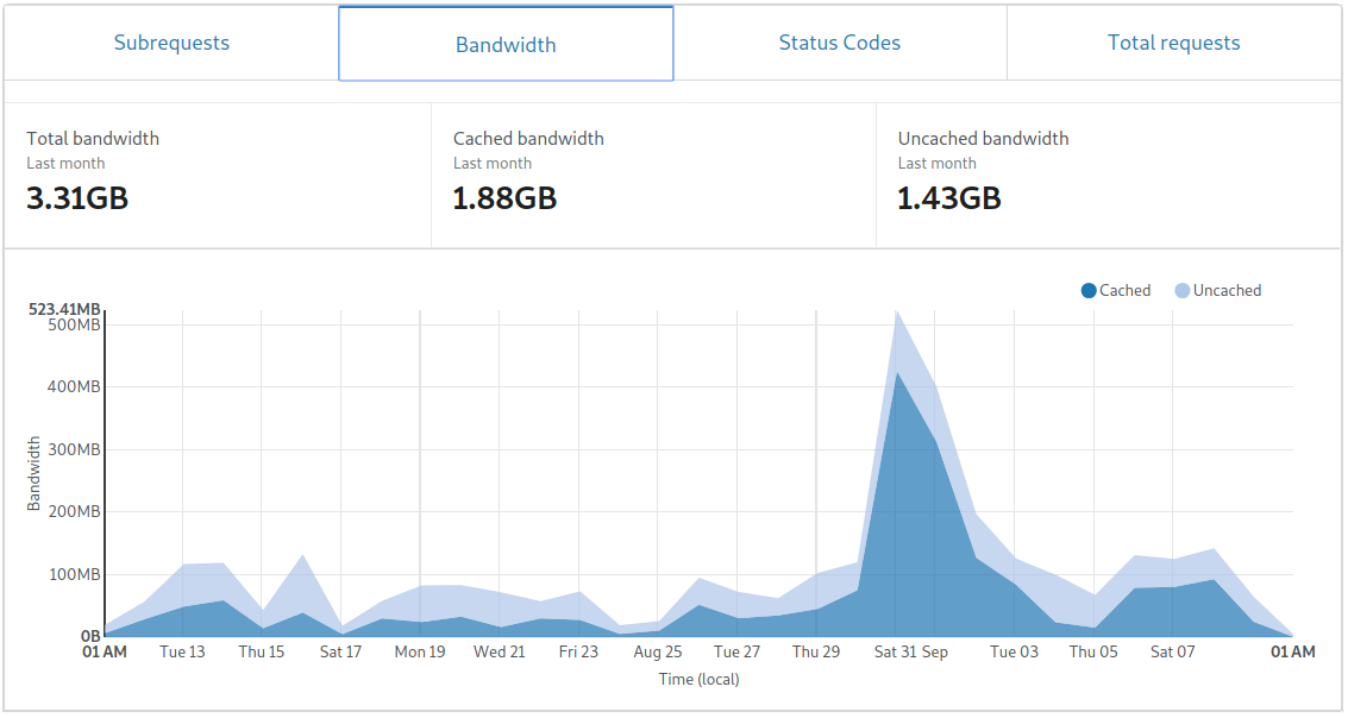

Figure 12: The bandwidth cached by the worker in August 2019

With the worker in place, 56.7% of bandwidth has been cached on average. At peak load, however, on the 31st of August, the worker cached 81.3% of the bandwidth.

6.2 General evaluation

Regular discussions and meetings with the manager and lead developer were required to check that the project is on track and that the design is on the right path.

The code of the worker was added as a repository among the others of the company on the account used to manage the versioning of the various projects of Padoq. Appropriate documentation has been added to the worker repository to enable the review by other colleagues.

For the addition of features to the Padoq API, every piece of implementation has been branched off the main version of the code and then merged back when finished via a code review request to the lead developer.

Finally, to check the finalised feature of video posts on the apps, manual testers have been regularly checking that the functionality meets the requirements.

The most significant piece of evaluation, however, is the positive user feedback that came with the introduction of the new video posts feature onto the live versions of the Android and iOS apps.

7 Reflective Evaluation and Conclusions

An integration with a highly performant and scalable CDN has been achieved over the course of this project, in the context of the back-end and mobile clients of the applications of Padoq, a start-up looking to revolutionise the way people organise and communicate. This integration enabled the offloading of computation and bandwidth from the servers of Padoq and the development of video posts using the streaming capabilities of the CDN. Different languages and tools have been used to develop and extend various components of the Padoq system while keeping a high standard for testing and continuous integration.

When choosing the CDN provider, the quality of the technologies, documentation and the various features such as video streaming have been taken into consideration. After the decision to integrate the Cloudflare CDN, choices of design, languages and tools have been dependent and flowing from the stack and configuration supported by the CDN.

The goals of improving the scalability and stability of the Padoq servers has been achieved, and the new video posts feature has been introduced as planned.

The success of the project can be seen from the fact that it is being used on the live version of the app available in the app stores on Android and iOS. Furthermore, Padoq provides a white-label version of the app that can be bought by other companies and customised with their branding to be used as an organisation tool for their communities (Padoq, 2019). The CDN and video streaming integration are being used across all those white-label applications sold to various companies as well.

The worker system integrated with the CDN over the course of this project is a powerful tool, and plans are being discussed to use it to offload the moderation and analysis of advertising on the apps at the edges of the CDN in the future.

A variety of techniques have been learned and tools have been used to improve numerous components across the systems of Padoq. In conclusion, the project has been a success, both regarding the quality of the product generated and the potential that it holds for future development.

References

Anicas, M. (2015) *How To Mitigate DDoS Attacks Against Your Website with Cloudflare*

Cloudflare (2019) *https://www.baeldung.com/java-dao-pattern [accessed September 2019]*

Cloudflare (2019) *https://developers.cloudflare.com/workers/about/limits [accessed September 2019]*

Cloudflare (2019) *https://developers.cloudflare.com/workers/about/how-it-works/ [accessed September 2019]*

Cloudflare (2019) *https://www.cloudflare.com/cdn [accessed September 2019]*

Cloudflare (2019) *https://www.cloudflare.com/products/Cloudflare-workers [accessed September 2019]*

Crunchbase (2019) *https://www.crunchbase.com/organization/padoq [accessed September 2019]*

Eclipse Vert.x (2019) *https://vertx.io [accessed September 2019]*

Node.js Foundation (2019) *https://nodejs.org/en [accessed September 2019]*

Padoq (2019) *https://padoq.com [accessed September 2019]*

Padoq (2019) *https://padoq.com/communities [accessed September 2019]*

Tus.io (2019) *https://tus.io [accessed September 2019]*

Web Hypertext Application Technology Working Group (2019) *https://html.spec.whatwg.org/multipage/workers.html [accessed September 2019]*