Technical University of Moldova
Inginerical department S.A.

Report

Practice in Production

University Practice Manager — Elena Gogoi
Company Practice Manager — Oleg Burlacu

Author: Terman Emil FAF161

Chisinau 2018

Contents

Introduction

We were given a task to a group of students. We had to implement a Television streaming manager, which would stream multiple channels to different users. We chose to implement it as a WebAPI in .NetCore 2.

The subject was very interesting since it's something that StartNet uses to stream television to their users. I was already familiar with .NetCore, but I still had difficulties.

The aim of this project was to make us more familiar with the real world requirements.

Objectives

- make a .NetCore Web API streaming server which would provide M3U8 files to the users;
- assign tokens to every user and manage their security;
- monitor ongoing requests;
- manage the chunking processes, so that the streaming would continue, even if the processes would get killed, or the internet is stopped for a few seconds;
- live updating of the streams when the config is changed;
- Live Streaming;
- TimeShift Streaming - indicate a number of milliseconds from live;
- FixedTime Streaming - indicate a time from where to start streaming;
- use the Dependency Injection pattern;

End Result

By introducing the following link examples in the VLC's network player, we should get:

- **Live Streaming**: *http://localhost:39986/api/Stream/JurnalTV/index.m3u8*
- **Time Shift of 10 seconds Streaming**: *http://localhost:39986/api/Stream/JurnalTV/index.m3u8?timeShiftMills=10000*
- **Fixed Time Streaming starting from the given time**: *http://localhost:39986/api/Stream/JurnalTV/index.m3u8?timeStr="2018-07-04T16:52:00+03:00"*

If the streamming processes get killed or they simply exit, due to some error, they should be preoperly restarted. No substantial glitches should be present in this case. The chunk indexing should continue and the disconinuities should be properly managed in the M3U8 output.

Theoretical analysis

HLS Streaming

There's been a major shift happening in the world of online video over the last decade. Adobe's Flash video technology, which had been the main method of delivering video via the internet, is rapidly declining. It's being replaced by video delivered using protocols like HLS streaming and played in HTML5 video players.

For users, this is a great change. HTML5 and HLS are open specifications, which means they can be modified and used by anyone for free. They're also safer, more reliable, and faster than what came before.

Content producers have more of a mixed bag. Sure, there are some major advantages to using these new technologies. However, there are also disadvantages, especially when it comes to replacing legacy systems and technologies with new standards that may not work the same across all platforms. Growing pains are inevitable.

This essay, which is aimed at both longtime broadcasters and newcomers to streaming media, focuses on HLS streaming. We've aimed to make it relevant for all kinds of streamers, whether you do live streaming of sports events, or you want to stream live video on your website. We'll cover basic definitions, discuss other streaming protocols, and answer the question posed in the title of this essay: what is HLS streaming and when should you use it? Let's get right into it.

What is HLS?

HLS stands for HTTP Live Streaming. Essentially, HLS is a media streaming protocol that is used for delivering visual and audio media over the internet.

The HLS streaming protocol works by chopping MP4 video content into short, 10 second chunks. These short clips are delivered via HTTP, which makes HLS compatible with a wide range of devices and firewalls. Latency for HLS live streams compliant with the specification tends to be in the 15-30 second range.

When it comes to quality, HLS streaming shines. On the server side, content creators frequently have the option to encode the same live stream at multiple quality settings. Players can then dynamically request the best option that is available to them given their bandwidth at any given moment. From chunk to chunk, the data quality can differ.

One moment, you might be sending full high-definition video. Moments later, a mobile user may encounter a "dead zone" where their quality of service declines. The player can detect this decline in bandwidth and begin delivering lower-quality movie chunks at this time. This reduces buffering, stuttering, and other problems.

M3U8 Files

A file with the M3U8 file extension is a UTF-8 Encoded Audio Playlist file. They are plain text files that can be used by both audio and video players to describe where media files are located.

For example, one M3U8 file may give you references to online files for an internet radio station. Another might be created on your computer to build a playlist for your own personal music or a series of videos.

An M3U8 file can use absolute paths, relative paths, and URLs to refer to specific media files and/or entire folders of media files. Other text information in an M3U8 file may be comments that describe the contents.

A similar format, M3U, can use UTF-8 character encoding, too, but may include other character encodings as well. Therefore, the .M3U8 file extension is used to show that the file is in fact using UTF-8 character encoding.

FFMPEG Chunking Tool[1]

FFmpeg is a free software project, the product of which is a vast software suite of libraries and programs for handling video, audio, and other multimedia files and streams. At its core is the FFmpeg program itself, designed for command-line-based processing of video and audio files, widely used for format transcoding, basic editing (trimming and concatenation), video scaling, video post-production effects, and standards compliance (SMPTE, ITU). FFmpeg includes libavcodec, an audio/video codec library used by many commercial and free software products, libavformat (Lavf), an audio/video container mux and demux library, and the core ffmpeg command line program for transcoding multimedia files. FFmpeg is published under the GNU Lesser General Public License 2.1+ or GNU General Public License 2+ (depending on which options are enabled).

The name of the project is inspired by the MPEG video standards group, together with "FF" for "fast forward". The logo uses a zigzag pattern that shows how MPEG video codecs handle entropy encoding.

FFmpeg is part of the workflow of hundreds of other software projects, and its libraries are a core part of software media players such as VLC, and has been included in core processing for YouTube and the iTunes inventory of files. Codecs for the encoding and/or decoding of most of all known audio and video file formats is included, making it highly useful for the transcoding of common and uncommon media files into a single common format.

Asp.Net Core[2]

ASP.NET Core is a free and open-source web framework, and the next generation of ASP.NET, developed by Microsoft and the community. It is a modular framework that runs on both the full .NET Framework, on Windows, and the cross-platform .NET Core.

The framework is a complete rewrite that unites the previously separate ASP.NET MVC and ASP.NET Web API into a single programming model.

Despite being a new framework, built on a new web stack, it does have a high degree of concept compatibility with ASP.NET MVC. ASP.NET Core applications supports side by side versioning in which different applications, running on the same machine, can target different versions of ASP.NET Core. This is not possible with previous versions of ASP.NET.

Dependency Injection

In software engineering, dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

This fundamental requirement means that using values (services) produced within the class from new or static methods is prohibited. The client should accept values passed in from outside. This allows the client to make acquiring dependencies someone else's problem.

The intent behind dependency injection is to decouple objects to the extent that no client code has to be changed simply because an object it depends on needs to be changed to a different one.

Dependency injection is one form of the broader technique of inversion of control. As with other forms of inversion of control, dependency injection supports the dependency inversion principle. The client delegates the responsibility of providing its dependencies to external code (the injector). The client is not allowed to call the injector code; it is the injecting code that constructs the services and calls the client to inject them. This means the client code does not need to know about the injecting code, how to construct the services or even which actual services it is using; the client only needs to know about the intrinsic interfaces of the services because these define how the client may use the services. This separates the responsibilities of use and construction.

There are three common means for a client to accept a dependency injection: setter-, interface- and constructor-based injection. Setter and constructor injection differ mainly by when they can be used. Interface injection differs in that the dependency is given a chance to control its own injection. Each requires that separate construction code (the injector) takes responsibility for introducing a client and its dependencies to each other.

Processes

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

While a computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU (core) executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow high responsiveness for interactive user applications. In time-sharing systems, context switches are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This seeming execution of multiple processes simultaneously is called concurrency.

For security and reliability, most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

Signals

Signals are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems. A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred. Signals originated in 1970s Bell Labs Unix and have been more recently specified in the POSIX standard.

When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.

Embedded programs may find signals useful for interprocess communications, as the computational and memory footprint for signals is small.

Signals are similar to interrupts, the difference being that interrupts are mediated by the processor and handled by the kernel while signals are mediated by the kernel (possibly via system calls) and handled by processes. The kernel may pass an interrupt as a signal to the process that caused it (typical examples are SIGSEGV, SIGBUS, SIGILL and SIGFPE).

In-memory Database[3]

An in-memory database (IMDB, also main memory database system or MMDB or memory resident database) is a database management system that primarily relies on main memory for computer data storage. It is contrasted with database management systems that employ a disk storage mechanism. In-memory databases are faster than disk-optimized databases because disk access is slower than memory access, the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk.

Applications where response time is critical, such as those running telecommunications network equipment and mobile advertising networks, often use main-memory databases. IMDBs have gained a lot of traction, especially in the data analytics space, starting in the mid-2000s – mainly due to multi-core processors that can address large memory and due to less expensive RAM.

A potential technical hurdle with in-memory data storage is the volatility of RAM. Specifically in the event of a power loss, intentional or otherwise, data stored in volatile RAM is lost. With the introduction of non-volatile random access memory technology,[when?] in-memory databases will be able to run at full speed and maintain data in the event of power failure.

Redis[3]

Redis is an open-source in-memory database project implementing a distributed, in-memory key-value store with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, hyperloglogs, bitmaps and spatial indexes. The project is mainly developed by Salvatore Sanfilippo and is currently sponsored by Redis Labs.

Redis typically holds the whole dataset in memory. Versions up to 2.4 could be configured to use what they refer to as virtual memory[22] in which some of the dataset is stored on disk, but this feature is deprecated. Persistence is now achieved in two different ways: one is called snapshotting, and is a semi-persistent durability mode where the dataset is asynchronously transferred from memory to disk from time to time, written in RDB dump format. Since version 1.1 the safer alternative is AOF, an append-only file (a journal) that is written as operations modifying the dataset in memory are processed. Redis is able to rewrite the append-only file in the background in order to avoid an indefinite growth of the journal.

By default, Redis writes data to a file system at least every 2 seconds, with more or less robust options available if needed. In the case of a complete system failure on default settings, only a few seconds of data would be lost.

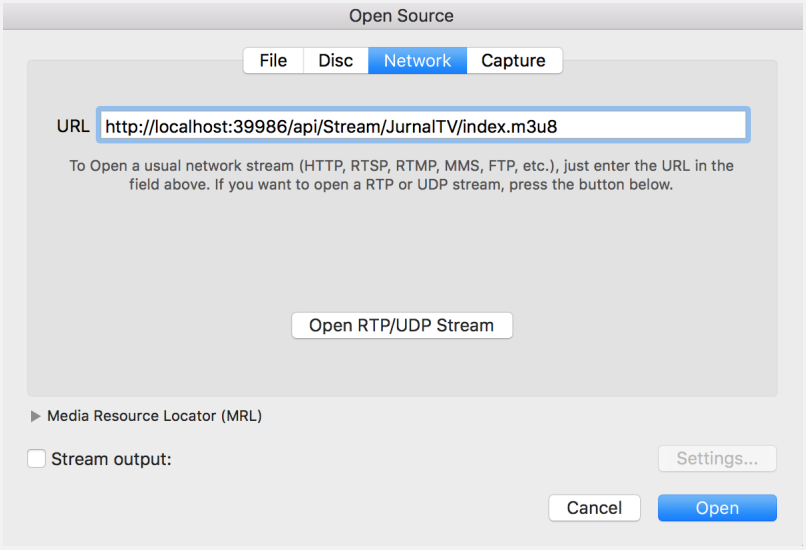
Demonstration

Img 1: Monitored API result

In *Figure 1* is shown the raw output of the M3U8 file of the Streaming API. It can be seen in this case, that the requested channel is *RadioMD*.

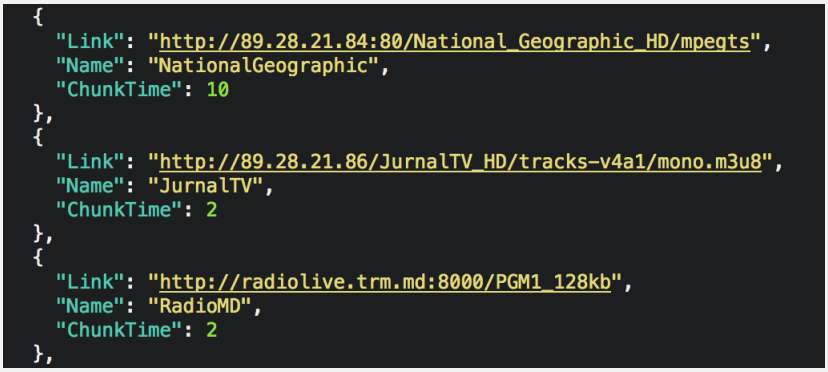
The first time the user accesses this API, he needs to provide a Channel name and a registration token to determine if it's a paid user or a guest. In the above example, we can see that it's a paid user. Then, he is redirected to another link, with a different token. This token serves as a key to the user's session.

To remember the user state (the last requested chunk index, channel, list size, etc), I use a **Redis in memory Database**. Why not simply use HTTP Sessions? Well, because not everyone supports cookies. For example, **VLC** doesn't.



Img 2: Using the API in VLC

In *Figure 2* is demonstrated how the API can be used. For testing purposes, we have used only VLC. But it can be used for a large varaity of services.



Img 3: Channel configurations

In *Figure 3* is represented how the Streaming Sources are configured. For each source, a source link, a chunk time in seconds and a channel Name is provided. The incomming input is chunked in pieces of ChunkTime seconds. For future, we will reformat the input in different chunks of different resolutions for different Bandwidths, so that the user gets a clean and smooth streamming.

Implementation

The whole project relies on Dependency Injection.

We have 2 Singletons:

– StreamingProcManager;

– StreamingSourceUpdater;

And we have one main controller: **StreamController**, which is basically the part that gives M3U8 files.

The project is built with n-tier Architecture: Web Request Manager and FFMPEG Logic Layer. We used this approach, to isolate the logic and make it possible to move the project on multiple server. Still, we have a long way to go.

On Startup, the Streaming Processes are run and the .ts Chunks start getting generated on the machine, in the directory specified in the appsettings.

In appsettings, a **Hash SALT** is also specified. If it's an Environment Variable Name, then the value is loaded from the environment. This SALT is used at **ITokenBroker** which initializes the tokens whith a Hash function. In my case, I use **SHA256**. It's used, because theoretically, it possible to generate the same hash, if a Hacker tries.

Conclusion

At this practice, we have implemented a Television Streaming API, which uses ASP.NET Core for the server part and FFMPEG for chunking. Compared to the real life implementations, we still have a long way to go. This project is not something we can implement in month, so I will continue working on it. Our initial aim was to add Comercial management by the end of this Practice. But, there was too much we had to learn and too little time.

It was a great opportunity to learn from Proffessionals. Mr. Burlacu gave us clear indications and very good advice. He was very busy with his own work, so we couldn't meet very often to discuss the project. We only had a very vague idea of what needs to be done and what to use. I think this made us explore more and force to come up with ideas of our own. From time to time, he'd give us some keywords which again, we would need to explore and implement. In this way, I have learned a vast number of tools and how to use them. I had to try different implementations and learn by try and error which one works and which one is better.

I had the opportunity to learn about a very powerful tool. Something that not only StartNet uses, but almost any video hub, like YouTube or Spotify. Besides learning how to implement these things, I have also learned how the video streaming apps work in the real life: what it needs, what tasks can be distributed on different servers and how to manage them, what is the approximate perfomare, what containers are better, etc. All of these together, allow me to have a broader look at a project, which I think is a stepping stone in becomming a world class Engineer.

Bibliography

1. FFMPEG *https://ffmpeg.org/ffmpeg-formats.html*
2. Microsoft Docs *https://docs.microsoft.com/*
3. Redis in-memory database *https://github.com/antirez/redis*