SDCC project: Soundtime

A simple (and serverless) music sharing service

Eduard Manta, student,
University of Rome Tor Vergata

***Abstract*—This document describes the Soundtime music sharing service, illustrating its core architecture and implementation, explaining the various design decisions taken during its creation along with the limitations of the system in its current state.**

***Keywords*—music, serverless, Amazon Web Services.**

I. Introduction

Soundtime, as declared in the subtitle, is “a simple [...] music sharing service” designed in a serverless fashion and built making use of the Amazon Web Services; more specifically the pool of services used, directly by the application or indirectly during its creation and deployment, is comprised of: Lambda, S3, API Gateway, DynamoDB, CloudFront and Route53. The application enables users to share audio files by uploading them to the service, letting other users retrieve the file to listen or download it via a short ID associated to it; files that have not been accessed for longer than 14 days are automatically deleted.

II. Architecture description

Using the classic 4+1 Model, introduced by Philippe Kruchten in 1995, an application can be described from various points of view, each of which is concerned with conveying a different level of information regarding the system, its purpose and its operation. Being designed more than 10 years before the idea of "The Cloud" was being put into reality by AWS, the 4+1 Model was built as a way to design and document monolithic applications and thus is not well suited to provide a concise and accurate description of a serverless application as many challenges and aspects that this model focuses on (e.g. scalability, availability, distribution, physical interconnection between the hardware resources) are trivial to achieve or completely eliminated in a cloud computing setting. Even taking this into account, the model is a valid tool — even if in some of the views is an unnecessary and verbose one at that — for describing the architecture of an application in the modern cloud era.

A. Use case view

As it could be easily deduced from the system's description, the functionality offered to the User is rather little and thus the list of use cases is rather short as well: the User should be able to Upload, Listen and, should she/he want it, Download a file as illustrated by the diagram in Fig. 1.

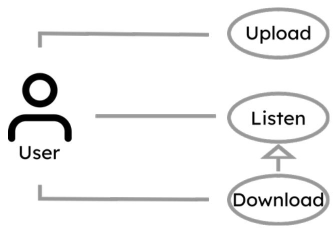

Fig. 1. Use case diagram for the application.

B. Logical view

A bird's eye view of the system's layout and the interactions between the services that are used by it can be seen in Fig. 2: the User interacts with the system via a web Client that in turn makes use of the API exposed using API Gateway to trigger the Lambda functions that implement the business logic of the application; the system also uses DynamoDB as a data store for metadata regarding the uploaded files and S3 for file archiving. A noteworthy interaction that is represented in the diagram is the direct one between the client and S3: this is the result of limitations of Lambda and API Gateway that led to design decisions that will be discussed below.

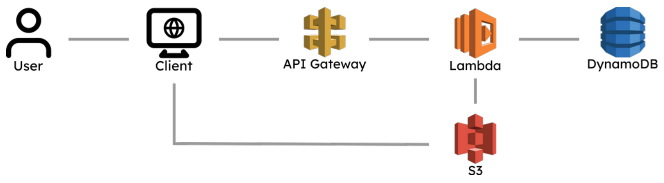

Fig. 2. A simplified Communication diagram representing the system as a whole; lines between the icons represent direct interactions between the components at their ends.

The upload functionality offered by the system is presented in Fig. 3 under the form of a Sequence diagram detailing the order of the operations involved and the services that are used. In order to upload a file, the web Client contacts the API Gateway endpoint encountering a limitation on the maximum size for a request body that the endpoint can process: 10 MB. While 10 MB might be a reasonable limit in many cases, it is also dangerously close to the size of a high quality MP3 encode of a 4 minute song and about a third of the size of a song of the same length encoded losslessly in FLAC; if this wasn't enough the maximum size of the input event trigger that Lambda accepts is 6 MB, making uploads either unsafe (by using the endpoint as proxy to access a public S3 bucket), difficult (by designing an API and a set of Lambda functions that would enable chunked uploads) or impossible for files bigger than 6 MB. An alternate solution to this problem is authorizing direct uploads to S3 via a pre-signed link valid for a limited amount of time allowing for uploads of files as big as 5 GB in a single request; the signed links are generated using a Lambda function invoked by the API endpoint only after the Client declares the file's name, size, MIME type and MD5 checksum. Once the Client receives the signed link, it uploads the file to the bucket; if the file type and MD5 checksum do not match the file is not added to the bucket and the Client is notified. The Lambda function also stores some basic information about the file like its name, size, upload date and encoding inside a DynamoDB table along with a short ID that will be used to retrieve the file and a short time-to-live (TTL) that will have the entry removed if the upload is not completed within its lifetime.

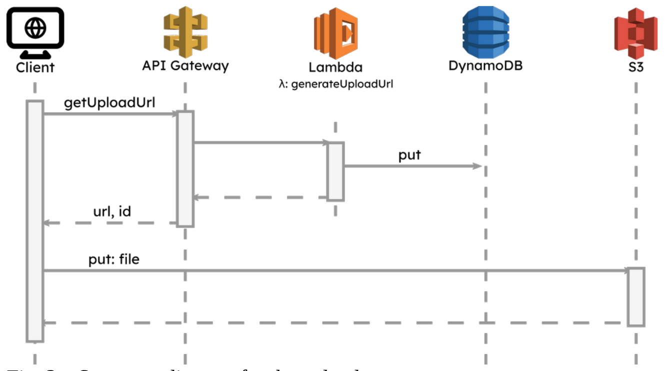

Fig. 3. Sequence diagram for the upload process.

Once the file is uploaded to S3, another Lambda function, set to be triggered upon 'put' events in the bucket, processes the audio file by extracting, if available, the metadata integrated in it (artist, title and cover art) and saving them into the DynamoDB table extending the entry’s TTL, while also encoding a copy of the file to a lower quality, later offered to Users as a preview of the song before downloading it, as shown in Fig. 4; the encoding step, initially thought to be dependant on the AWS Elastic Transcoder service, is handled entirely within Lambda by using a static build of FFMPEG as mean for converting the file, offering significant cost savings and a better free tier plan in terms of dollars per minute of encoded audio over the Elastic Transcoder.

Requesting information about the file is done by sending a request containing the ID of the file to an API endpoint, triggering a Lambda function that accesses the DynamoDB table to retrieve metadata and update its time-to-live; the function also attaches to the response a signed link to the low quality version of the file that the Client uses inside an audio player allowing the User to preview the track before downloading. This is shown by the Sequence diagram in Fig. 5.

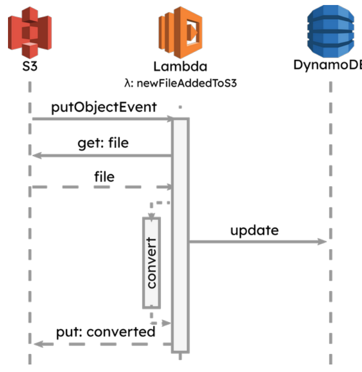

Fig. 4. Sequence diagram illustrating the processing of the file once uploaded to the S3 bucket.

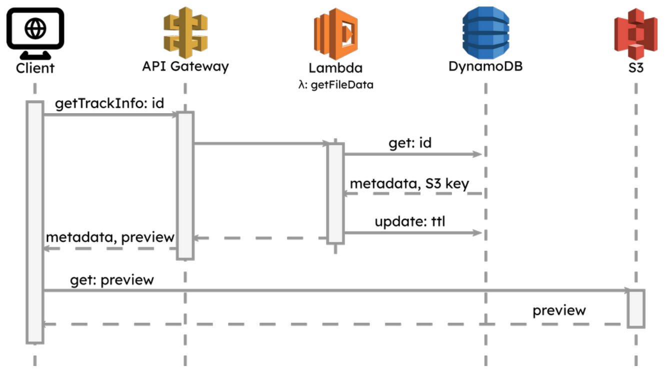

Fig. 5. Sequence diagram representing how a file’s metadata and preview is fetched when requested.

File download, as shown in Fig. 6, has a similar flow of events and also starts with the Client sending a request containing the ID to an API endpoint, again generating a signed link, this time to the original file, and returning it to the Client that will then use it to start the file transfer; the Lambda function generating the link also accesses the DB table in order to retrieve the original file name and set it in the signed link's headers as otherwise the downloaded file would have a different name and extension compared to the uploaded one.

As mentioned before, files uploaded to the service have a limited time-to-live tied to the last request received for them: once 14 days go by without a file being requested the DB entry related to it is marked for deletion and once deleted inserted into a DynamoDB Stream that triggers a Lambda function that the file it refers to; it is important to note that this method of deletion is not instantaneous as there is a delay between the entry being marked for deletion and being actually deleted from the table together with another (tunable) delay between its deletion and insertion in the stream and the Lambda function being invoked, in order not to launch hundreds of separate Lambda instances while the stream is being populated. For these reasons, along with the choice to allow direct uploads to the S3 bucket, the lifecycle of a file inside the system is slightly more complicated than "uploaded, active, deleted", allowing for files to be "saved" by a request while being marked for deletion, thus extending their lifetime to another 14 days, or entries in the DB table referencing files that have now yet been uploaded to the S3 bucket. This lifecycle is illustrated in Fig. 7 under the form of a state diagram.

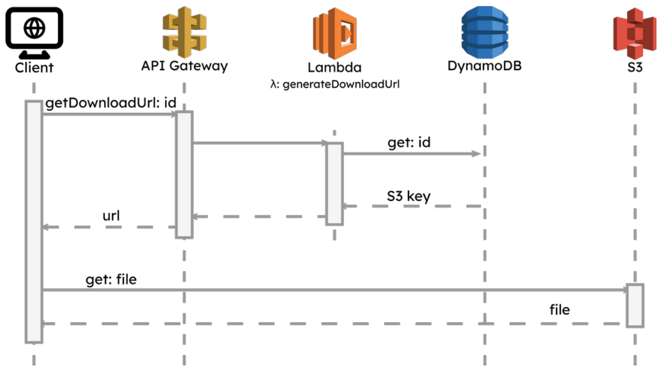

Fig. 6. Sequence diagram for the download process.

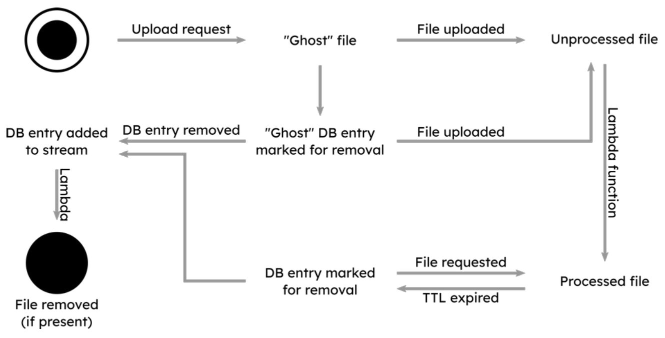

Fig. 7. State diagram representing the lifecycle of a file inside the system.

C. *Process view*

The Process view has the job of analyzing and responding to non-functional requirements such as constraints in the operating environment the system has to abide by and challenges it might face during operation, designing the control flow, the messages exchanged between the components of the application and coordinating concurrency.

Since the application is entirely serverless there is little to be discussed in this view that has not yet been covered or can immediately be deduced from the Sequence diagrams, as concurrency, distribution, availability and communication between the services used by the application are completely managed by AWS and thus are subject to the provider's resource availability and QoS policies.

D. *Implementation view*

The Implementation view is tasked with organizing and structuring the codebase in a coherent manner, documenting the scope of the various packages/libraries/subsystems that compose the application, their relationships of interdependence, usage of external libraries, providing a tool for work organization and planning, evaluation of the costs of development. Again, as important as documenting these aspects for a monolithic application is, for a serverless application — especially a simple one like Soundtime — such documentation serves arguably little purpose given the natural separation of the codebase into functions that are most likely to never have to interact directly with each other, resulting into a Package diagram made of a series of boxes with no interconnection whatsoever.

Generalizing the concept of documenting interdependence to not only code but cloud services and resources provided by them, a more fitting diagram illustrating the relationships of access between the various services/resources — similarly to Fig. 2, but at a higher level of detail — can be formed to provide an accurate description of how the implementation of the system has been carried out, as shown in Fig. 8.

E. *Deployment view*

This view, dedicated to taking into account the physical challenges of the non-functional requirements such as scalability and availability by designing the infrastructure the system will operate in and plans for recovery in case of hardware failures, might appear futile from the cloud based IaaS perspective and even more from the PaaS and FaaS points of view, but might be an important asset in the creation of complex hybrid or multi cloud systems by allowing to take a better look at the challenges involved in the interoperation between services offered by different vendors and by giving more insight into the planning for allocations of resources — albeit in the case of cloud systems not physical ones — in the event of outages, traffic spikes or other forms of emergency.

For simple, single-vendor, serverless applications on the other hand, a deployment diagram might not be absolutely necessary but can still provide a few insights on how end users interact with the system; the deployment diagram in Fig. 9 takes care of illustrating this.

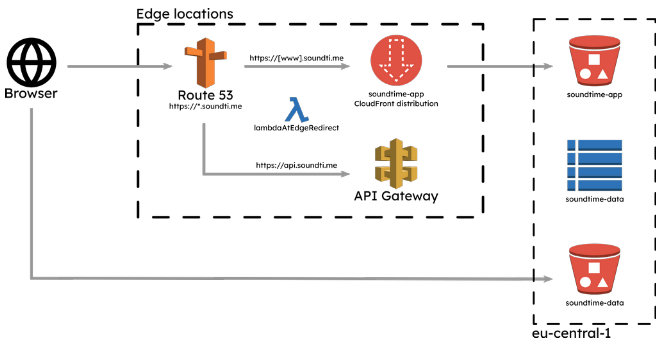

Fig. 9. Deployment diagram for the system

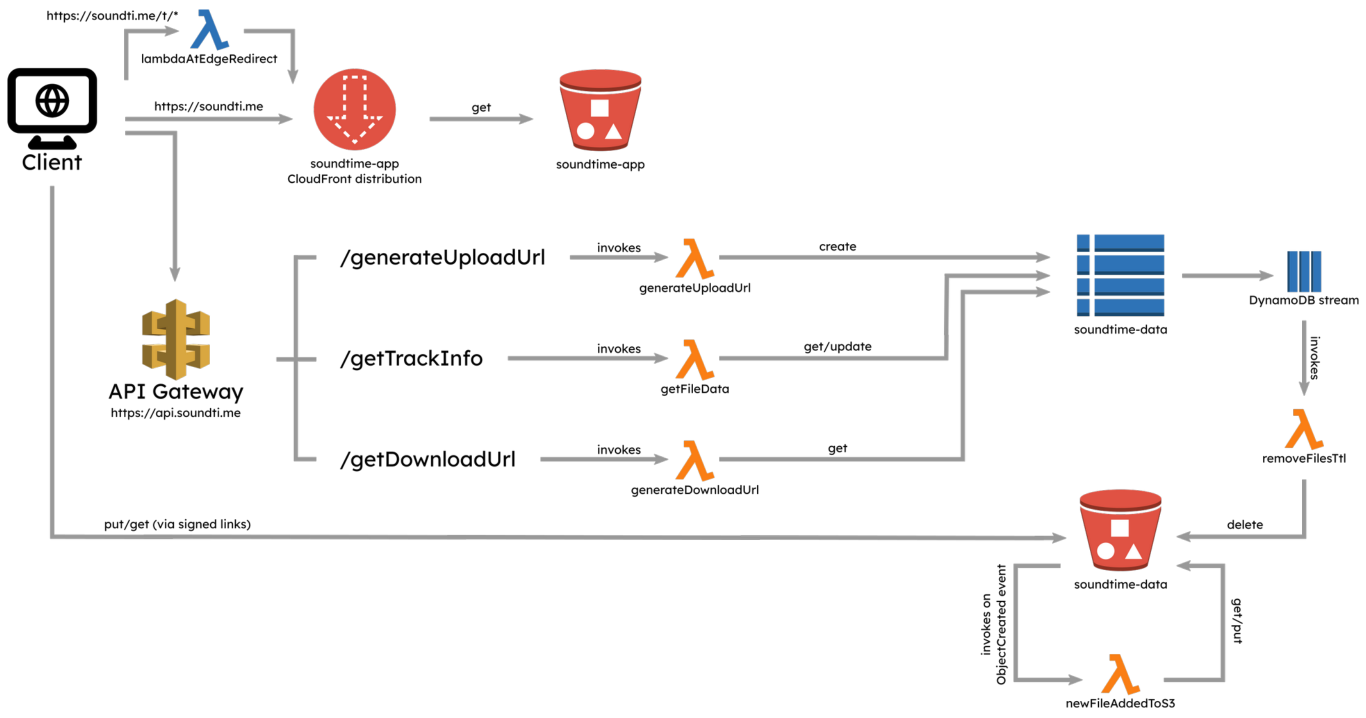

Fig. 8. Diagram showing the relationships between resources and services

III. Implementation details

A. *Client*

The client is a web Single Page Application (SPA) developed using the AngularJS framework and publicly available at https://soundti.me or https://www.soundti.me; being a SPA it relies on loading the index.html file upon the first access to the website no matter the sub-path in the URL, routing the navigation internally without having to load other static files, interacting with the API instead to fetch and display content.

The application allows for the upload, preview and download of the files that, once uploaded, can be reached via URLs of the form https://soundti.me/t/[6+ characters code]; upon requesting such a URL the application launches a request to the API for the metadata related to the file if available, otherwise if the file does not exist it displays an error stating this. The choice for the format of the URLs came from a user experience perspective and a simple consideration: a short URL is easier to remember and share. The minimum of 6 characters was chosen as a lower bound as it represents a sweet spot between being short and generating low collision probability: using the generalized version of the "birthday paradox problem", the chance of collision between random combinations of 6 alphanumeric characters (using both lower and upper case letters) reaches 1% when about 33800 of these combinations are generated.

A few sample tracks that can be previewed and downloaded through the service are available at

- https://soundti.me/t/zIsfBy
- https://soundti.me/t/Vfna1J
- https://soundti.me/t/J2Ovl2
- https://soundti.me/t/1djWyO

B. *S3*

S3 is used as a way to provide storage for both the web application and its data, in the aptly named soundtime-app and soundtime-data buckets, both located in the eu-central-1 region.

Uploads/downloads are done directly to/from S3 by generating time limited pre-signed URLs that can be used to execute a PUT/GET HTTP request to/from the soundtime-data bucket; files stored within the bucket use a UUIDv4 as their object key. This bucket is also used for storing the low quality previews of each track that can as well be retrieved via pre-signed links; the previews are stored with the same key of the original file with the added prefix "previews/".

C. *CloudFront*

The Content Delivery Network service provided by AWS is used as a mean to provide access to the web client; the CDN uses as origin the soundtime-app S3 bucket that contains the static resources needed by the website, namely its root index.html, the Javascript bundle of its logic index.js and a few image assets like the favicons.

Since the SPA relies on the root index.html file being served in order to execute routing and the logic for the web page, the CloudFront distribution has been set up to redirect any requests that return a 403 or 404 error in the soundtime-app bucket to the index.html file. This redirection approach created a problem: since music uploaded to service is shared through links like https://soundti.me/t/[...] any request of the page related to a track would force CloudFront to generate a get request towards the S3 bucket that would in turn return a 404 error code causing the initial request to be eventually redirected, thus slowing down delivery to the user by generating unnecessary traffic towards the S3 bucket and eroding the 20000 get requests available in the S3 free tier (during testing of the web application more than 16000 get requests were generated in less than two weeks by the simple act of repeatedly requesting tracks while testing cache behavior). To combat this issue a simple Lambda@Edge function was used to redirect all requests whose path matched "/t/*" towards the index.html that would then be immediately served from CloudFront's cache when available.

D. *Route 53*

A Route 53 hosted zone is responsible for routing traffic directed to the soundti.me domain towards the CloudFront distribution (when requesting the root domain or its www. soundti.me sub-domain) or the Gateway API endpoint (available at the api.soundti.me).

E. *API Gateway*

The API is deployed in two stages: the edge-distributed, cache-enabled production stage reachable at api.soundti.me and the beta stage used during development and testing. The API for the Soundtime service is composed of just three resources, namely /getUploadUrl, /getTrackInfo and /getDownloadUrl; the names of these resources as well as the route they can be reached at is editable in the app/config/api.constant.js file.

For brevity reasons, more information about the API, the expected request parameters and format of the responses can be found in the API Gateway section of the README.md file distributed with the source code.

F. *DynamoDB*

All metadata about the files uploaded to the service is kept in a table named soundtime-data deployed in the eu-central-1 region. Each item in the table uses as key the 6+ character ID that is also used in the URLs of the service while a second index, based on the S3 key of each file, is kept to allow for retrieval of the entries when processing events triggered by S3.

All items in the table are in possession of a numeric attribute named ttl that, as the name suggests, represents the time-to-live (TTL) of the item inside the table in the form of UNIX epoch timestamp representing the last second the item will be valid; once the TTL of an item is up, within a few minutes, it will be removed from the table automatically.

Other information kept within the table is the original filename and format, the size in bytes of the file, upload date, the title, artist, album and cover art (as a Base64 encode image) if available in the original file.

G. *Lambda*

All functions except lambdaAtEdgeRedirect are executed in the eu-central-1 region and use Python 3.7 as runtime.

The three functions invoked by the API, namely generateUploadUrl, getFileData and generateDownloadUrl are set up to use up to 128 MB of memory and the timeout of 1 second (while all three functions instead return within 250 ms, 1 second is lowest timeout allowed).

On the other hand, the most demanding function, newFileAddedToS3, tasked with extracting metadata from the uploaded file and converting it via FFMPEG to a 128 kb/s Ogg Vorbis file for preview reasons, is configured to run using 2048 MB of memory — barely any of this memory is used, but as Lambda allocates CPU power proportionally to memory such a setting was necessary to encode files in a reasonable amount of time — and with a timeout of 1 minute, allowing to encode files long up to about 1 hour.

Another function with a long timeout, 1 minute and 30 seconds, is removeFilesTtl, that analyzes a DynamoDB Stream looking for items removed from the soundtime-data table because of an exhaustion of their TTL. Since the stream is populated at each operation on the table and to avoid processing just a few items each time the function is called, the stream is set up to wait for a time frame of as long as 5 minutes for up to 200 records in order to fill a pool of events that will be sent to the function; once either 200 records are collected or 5 minutes have passed the function is invoked to analyze the stream, filtering the operations that are not deletions for TTL and then removing the files from the S3 bucket associated to the entries removed from the DynamoDB table.

IV. Development

The development process did not make use of any serverless application deployment framework because of the steeper learning curve compared to using the AWS Management Console; despite this, AWS allows for exporting SAM and Swagger templates for easier deployment of, respectively, Lambda functions and API — the S3 buckets, CloudFront distribution and DynamoDB table will still have to be configured manually though. The bulk of the development regarding Lambda functions was done using the web editor and, occasionally, using a text editor and the AWS command line tool locally.

The web client was developed locally using npm for package management and workflow automation and Webpack for bundling of the distributable application.