Web app visualizing the history of listening to music on Spotify

By: Jacek Czupyt, Piotr Brysiak, Paweł Koźmiński, Marcos Navarro Juan

App Description

The main goal of this project is to implement and deploy a web application offering visualization of users' historical Spotify data. Spotify is an online streaming service offering its clients access to the history of listening. Nevertheless, this access is not automatic and requires a request on the Spotify webpage. The data can be downloaded in .json format with information about what, when and how long was played on the account. We are going to deploy and host the app using Amazon Web Services to provide high availability, good performance and effective cost management.
The app will be available to both anonymous and registered users. All users will be able to visualize and analyze their history, but only logged in ones will be allowed to store, manage and utilize many versions of the files.
The system will offer high availability thanks to the use of the AWS products. The module responsible for managing user accounts will be implemented on the basis of Amazon Cognito.
The first version of the application was implemented during the Data Visualization course. Now it can be accessed via this link. The historical data may be requested through Spotify account settings.

Functional Requirements

No.	Requirement	Priority
1.	Public users are allowed to register an account	Must
2.	Public users are able to sign in to the account	Must
3.	Logged in users can upload and store their historical JSON files on the service	Must
4.	Users will be able to select a JSON file in order to visualize its content	Must
5.	Logged in users can delete their file from the system	Must
6.	Logged in users can change the password of their accounts	Should

Non-Functional Requirements

1. Availability
    a. The app should be hosted in two different availability zones to secure disaster one
2. Cost-effectiveness
    a. The number of running instances should be proportional to the current traffic, to avoid unnecessary costs
3. Security
    a. Every user has access only to his own files
    b. No password leakages
    c. No possibility for public users to access the app via SSH
4. Disaster recovery scenario
    a. Recovery time objective - 72 hours
    b. Recovery point objective - 72 hours
5. Scalability
    a. The system will be scalable, ie. the number of running instances (t2.micro) will be reduced to 2 in case of the minimal demand and extended to 4 in case of high traffic. Desired number of running instances will be 2.

Architecture

For the final project, we created our VPC with two availability zones. The VPC has two private and public subnets to ensure high availability of the system, in case of failure one of the availability zones.

The Bastion host is created in the public subnet to allow administrators access to the autoscaling group instances and database in private subnets, in case of system failure or maintenance.

An internet-facing application load balancer is created to balance the traffic loads between the availability zones. It is mapped to the two public subnets and uses the Load Balancer security group, which allows for HTTP connections from any address. It uses a new simple target group, which forwards all HTTP traffic on port 3838.

A new autoscaling group is created using the launch template created in CloudFormation, which contains a bash script that installs the necessary R libraries, downloads our shiny application files and runs them on the instance. The created instances use the application security group, which accepts tcp traffic on port 3838 from the security group used by the load balancer, and can create PostgreSQL connections with the group used by the database. The autoscaling group is created in the two private subnets of VPC and is attached to the previously created load balancer. It has a minimum and desired number of instances equal to 2, and a maximum equal to 4.

A database subnet group is created on the private subnets of the VPC. A PostgreSQL RDS instance is then created on it, to host the database required by the application. It is a small db.t3.micro class, uses MultiAZ to ensure high availability, and uses the database security group, which allows for database connections from the autoscaling security group. The database performs backup once a day.

In our system architecture, we added health checks to ensure that the system is working correctly. In case of an error in the database or autoscaling group, the administrator will be notified by email. The database instance uses an RDS event subscription that checks if the database changes its status to failure or deletion. If it happens the Amazon Simple Notification Service sends a notification to the administrator email. The autoscaling group uses the Route53 health checks. The Route53 health check sends an HTTP request to the LoadBalancer DNS every 30 seconds. If it gets a 2xx response it accepts a health check. If Route53 got 120 errors in a row, it means the website has not been working for an hour. In that case, the administrator will be notified by an email about the problem with the web app. This amount of consecutive errors was chosen, as a new instance created by the autoscaling group generally takes up to 45 minutes to fully deploy, primarily because of package installations. An hour-long delay in the notification is acceptable, and helps avoid false alarms.

For user authentication, we use the Amazon Cognito service. It offers account creation and authentication of the user. If a user wants to sign up it provides an email and

password. After that he will get an email from amazon to verify it by clicking a link. After that he will have a persistent account in our system.

Currently authorization operations are performed inside our application, by communicating with the cognito api. The original plan was to redirect to the frontend authorization interface hosted by cognito, however this required our application to be hosted on https, which was not possible due to the limitations of the laboratory environment.

RPO and RTO estimates

The services provided by our system are not of particularly high business importance, and their immediate availability is rarely critical. Therefore we estimate that the RTO of the system is relatively long - about 120 hours. To achieve this, we use the email notification system that alerts administrators or developers about an outage. We expect most problems and outages to be solvable relatively quickly, and therefore expect the recovery time will in practice be no longer than 24 hours during weekdays, and no longer than 72 hours outside of holidays.

The data changes in our service are made by the users. So, changes to the data would likely depend on the number of users of our service. Additionally, considering the changes are exclusively composed of uploaded files, it is likely that the user will still have access to them and the ability to reupload them in the case of a data loss. This leads us to believe that the RPO of our system does not need to be very small - about 48 hours. To achieve this, we use daily backups of the database, which last up to a week, and may be restored in the event of a system failure, losing no more than around 24 hours of data.

Our system can for the most part handle a single availability zone failure. The load balancer and auto-scaling group ensure that the incoming traffic will be directed toward the instances located in the still functional availability zone. The RDS hosting our database is a Multi-AZ instance, providing appropriate failover. The only exception is the bastion host, which is located in only one availability zone. This means that a single availability zone failure may result in restricted administrative access to the system.

The recovery time of our system will depend mostly on the administrator and active developers who detect the system failure by notification email or system check. The system after failure can be automatically reloaded from the Cloudformation file that will automatically create all necessary resources like RDS, VPC with subnets and security groups, LoadBalancer, etc. Starting the system from Cloudformation usually takes around 35 minutes. So the recovery time will be equal to the time needed for cleanup of the old system (~30min) + creating the system from Cloudformation (~45min).

ALARM: "App alarm" in US East (N. Virginia) Inbox ×

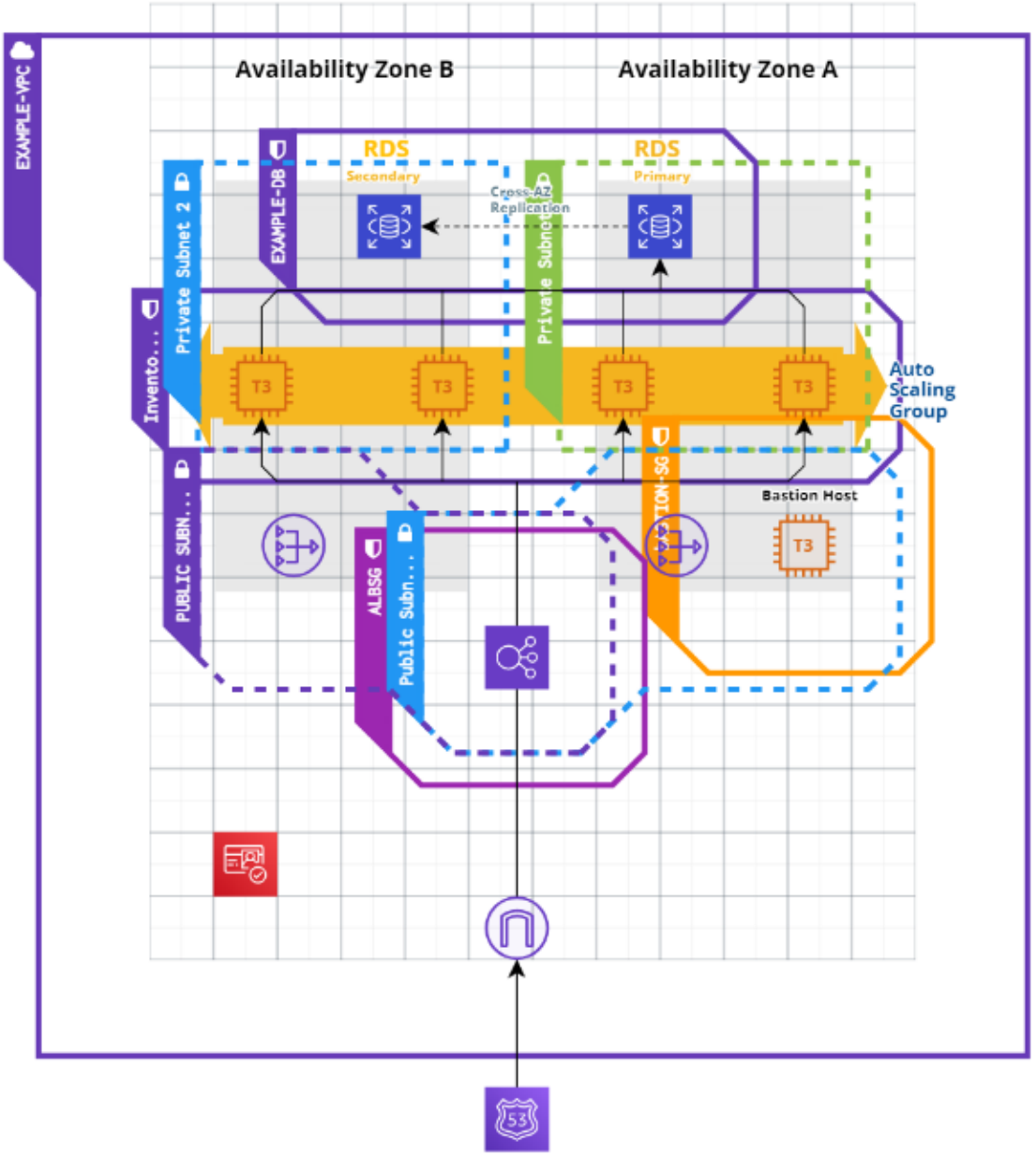
AWS Notifications <no-reply@sns.amazonaws.com> 7:17 PM (3 hours ago)

to me

You are receiving this email because your Amazon CloudWatch Alarm "App alarm" in the US East (N. Virginia) region has entered the ALARM state, because "Threshold Crossed: 60 out of the last 60 datapoints were less than the threshold (1.0). The most recent datapoints which crossed the threshold: [0.0 (23/06/22 17:16:00), 0.0 (23/06/22 17:15:00), 0.0 (23/06/22 17:14:00), 0.0 (23/06/22 17:13:00), 0.0 (23/06/22 17:12:00)] (minimum 60 datapoints for OK -> ALARM transition)." at "Thursday 23 June, 2022 17:17:37 UTC".

View this alarm in the AWS Management Console:
https://us-east-1.console.aws.amazon.com/cloudwatch/deeplink.js?region=us-east-1#alarmsV2:alarm/App%20alarm

Example warning email notification



Graph of the design

Link to graph:
https://app.cloudcraft.co/view/a456972e-1f8a-4c0c-bf02-04a40c4b0e9c?key=69e270aa-b302-4775-8d63-a0a56d19db95