University of Coimbra

Service Engineering

A Three-Tier Application on the Cloud Soundsfine

Author:
Sebastian Rehfeldt
Nº 2016181684

Author:
Jani Hilliaho
Nº 2016167354

April 20, 2017

1 Introduction

This report serves as a document for presentation and discussion of a three-tiered application developed during the Service Engineering course at the University of Coimbra. The aim was to create a scalable web application for handling multi-user playlists and deploy it on the Amazon Cloud. Users should be able to register for an account and use it to upload songs to the cloud. These songs can be managed, collected into playlists and finally played by the users.

The next section of this report will introduce the main technologies used to develop the application. It also gives details on the development and deployment process. Section 3 will focus on the database tier and the service layer which can be accessed using a REST interface. The front-end will be presented in section 4 which is followed by a discussion of the auto-scaling and load-balancing feature of Amazon Beanstalk in section 5. Finally, the report will be rounded-off by a summary explaining our insights from the work with the given technologies.

2 Architecture

The web application consists of three tiers which are clearly separated from each other. The service layer was written in Python using the Flask micro-framework. The Flask server is used for handling user requests and to communicate with the database. The REST services of the Flask server follow the OpenAPI standard and were documented using Flasgger. Also the user authentication takes place inside this layer. To allow the data to be persistent, the Flask server communicates with a MySQL database which forms the next tier. The communication therefore is handled by SQLAlchemy which is an Object Relational Mapper (ORM). The advantage of such an ORM is that relational data from the database is directly mapped to Python objects and vice-versa. This removes the job of always manually converting data to the appropriate format. The users can interact with the web application using a React front-end which forms the third tier. The React interface is used to display the data to a user and to allow him to interact with the server during requests. React is also responsible for the routing inside the application. React is an open-source JavaScript library developed and maintained by Facebook. It aims for a high performance and easy programming model. More detailed information about the different tiers can be found in section 3 and 4.

2.1 Development process and code structure

During the project, Github was used for storing the code[1] and for creating issues. As a second tool, waffle.io[2] was used for handling the issues. It is a kanban board like tool connected to Github, and has all functions needed for agile development and made it easy to have an overview over all the issues. It was used to assign issues and select features to be developed next. It also helped to document important information about implementations, tools and processes. The project folder structure is shown in figure 1.

As the short descriptions should be already descriptive, only the most important parts are explained in more detail. First, the Flask server was developed and connected to the database. When starting to develop the client side, two different servers were run: the Flask server and a server to serve the React interface. To allow the React server to request the Flask API, cross-origin

[1] `https://github.com/jhilliaho/ES-Project-2`

[2] `https://waffle.io/`

client

- **build** - folder for minified application created by command “npm run build”
- deploy.sh - script for copying minified files to the server folder
- **node_modules** - folder for node modules
- package.json - holds various metadata relevant to the project
- **public** - folder for static html files and their css files
- **src** - folder for the React application

server

- .**ebextensions** - folder for configuration files used in deploying
  - efs_mount.config - EFS configuration
- api.py - contains methods for handling requests and interacting with the database
- application.py - main application and definition of routes; authentication
- configuration.py - contains e.g. database configurations for the application
- configuration_example.py - configuration example.
- db_seed.py - python script for creating test data
- flask.log - the application log file
- requirements.txt - a file for storing python requirements
- schema.py - creates the database connection and defines object classes for the ORM
- **static** - contains js and css files for front-end. Created by deploy.sh
- **swag** - contains api documentation
- **templates** - contains HTML files for the application. Created by deploy.sh
- **uploads** - contains uploaded songs if there is no EFS for them

Figure 1: Web application source code folder structure

requests were enabled. For a final deployment on Amazon, a deploy script was created at the client folder which creates a minified build version of the React interface and copies the necessary files to the server side which are then served to clients accessing the Flask server. The next part should give an overview on how the code was deployed to Amazon.

2.2 Deploying the application to Amazon Beanstalk

When deploying the application, the front-end part must be minified and copied to the ”server” folder, because it’s the only folder that is deployed to the cloud. It can be minified and copied to the ”server” folder by running the script ”deploy.sh” inside the ”client” folder. Submitted project and the project in GitHub already have the front-end minified, so there is no need to minify it again if it’s not modified. Before using the script for minifying, node modules must be installed by executing command ”npm install” inside the ”client” folder.

The application also needs a configuration file for the database connection and some other tasks, and it must be created before deploying this application by copying the file ”server/configuration_example.py” to ”server/configuration.py”. The configuration file must be filled with correct database settings if the server is started locally.

The application was deployed in Amazon Beanstalk using The Elastic Beanstalk Command Line Interface (EB CLI). Before deploying, the EB CLI was installed with pip (Python package installer), following the instructions of the installation guide[3]. The EB CLI provides interactive commands for managing environments from a local repository. The application was deployed

[3] `http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-install.html`

following the EB CLI usage guide[4].

After installing the EB CLI, the general configuration was made running command "eb init" inside the "server" folder. If the user wants to use the Elastic File System (EFS) with this application, it's important to select a region that supports it. At the moment when this application was built, the only region supporting EFS in Europe was "eu-west-1", which is located in Ireland. If the EFS system is used, it must be created separately and its id must be copied to the configuration file "server/.ebextensions/efs_mount.config". Otherwise the configuration file should be removed. EFS can be also created after launching the application, because the configuration files are checked every time the user deploys changes for the application. After the general configuration, an application instance was created using command

```
eb create <project_name> -db -db.engine mysql -db.pass <db_password>
--database.size 5 -db.user <db_user> -k <key_pair_name>
```

This command launches an application and also creates a mysql database and a user to it with username and password provided in the command. Executing this command can take over 30 minutes. After launching the application, it can be opened in browser with command "eb open". If there is need to change its source code, the application can be updated by command "eb deploy". The application can also be terminated by command "eb terminate".

3 Backend

The job of the backend is to handle user requests and to assure the persistence of data using a database. This section is therefore separated into a database and a service layer part.

3.1 Database Layer

A MySQL database is used for storing the data. The schema of the database can be seen in the figure 2.

[4] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html

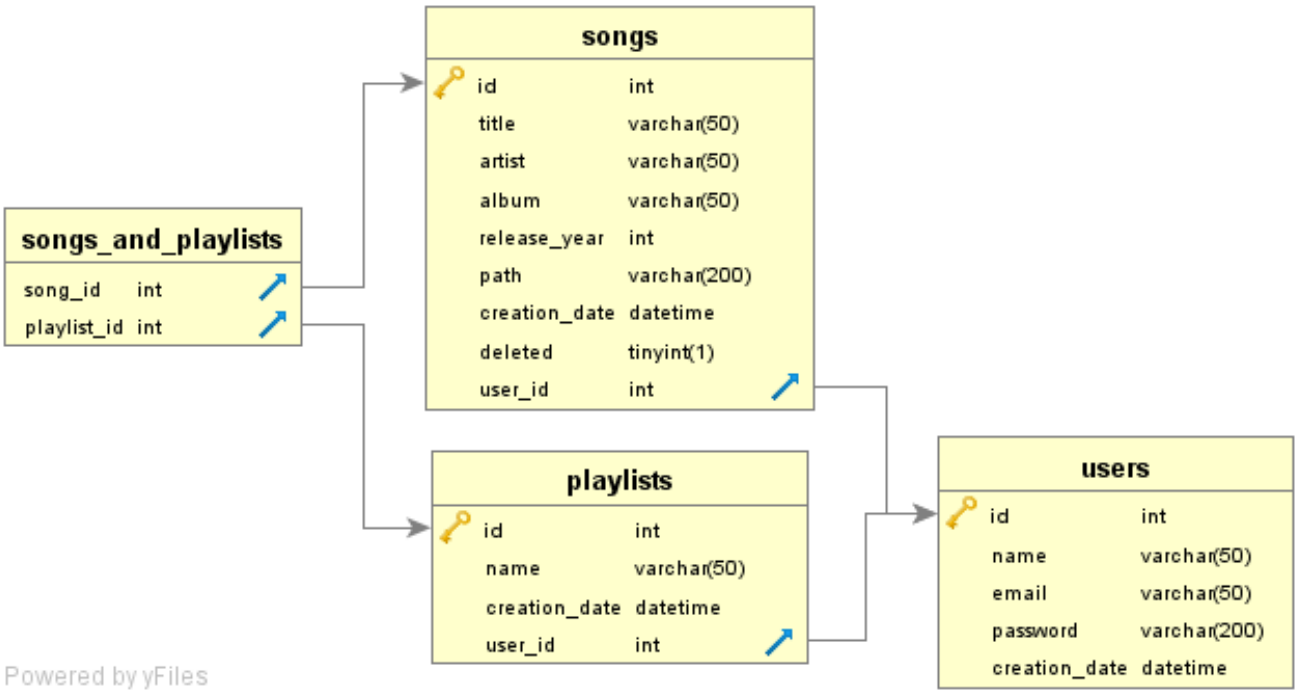

Figure 2: Database schema

The database schema was almost the first thing to plan during this project, because so much other features depend on it. In the schema hierarchy, there are many-to-one relationships from songs and playlists to user so that every song and playlist is owned by just one user. There is also a many-to-many relationship between songs and playlists, so that every song can be part of any number of playlists and vice versa. The relationships are used with "cascade delete", which means deleting all child objects if the parent is deleted. All of these relationships are described as Python code with SQLAlchemy library in the file "schema.py".

3.2 Service Layer

The main logic of the application is written in "application.py". It is responsible of the user authentication, database connection and handling the requests. The application was developed using Python virtual environments by storing all dependencies in the file "requirements.txt". This was mandatory for the deployment to Amazon and assures that the same version of modules are used by all contributors.

To connect to the database, application.py imports schema.py which creates the object classes and a creates a connection to the database using SQLAlchemy. Because SQLAlchemy returns the data as objects, a function "selectJson" was implemented to get selected fields of these objects in JSON form.

The "application.py" itself only defines the routes. When a route is requested, it calls one or more functions inside "api.py" which contains the real logic for a route. The concrete actions were separated to a second file to create smaller files which are easier to maintain. When using local development and a separate React server, the CORS mode must be used, and the Flask-Cors module was used for that. The documentation of the routes was done using the Flasgger module and external .yml file which contains implementation details for each route. The documentation

was done after creating the project. A discussion about this can be found in the summary at the end of this report.

To assure that only registered users can access the application, a security mechanism using email and password was implemented. When an user tries to log in, the username and password are checked and if they are correct, the user will be logged in. Flask-login[5] module was used for handling user sessions. For security reasons, all user passwords are hashed to avoid saving them as plain-text. However, HTTPS was not used in this project, which means that passwords can be read from traffic between the client and the server when user registers itself or logs in.

All API requests except the ones for registering and logging in are rejected if the user is not logged in. All GET requests are redirected to the login page and all other requests get "Forbidden" status code as response.

3.3 Elastic File System (EFS)

When scaling the application by using multiple server instances, the application could run into problems when the files are stored only in the server instances. To tackle this problem, the Amazon Elastic File System (EFS)[6] was used. EFS instances can be created when deploying the application, but that causes the EFS instance to be always removed when the application is terminated. To retain the data even if the application is removed and created again, the EFS instance was created before launching the application. The EFS is then mounted to the application by a configuration file in the folder ".ebextensions". If the EFS mount point is detected during the application startup, it is used for storing the files. Otherwise, the folder "uploads" is used for them inside the application folder.

4 Frontend

The front-end is used to display data to users and to allow them to interact with the application by uploading, managing and playing songs and playlists. The requirements of the assignment sheet were followed with also adding the possibility to play single songs in the songs section and full playlists in the playlist section. One additional feature is also that the user can decide if a song should be completely removed from disk and all playlists or if it should only be marked as deleted and removed from the song list. The second option has the benefit that the song can still be played in existing playlists.

The interface was styled using Bootstrap library and contains a home page, a song page, a playlist page and a user page. Also it is possible to logout the user from every place using the navigation bar. To structure the code, different components were implemented and composed. An overview about the component structure is given in figure 3.

[5] `https://flask-login.readthedocs.io/en/latest/`

[6] `https://aws.amazon.com/efs/`

App - Defines main component hierarchy and routes
- Layout - Main layout, e.g. navbar
  - Home - Welcome page
  - User - Page for changing username and email
  - SongList - Page for managing songs
    - AddSong - Component for adding a song
    - SongRow - Component for managing a single song
  - Playlists - Page for managing playlists
    - AddPlaylist - Component for adding a playlist
    - AudioPlayer - Component for playing song in playlist
    - PlaylistRow - Component for managing a single playlist
    - SongList - Component showing songs of a single playlist

Figure 3: Front-end component structure

The front-end source code can be found from folders "src" and "public", where "src" includes the React app, and "public" includes static HTML files for registering a new user, logging in and for the main page. An overview about the folder structure is given in figure 4.

src - folder for the React application
- **app** - "App" component
- **components** - folder for song and playlist components
  - **playlist** - components for managing playlists
  - **song** - components for managing songs
- **layout** - folder for page layout component
- conf.js - configuration file for front-end, includes e.g. hostname for api requests
- index.css - stylesheet for index.js
- index.js - React application root file

public - folder for static html files and their css files
- **static** - includes a favicon, and css files for login and register pages,
- index.html - main page HTML file
- login.html - HTML page for login
- register.html - HTML page for registering a new user

Figure 4: Front-end folder structure

The application consists of three HTML pages which are registering page, login page and the React application. The login and registering logics are separated from the React app to enable easier user authentication by redirecting all unauthorized GET requests to the login page. The React application is built as a single-page application (SPA) which means that it loads a single HTML page and dynamically updates it based on user interaction. The visible part of the application consists of layout module (navbar) and the content created by other components. The application also has a routing component to allow navigating inside the application using forward and backward buttons or changing the page URL.

The React application fetches the data needed from the REST API implemented for this project. All data requests are made at the background by AJAX without need to refresh the page.

5 Scaling

Scaling means launching and closing server instances based on the server load. Because customers pay only for used resources in cloud computing, they should always have just as many server instances running as needed at the moment. In Elastic Beanstalk configuration it's possible to define triggers for launching and closing instances based on different conditions.

In this project, load balancing was enabled limiting the amount of instances to 1-4. The scaling trigger was based on CPU utilization because the server was used to calculate Fibonacci values which causes a high CPU load. A new server instance was launched if the CPU utilization of all running instances had been over 99% for one minute. Apache JMeter[7] was used for sending the request and logging the responses. In this test, 10 threads were used for sending requests to calculate the 33rd Fibonacci value so that they sent a new request instantly after getting the response for previous one. The response times are shown in figure 5.

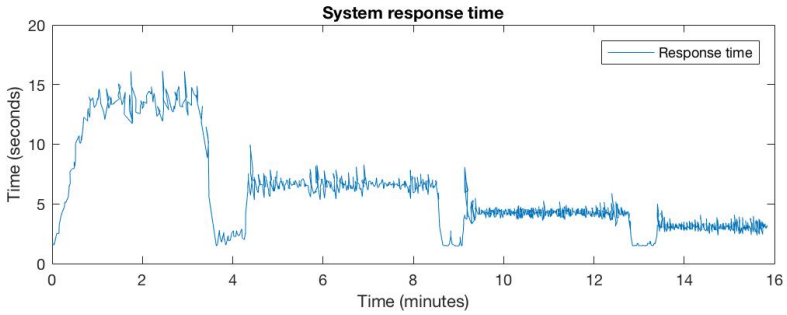

Figure 5: Response time

In this figure it's easy to see response times going down three times as a new instances were launched. When launching a new instance, the load balancer rejected all requests if all of the instances were already handling previous requests. That can be seen as deep pits in the figure.

The instance loads were logged in Amazon CloudWatch, which is a monitoring service for AWS cloud resources. The load graph is shown in figure 6. This figure shows that with this configuration, the load balancer had 4 instances running after 15 minutes of sending requests.

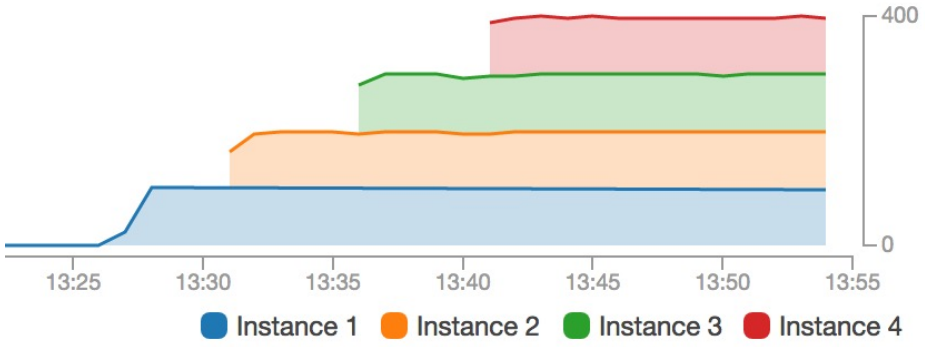

Figure 6: Instance loads

[7] http://jmeter.apache.org/

6 Summary

The objective for this project was to create a three tier application using OpenAPI standard and Flask for web services, React for the client, and SQLAlchemy as Object Relational Mapper (ORM) and for creating the database connection. The application should be deployed on Amazon Elastic Beanstalk and it should be scalable. By now, all requirements found from the instructions are implemented and they should work correctly.

Because many of technologies used for this project were new for us, we learned a lot of things during this project. Regardless of hours spent refactoring and writing the code again and again, we are sure that many of the implemented features could have been done in better way. Maybe the most significant example of doing things in a stupid way was to write the REST API by hand documenting it afterwards with Flasgger, because the API could have been created automatically from the Flasgger documentation. The React application component structure may also be totally different if we need to write a similar application again.

Based on our experience, React, Flask and Flasgger are great tools for writing web applications, and the application flexibility is even better if its deployed on Amazon Elastic Beanstalk. After struggling some hours with Amazon EFS, we could even share the song files between the application instances.

Programming this application with so many new technologies wasn't just a bed of roses, because it's not easy to learn so many things simultaneously. One funny example was trying to use "print" function for debugging also in JavaScript. Despite of working well for printing messages in Python console, it didn't work as expected in Javascript, but instead tried to print our application to real paper with printer.