← View All Projects

Description

Slow + Reverb Music Streaming Webapp

GitHub: https://github.com/mattjinks/Slow-Reverb-Music.git

This project builds a unique microservices-based music streaming web application featuring slow and reverb audio effects. Utilizing Microsoft's Azure cloud services, the application includes cloud storage solutions for media files and meta-data as well as a custom API for managing the music library. Azure Pipelines was used to automate build, push, and deployment processes. Web Audio API is integrated to accomplish the reverb and slowing audio effects.

Technologies Used:

Microsoft Azure *(SQL Database, Blob Storage, Functions, API Management, Container Registry, Web App)*

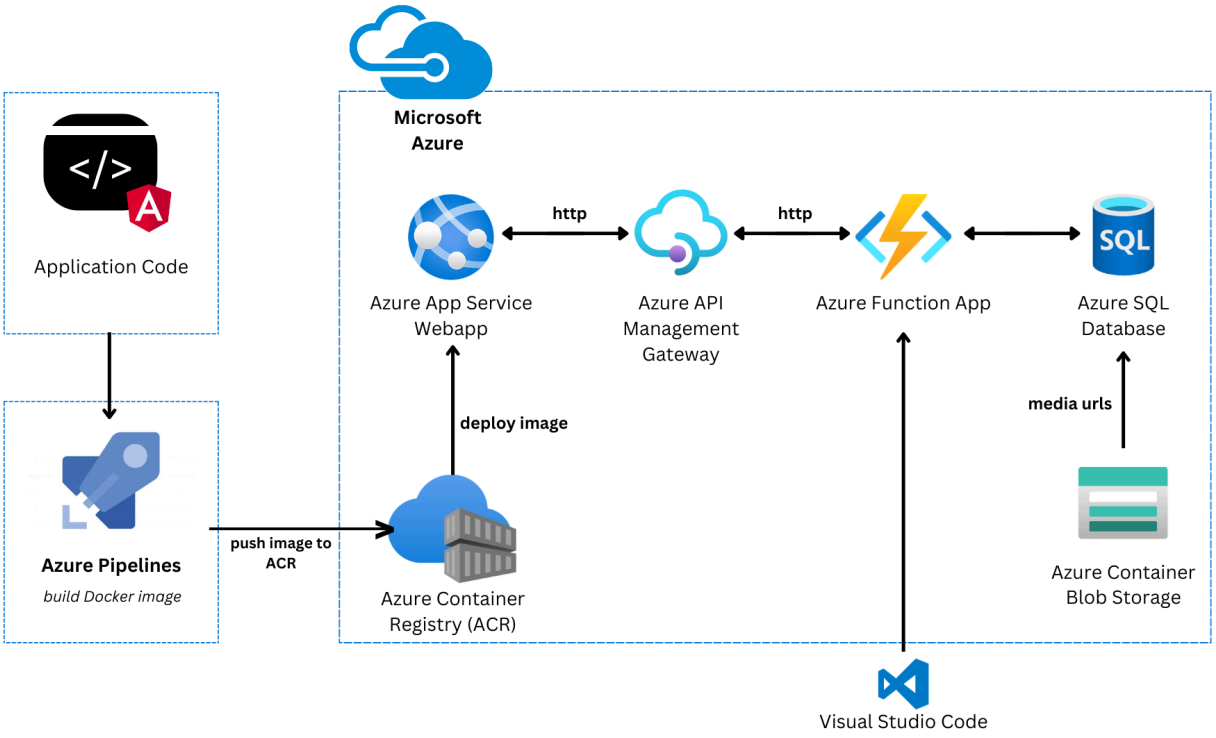
Docker, Azure Pipelines, Angular, TypeScript, HTML, CSS, Web Audio API

Why is this app useful?

"Slowed + Reverb" music has gained immense popularity due to its ability to transform songs, enhancing emotional impact and revealing hidden nuances. A music streaming app featuring this style could attract a large audience seeking new listening experiences, as evidenced by the trend's viral success on platforms like YouTube and TikTok. By offering audio effects customization, such an app could establish itself in a growing niche market.

Demo

Cloud Services & Architecture



This project leverages a suite of Azure services to create a comprehensive music streaming solution.

• **Azure SQL Database:** Provides a reliable and scalable relational database for storing album and song metadata

• **Azure Blob Storage:** Offers a cloud storage solution for music files and images.

• **Azure Functions:** Deployed using Visual Studio Code, these serverless functions provide a flexible backend for retrieving data from storage and handling API requests.

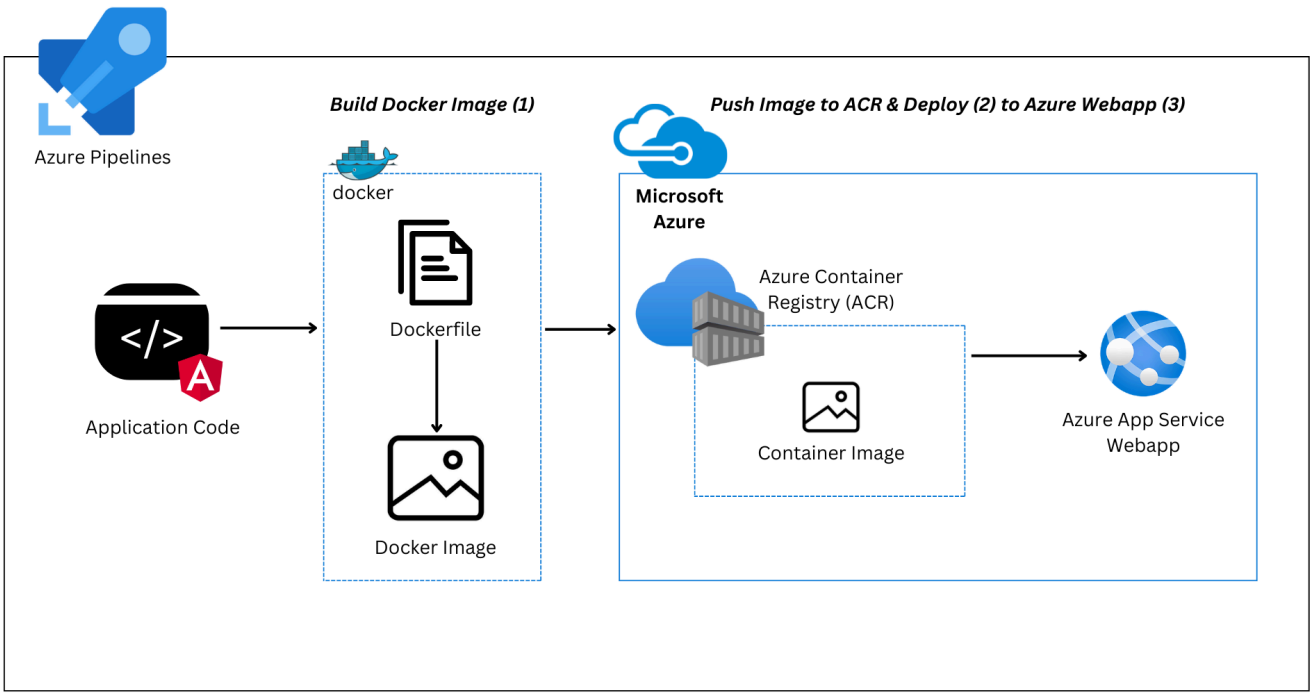
• **Azure API Management:** Streamlines the creation and exposure of APIs, enhancing security, scalability, and control over the project's backend services.

• **Azure Container Registry (ACR):** Serves as a private repository to store Docker container images that are built using an Azure Pipeline.

• **Azure Web App:** Hosts application (containerized in Docker image store in ACR) that is accessible to users. Provides the infrastructure and runtime environment for the application.

This combination of Azure services empowers the project with a robust infrastructure.

Deployment



Using Azure Pipelines, I built a pipeline that runs three tasks:

1. Build a Docker image

2. Push image to Azure Container Registry

3. Deploy image to Azure Webapp

User Interface

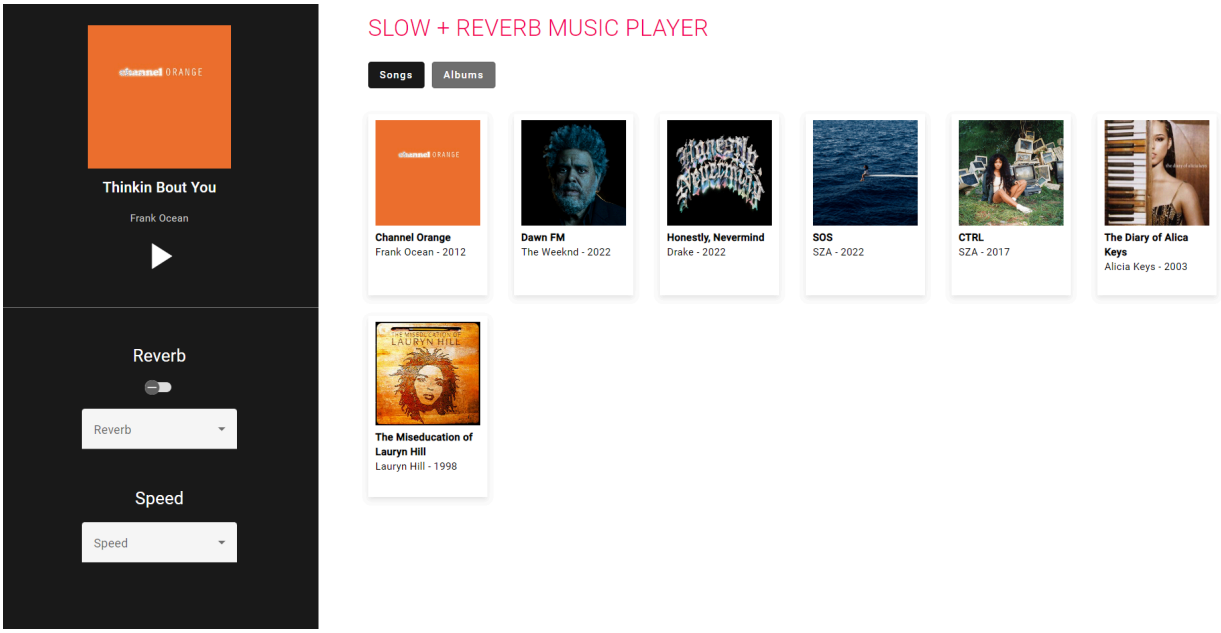
The view consists of three key elements:

• A music library that dominates the view, displaying the complete catalog of songs for streaming.

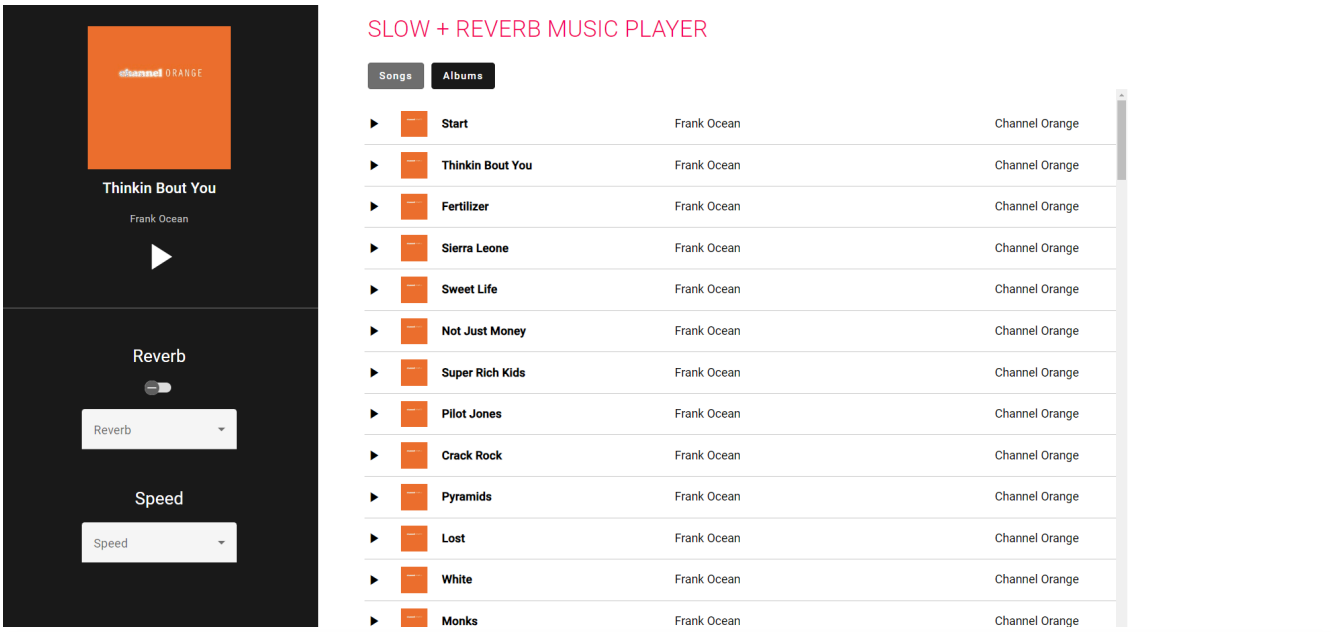
• A top-positioned filter bar with “Songs” and “Albums” buttons for user-driven browsing.

• An audio player that enhances the listening experience with cover art, song information, and controls for playback, reverb, and speed.

Screenshot of Web App: Library displayed as "album-list"



Screenshot of Web App: Library displayed as "song-list"



Front-End Implementation

The front-end application is created using Angular. Angular was chosen for this project because it has robust Single-Page Application (SPA) capabilities. An SPA dynamically rewrites the current web page with new data from the web server, in contrast to the traditional way of the browser loading entire new pages which ultimately provides a smoother user experience.

Angular's routing module helps in handling navigation from one view to another while keeping the application within a single page. As the user interacts with the application, Angular changes the browser's URL dynamically, but the page itself does not reload, thus maintaining a SPA's behavior. This project consists of 3 routes (excluding empty path).

Routes: Empty path defaults to SongListComponent

```
const routes: Routes = [
  { path: 'song-list', component: SongListComponent },
  { path: 'album/:id', component: AlbumComponent },
  { path: 'album-list', component: AlbumListComponent },
  { path: '', component: SongListComponent },
];
```

I wanted to take advantage of Angular's component-based architecture. In Angular, Components are the fundamental building blocks for creating user interfaces. A component consists of 3 main parts:

• **A TypeScript Class:** defines the component's logic, properties, methods, and interactions with services.

• **HTML Template:** This defines the component's visual representation using HTML.

• **CSS styles:** Specific to the component to customize its appearance.

Key Advantages of Angular Components:

• **Reusability:** Components can be used multiple times throughout your application, promoting a "write once, use anywhere" philosophy. This reduces code duplication. In this project for example, a list of songs is a list of "Song" components.

• **Modularity:** Components encapsulate logic, data, and presentation, making the codebase more organized, maintainable, and easier to test.

• **Data Binding:** Components can bind data from their TypeScript logic to the template and vice versa, enabling dynamic updates of the user interface.

Angular Services

Angular services, as TypeScript classes, provide a centralized way to manage functionality and data shared across components within an application. This architecture is ideal for maintaining consistent state, interacting with external APIs, or performing calculations.

This project creates and leverages two Angular Services:

• **The Music Service** facilitates communication with a custom API, enabling efficient retrieval of data and media resources.

• **The Audio Processing Service** manages the playback state of songs (playing/paused) and utilizes the Web Audio API to apply audio effects. I didn't create a microservice to avoid latency issues.

← View All Projects