Project 6

Alex Mazur and Ryan Oliva

Project Summary

- **Title:** Cloud Groove
- **Names:** Alex Mazur and Ryan Oliva
- **Description:** Our project will be a web-accessed application that allows users to login, upload their own music, and then stream that music to any of their devices with a web browser. This application will be hosted using AWS.

Summary

Work done

Currently our Java Spring MVC application has a user login/signup, a user home that lists their uploaded media, a method for uploading their own personal media, and a way to play media that they have uploaded right in the browser. We decided to break our project up into four microservices. These services are called WebService, UserService, ContentService, and SongService. Each is at different levels of functionality. The WebService functions as a traditional controller in the MVC model, it is the only service that “talks” to our view. The UserService can handle user login and signup and interacts with a local DB instance. The SongService interacts with a local DB instance to add songs and playlists. The SongService microservice supports some playlist operations, but the WebService and view do not. Current supported operations include adding songs, getting songs, getting playlists (no way to add new playlists), and getting song metadata. The ContentService allows a user to upload songs as well as deliver songs for streaming. The ContentService can upload and deliver songs locally or via AWS and is currently set up to upload and stream content from AWS. The ContentService also communicates with the SongService to post song meta-data to the SongService database when a song is uploaded.

Work Split

The split of work is hard to really say. We utilized Intellij’s CodeTogether to pair programs often, so a lot of the features had both hands involved. Neither of us had used Java Spring before, so we spent a lot of time learning all of these new tools together.

Changes or Issues

We originally planned for a monolith application, so the biggest change is in the form of our project architecture. Instead, we have multiple applications, or microservices, running together. Most of the issues we encountered had to do with dealing with file uploading and downloading between microservices, but those have been solved. We have an application that does work to a minimum standard.

Patterns

We use a lot of patterns, some are hand made and some are abstracted behind Java Spring tools. The main pattern for our application is a microservice architecture pattern. I'm not sure if this is an official design pattern, but it's important to the design of our project and is a big change from a monolithic OO-application. Instead of a monolith, we have four independent services that communicate with each other via REST API calls; how each service works is irrelevant to every other service.

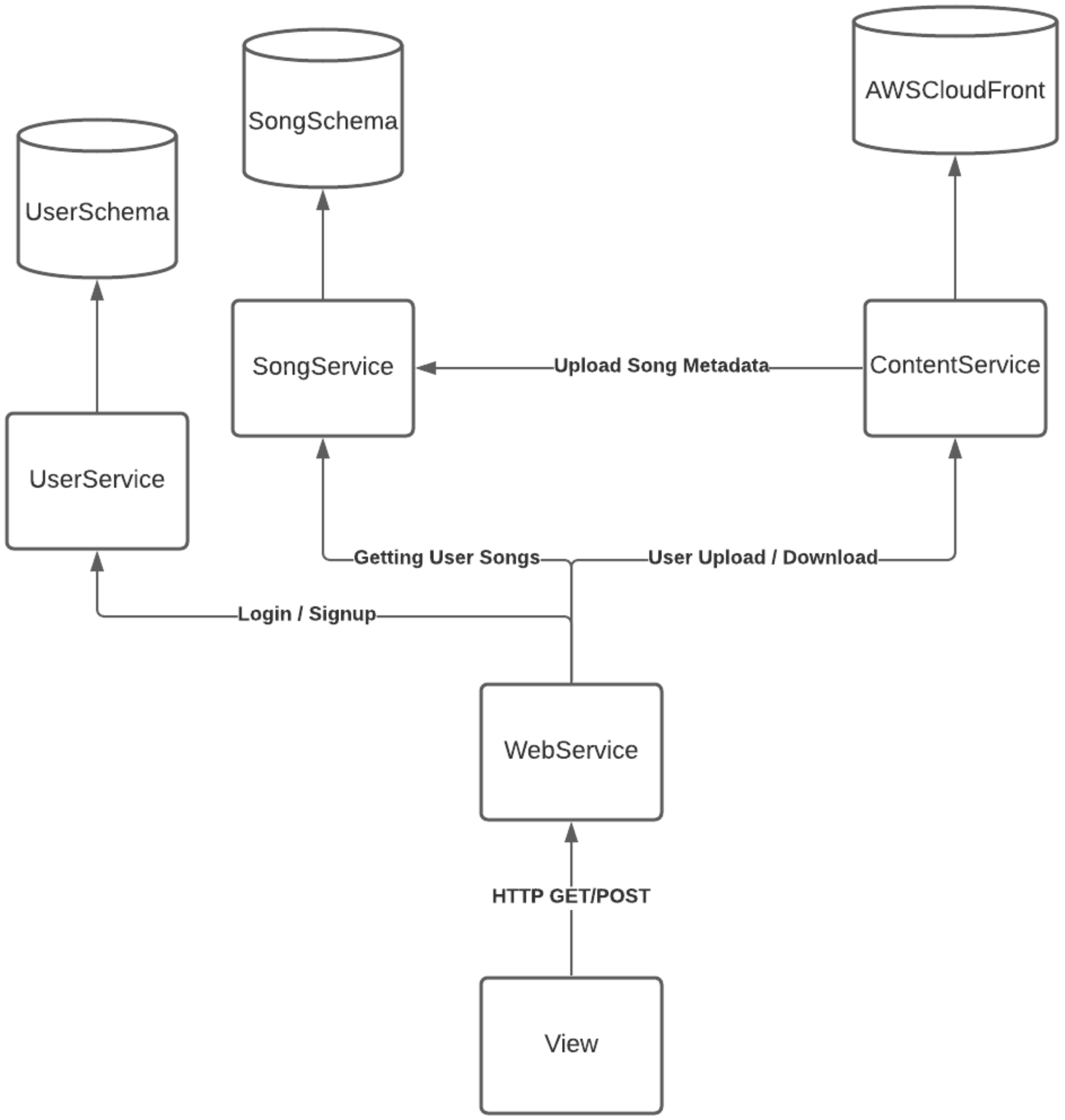
Another main pattern we use is MVC, specifically in our WebService. Some of this is abstracted away for us by Java Spring. The model, for example, is baked in Spring, so we just include it in our controller and can manipulate it easily. We pass our model to a tool called Apache Freemarker Templating Engine, which generates an HTML file based on our model and sends it to the user.

Throughout our microservies we heavily utilize builder and factory patterns. We use a factory pattern in our ContentService to determine the provider of the upload and download services. Anytime we need to retrieve data from our database (in the UserService and SongService) we use a builder pattern to construct local objects based on the data returned from our queries. This pattern is cheating in a way, since we use lombok annotations to include that functionality.

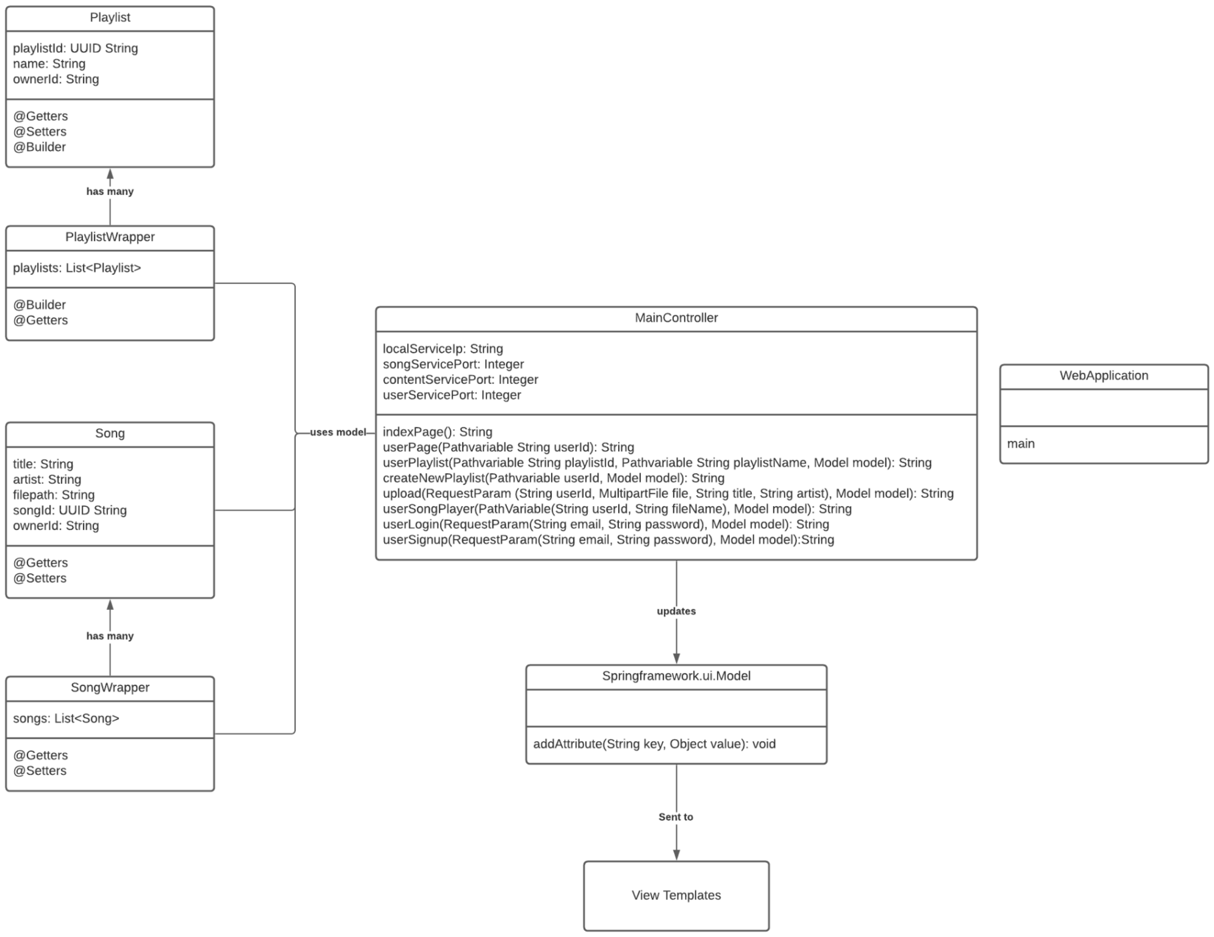
It can be argued that we are using a facade pattern here. Our WebService is the facade to the other microservices that are working behind the scenes.

Class Diagram

First, it's important to understand the relationship between our microservices. Provided below is a high-level diagram of how the microservices communicate with each other, and what tasks they expect from each other.

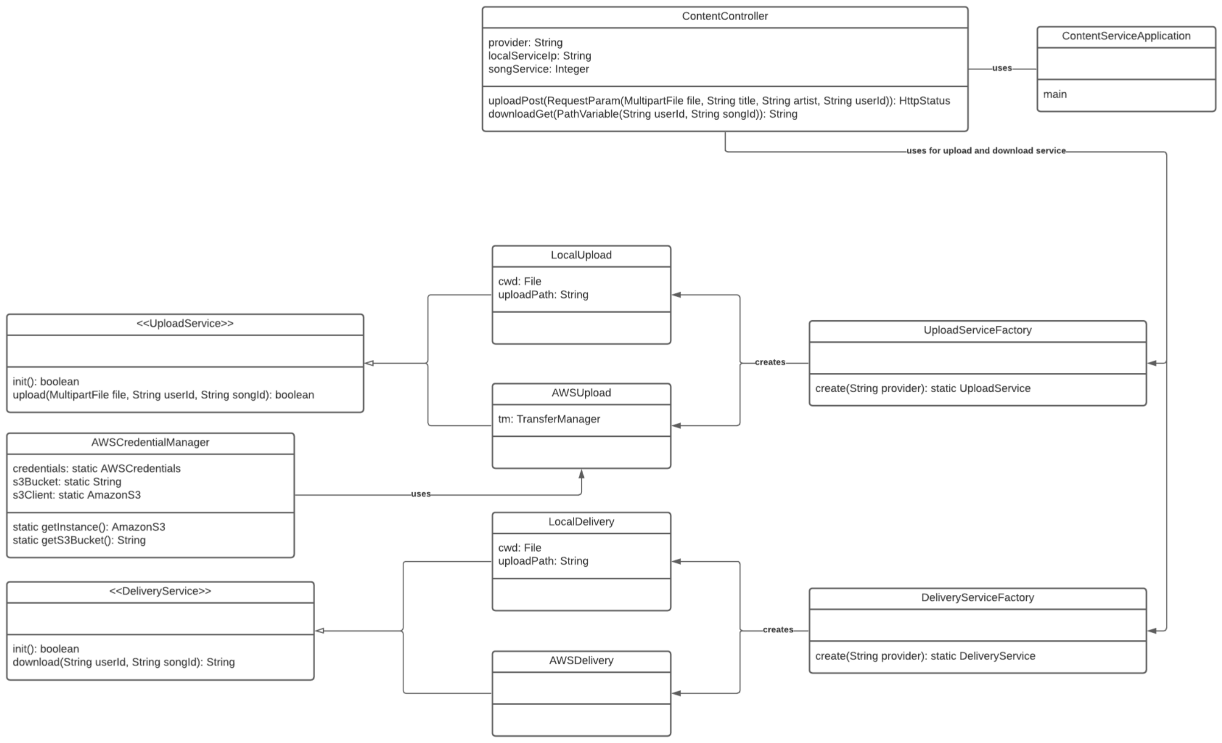


The first class diagram is going to be of the WebService microservice. This highlights an MVC pattern, as well as some builder patterns for out models. This also shows a facade pattern, since this is what the user interacts with, and the true complexity is abstracted behind this interface. You can find this diagram directly below.

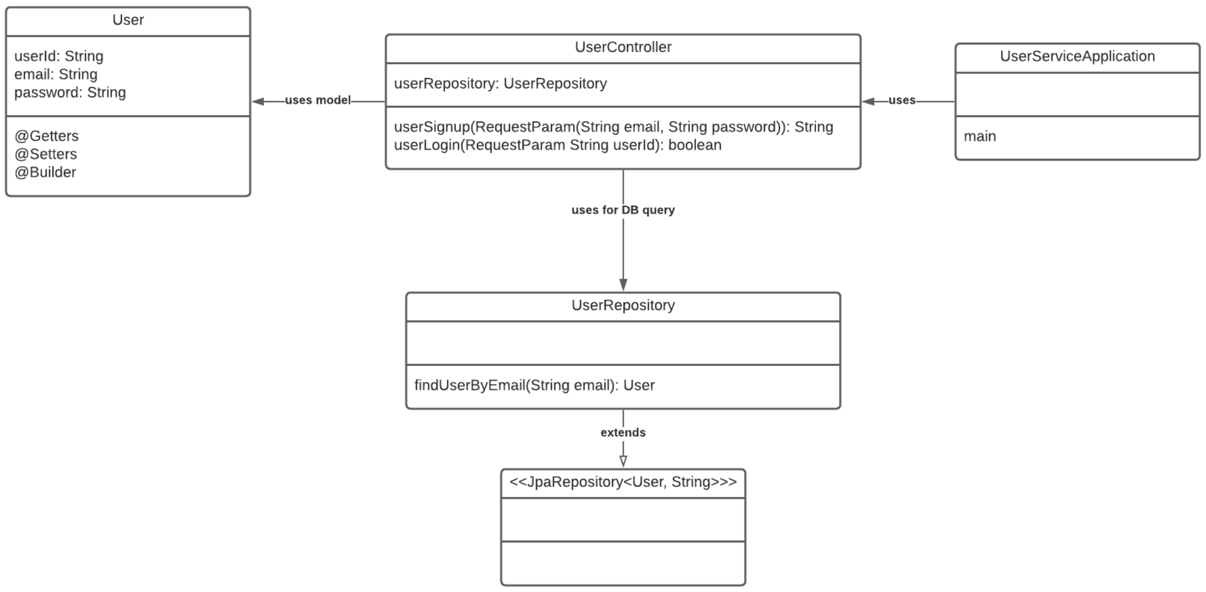


Playlist
playlistId: UUID String
name: String
ownerId: String
@Getters
@Setters
@Builder
has many
PlaylistWrapper
playlists: List<Playlist>
@Builder
@Getters
Song
title: String
artist: String
filepath: String
songId: UUID String
ownerId: String
@Getters
@Setters
has many
SongWrapper
songs: List<Song>
@Getters
@Setters
uses model
MainController
localServiceIp: String
songServicePort: Integer
contentServicePort: Integer
userServicePort: Integer
indexPage(): String
userPage(Pathvariable String userId): String
userPlaylist(Pathvariable String playlistId, Pathvariable String playlistName, Model model): String
createNewPlaylist(Pathvariable userId, Model model): String
upload(RequestParam (String userId, MultipartFile file, String title, String artist), Model model): String
userSongPlayer(PathVariable(String userId, String fileName), Model model): String
userLogin(RequestParam(String email, String password), Model model): String
userSignup(RequestParam(String email, String password), Model model):String
WebApplication
main
updates
Springframework.ui.Model
addAttribute(String key, Object value): void
Sent to
View Templates

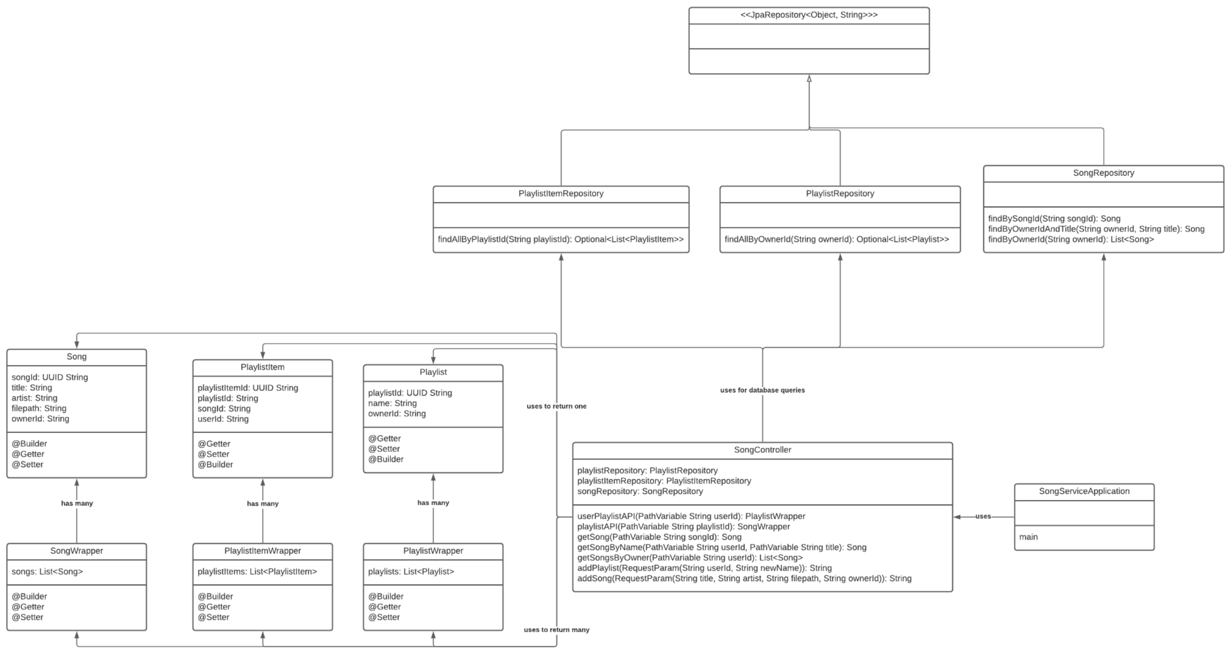
Next up is the ContentService diagram. It has a main controller that drives its function, but this time there is no view or model. This service simply accepts REST API calls to the endpoints it's listening on and returns some back to the caller. This microservice highlights an example of a factory pattern. The microservice has two main operations, upload and download, but how it uploads and downloads is irrelevant. The controller deals with an UploadService and DeliveryService interface, and retrieves a concrete object via a factory. The factory returns an object based on an application setting. It can upload and download files either locally or via AWS CloudFront into an S3 Bucket.


ContentController
provider: String
localServiceIp: String
songService: Integer
uploadPost(RequestParam(MultipartFile file, String title, String artist, String userId)): HttpStatus
downloadGet(PathVariable(String userId, String songId)): String
uses
ContentServiceApplication
main
uses for upload and download service
LocalUpload
cwd: File
uploadPath: String
<<UploadService>>
init(): boolean
upload(MultipartFile file, String userId, String songId): boolean
creates
UploadServiceFactory
create(String provider): static UploadService
AWSUpload
tm: TransferManager
AWSCredentialManager
credentials: static AWSCredentials
s3Bucket: static String
s3Client: static AmazonS3
static getInstance(): AmazonS3
static getS3Bucket(): String
uses
LocalDelivery
cwd: File
uploadPath: String
<<DeliveryService>>
init(): boolean
download(String userId, String songId): String
creates
DeliveryServiceFactory
create(String provider): static DeliveryService
AWSDelivery

This next diagram is of the UserService microservice. It verifies a user login, or signs a user up. Notice that the DB queries are abstracted behind the UserRepository. UserRepository is a custom class that extends a class from Java Spring JPA that we can make to abstract our database queries.


User
userId: String
email: String
password: String
@Getters
@Setters
@Builder
uses model
UserController
userRepository: UserRepository
userSignup(RequestParam(String email, String password)): String
userLogin(RequestParam String userId): boolean
uses
UserServiceApplication
main
uses for DB query
UserRepository
findUserByEmail(String email): User
extends
<<JpaRepository<User, String>>>

Lastly we have the SongService. The SongService is one of the most “in-progress” microservices, and a lot of the methods listed in the controller are a work in progress. It does have the basic functionality required for a user to upload a song, download a song, and view all of their songs and playlists. Work must be done on the View, WebController, and SongController to facilitate more playlist interactions. You can find the class diagram below:



<<JpaRepository<Object, String>>>
PlaylistItemRepository
findAllByPlaylistId(String playlistId): Optional<List<PlaylistItem>>
PlaylistRepository
findAllByOwnerId(String ownerId): Optional<List<Playlist>>
SongRepository
findBySongId(String songId): Song
findByOwnerIdAndTitle(String ownerId, String title): Song
findByOwnerId(String ownerId): List<Song>
Song
songId: UUID String
title: String
artist: String
filepath: String
ownerId: String
@Builder
@Getter
@Setter
PlaylistItem
playlistItemId: UUID String
playlistId: String
songId: String
userId: String
@Getter
@Setter
@Builder
Playlist
playlistId: UUID String
name: String
ownerId: String
@Getter
@Setter
@Builder
uses to return one
uses for database queries
SongController
playlistRepository: PlaylistRepository
playlistItemRepository: PlaylistItemRepository
songRepository: SongRepository
userPlaylistAPI(PathVariable String userId): PlaylistWrapper
playlistAPI(PathVariable String playlistId): SongWrapper
getSong(PathVariable String songId): Song
getSongByName(PathVariable String userId, PathVariable String title): Song
getSongsByOwner(PathVariable String userId): List<Song>
addPlaylist(RequestParam(String userId, String newName)): String
addSong(RequestParam(String title, String artist, String filepath, String ownerId)): String
uses
SongServiceApplication
main
has many
has many
has many
SongWrapper
songs: List<Song>
@Builder
@Getter
@Setter
PlaylistItemWrapper
playlistItems: List<PlaylistItem>
@Builder
@Getter
@Setter
PlaylistWrapper
playlists: List<Playlist>
@Builder
@Getter
@Setter
uses to return many

Plan for Next Iteration

Our next iteration is going to largely be polish based, with a few technical additions. One main goal is to introduce functionality to add playlists and add or remove songs to playlists. Another is deletion of songs, users, playlists, etc. from our databases and file store.

If time permits, we would like to implement a follow feature, ala twitter or other social media. Essentially allow a user to lookup another user and follow their page, and stream their downloaded media. We could allow users to mark their content as public or private. For this feature we would also need to fully implement sessioning.

Lastly, an overhaul of the media player on the frontend. We are currently using the built in HTML audio player to play files. We would like to overhaul this feature and make something custom with additional control features.