Project 7 - CloudGroove

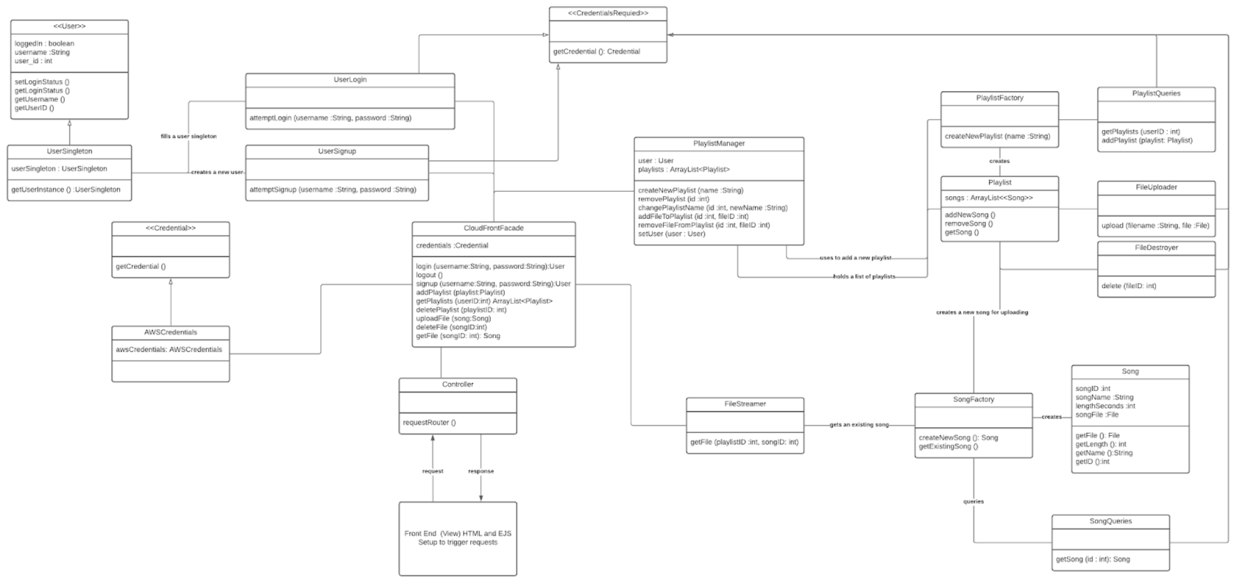
Alex Mazur and Ryan Oliva

Final State of System

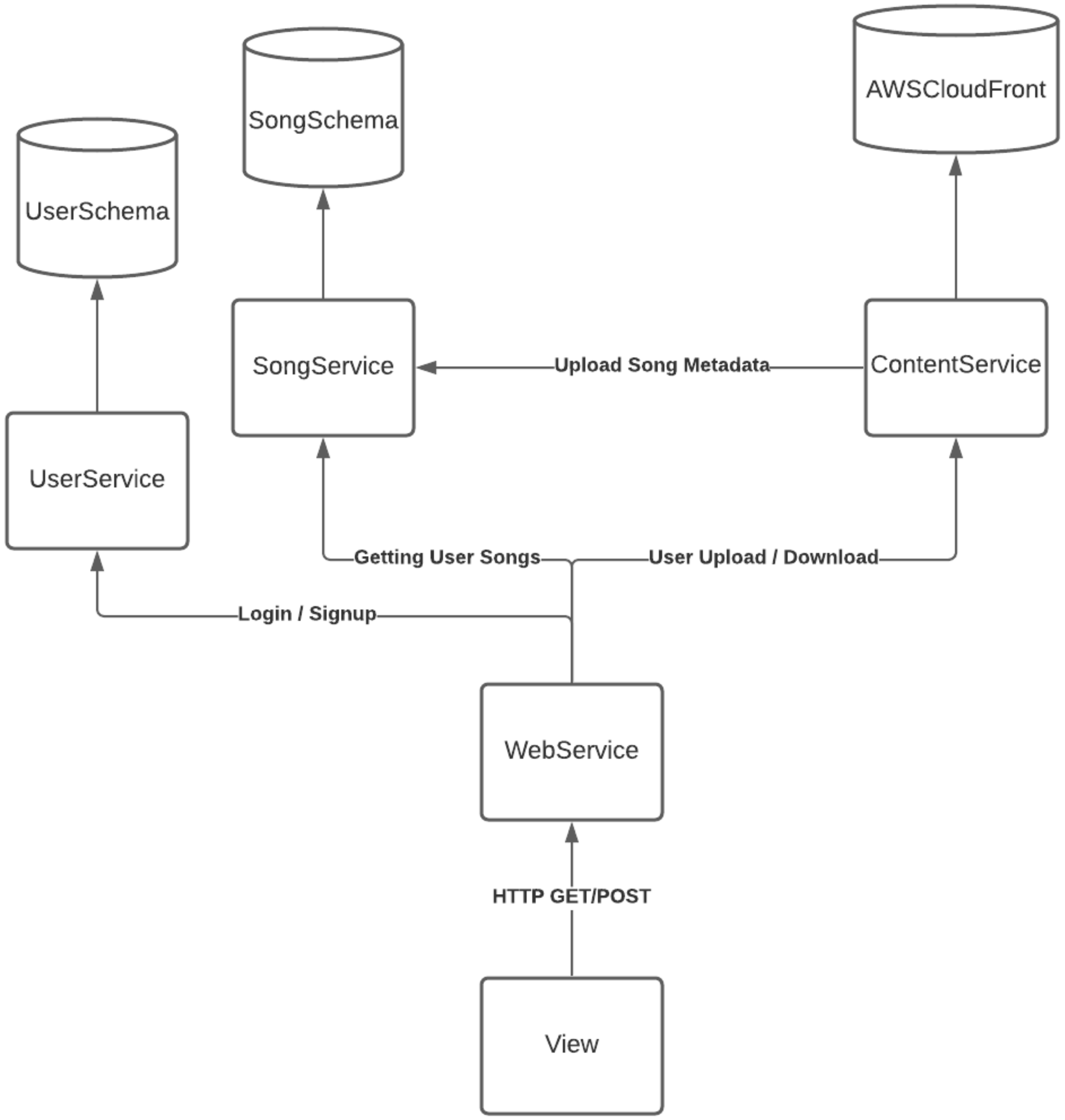
For our final project we made a fully functioning Java Spring Microservice web application that is deployed entirely to the cloud. The application has basic user account functions such as logging in and making an account. Each user must have their own media to upload and once uploaded the user will be able to stream their music from any device that supports a web browser. In order to facilitate the multiple functions of our application we are using a suite of AWS tools. The services we used include EC2 to run our backend services, an RDS postgres instance for user accounts and song data, and AWS CloudFront pointed towards an AWS S3 bucket for song storage and streaming. Some features that we did not fully implement include a friends system for sharing music and a playlist system. For project 6 we basically had a fully functioning application, but many of the systems were operating locally. We spent a lot of time setting up the project to be entirely cloud based and as a result we did not get around to implementing some of our extra features. At the end of the day, we spent the majority of our time familiarizing ourselves with a new framework and other new technologies, so simply setting things up was a good chunk of work.

Final Class Diagram and Comparison Statement

I'd like to start this section off by showing the class diagram from project 5.


<<User>>
loggedIn : boolean
username :String
user_id : int
setLoginStatus ()
getLoginStatus ()
getUsername ()
getUserID ()
UserSingleton
userSingleton : UserSingleton
getUserInstance () :UserSingleton
fills a user singleton
creates a new user
UserLogin
attemptLogin (username :String, password :String)
UserSignup
attemptSignup (username :String, password :String)
<<CredentialsRequied>>
getCredential (): Credential
<<Credential>>
getCredential ()
AWSCredentials
awsCredentials: AWSCredentials
CloudFrontFacade
credentials :Credential
login (username:String, password:String):User
logout ()
signup (username:String, password:String):User
addPlaylist (playlist:Playlist)
getPlaylists (userID:int) ArrayList<Playlist>
deletePlaylist (playlistID: int)
uploadFile (song:Song)
deleteFile (songID:int)
getFile (songID: int): Song
Controller
requestRouter ()
request
response
Front End (View) HTML and EJS Setup to trigger requests
PlaylistManager
user : User
playlists : ArrayList<Playlist>
createNewPlaylist (name :String)
removePlaylist (id :int)
changePlaylistName (id :int, newName :String)
addFileToPlaylist (id :int, fileID :int)
removeFileFromPlaylist (id :int, fileID :int)
setUser (user : User)
uses to add a new playlist
holds a list of playlists
PlaylistFactory
createNewPlaylist (name :String)
creates
Playlist
songs : ArrayList<<Song>>
addNewSong ()
removeSong ()
getSong ()
PlaylistQueries
getPlaylists (userID : int)
addPlaylist (playlist: Playlist)
FileUploader
upload (filename :String, file :File)
FileDestroyer
delete (fileID: int)
creates a new song for uploading
FileStreamer
getFile (playlistID :int, songID: int)
gets an existing song
SongFactory
createNewSong (): Song
getExistingSong ()
creates
Song
songID :int
songName :String
lengthSeconds :int
songFile :File
getFile (): File
getLength (): int
getName ():String
getID ():int
queries
SongQueries
getSong (id : int): Song

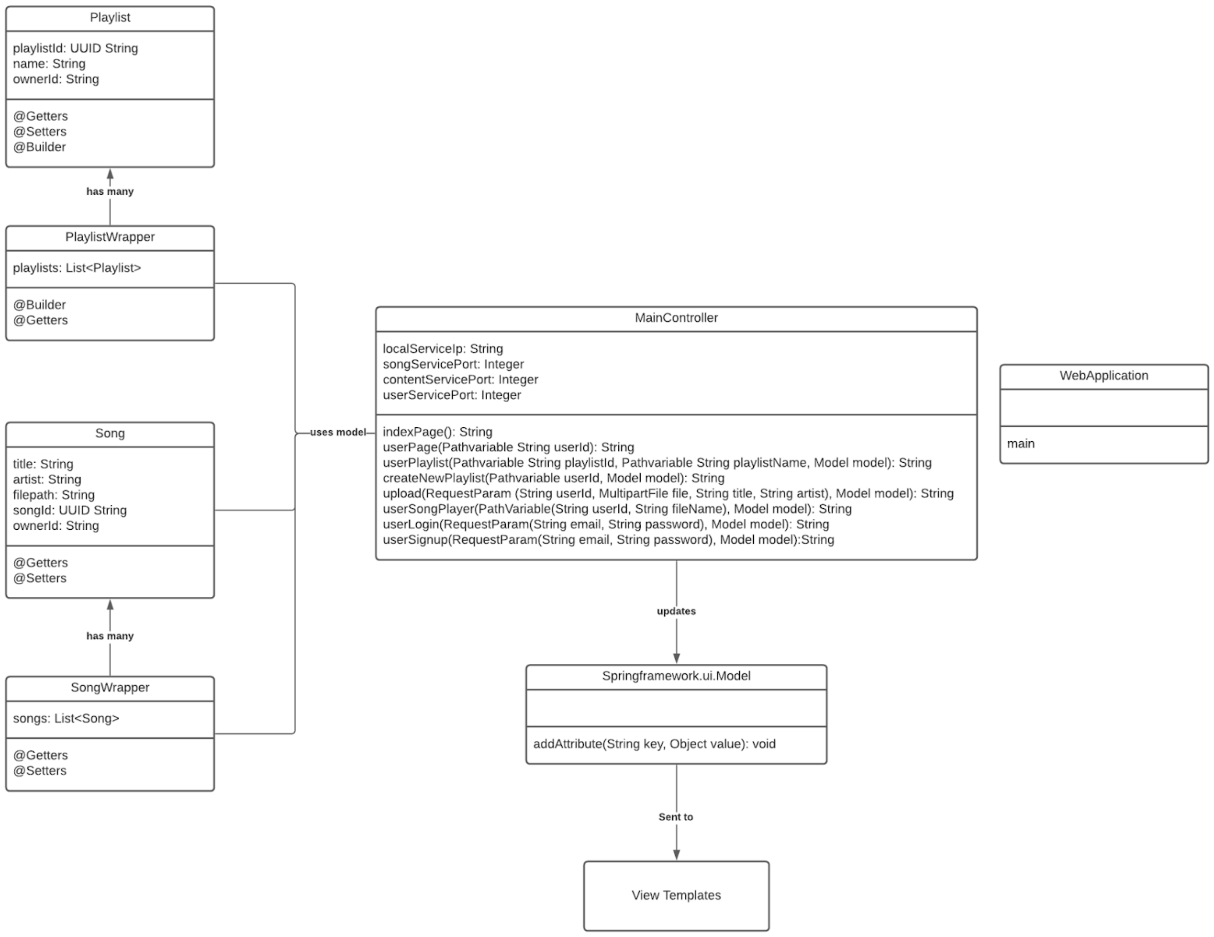
This diagram highlights a monolithic design, with some features that never made it to our final project. In an effort to pursue better design for a web service, we decided to microservitize our project. We identified the areas of this application that serve a distinct service, and thus our project was broken down into four separate applications. From a high level perspective this both complicates and simplifies the project. We now have four separate applications that can break, but it is easier to see how they all interact with each other. Below is a high-level diagram of how our final project functions.



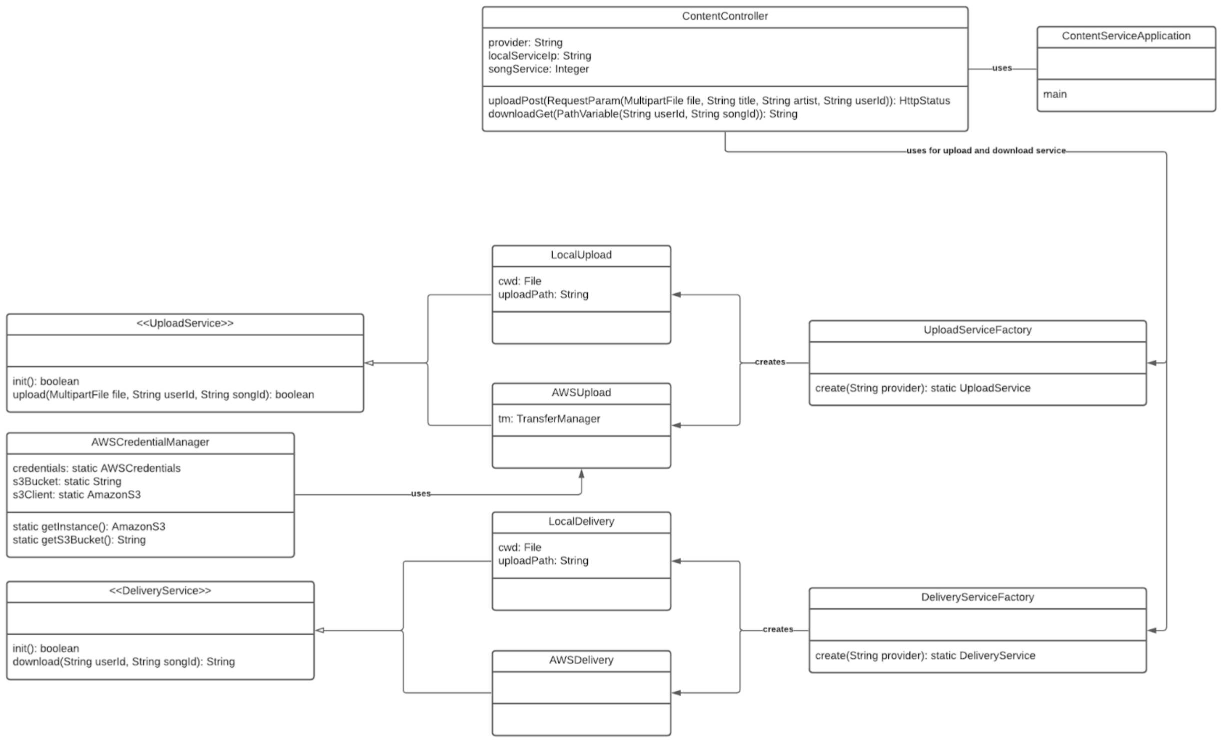
Each square is it’s own microservice (aside from the view square), and they all function separately from each other. Each microservice is listening for HTTP requests on certain URL endpoints, and has functions mapped to those URLs. For example, our view may submit a form to ‘/signup’. This request is received by the WebService. The WebService reads the data from the user and sends it to another endpoint, ‘:8081/api/signup’. The UserService is listening on port 8081 for this URL, so it receives the request, does the work of signing up the user, and then returns a response. This way, our microservices really don’t know about each other. They only know about sending HTTP requests to URLs. Each service doesn’t know or care about any other service running. This simplified our project because we have smaller and more cohesive parts. When things broke, we immediately knew where. This also made things slightly more complex, because at the end of the day we needed to keep track of our API endpoints, as well what type of request each endpoint is looking for, and what response it will return. We truly needed to maintain four distinct projects. There are solutions for microservice communications, but we didn’t get around to implementing anything. I think a recreation of something like Eureka Discovery Server (a sort of observer pattern for microservices) would have been a great edition. Overall, we think the microservice approach was easier to code, debug, and iterate on.

Our class diagrams didn’t change much from project 6 to project 7. With project 6 we delivered a fully functioning application that hit all of our major milestones. Of course, we didn’t implement every feature. We spent the majority of our time between project 6 and 7 eliminating bugs, changing the views in certain ways, and getting the entire application up and running on AWS.

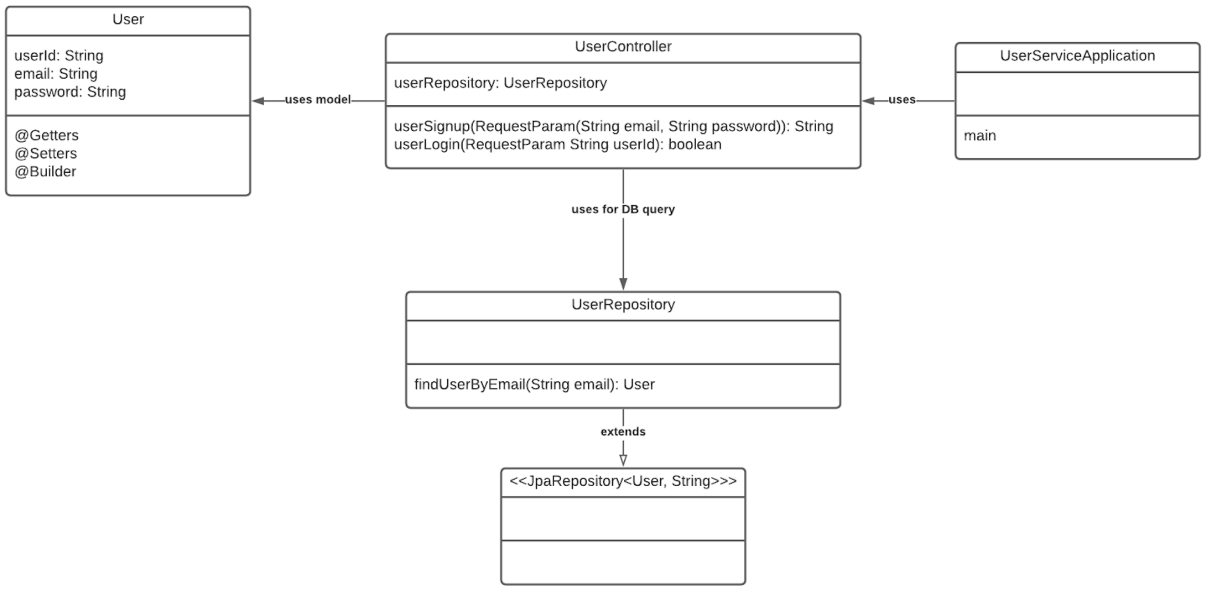
The first class diagram is going to be of the WebService microservice. This highlights an MVC pattern, as well as some builder patterns for out models. This also shows a facade pattern, since this is what the user interacts with, and the true complexity is abstracted behind this interface. You can find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

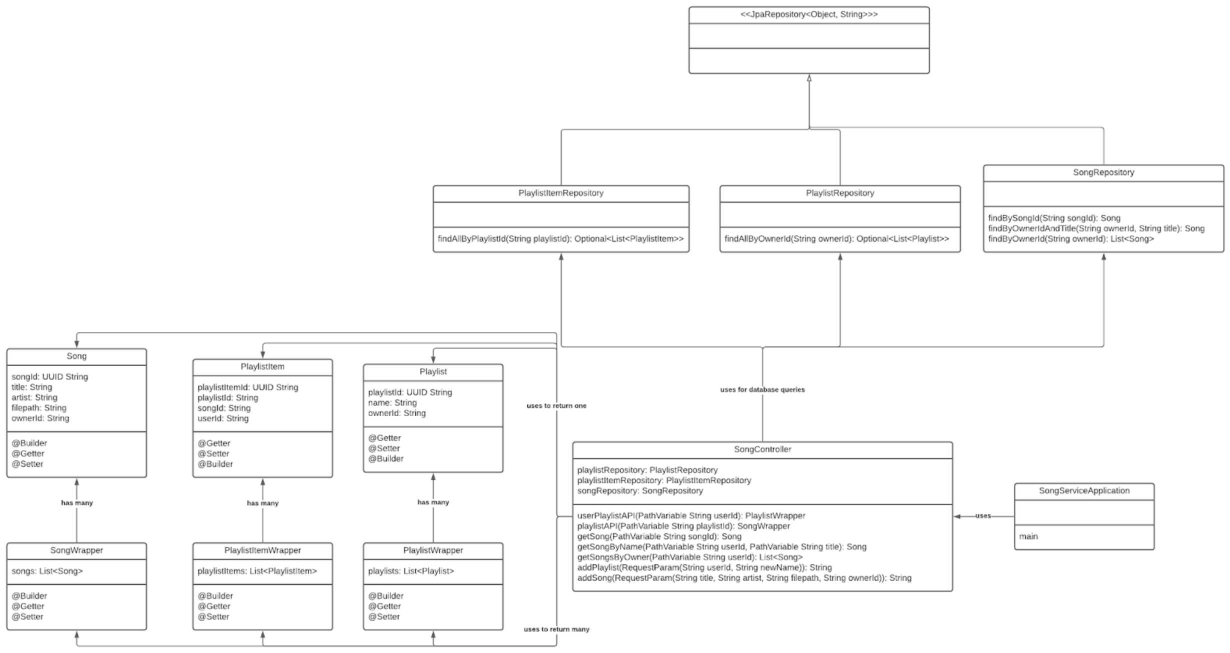
Next up is the ContentService diagram. It has a main controller that drives its function, but this time there is no view or model. This service simply accepts REST API calls to the endpoints it's listening on and returns some back to the caller. This microservice highlights an example of a factory pattern. 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. At first we had a locally running database instance, and things worked great. During project 7 we transitioned to using an AWS RDS postgres instance. The transition required almost no changes in actual code, due to JPA and JDBC abstracting our queries away.


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 has methods listed that we developed, but did not code views for. It contains ways to manage user playlists and songs. We only have the view coded for individual song management.


<<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
has many
has many
has many
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
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

Third Party Code vs Original

Each microservice started off using Spring Initializer, which is a tool built into Intellij IDEA (https://start.spring.io/). The entire application is using the Java Spring framework (https://spring.io/). We are also using other tools within the framework, such as Spring JPA (https://spring.io/projects/spring-data-jpa) and JDBC (https://spring.io/projects/spring-data-jdbc). JPA allows us to write database queries in plain Java, and JDBC allows us to connect to a database of our choosing, without having to write any actual code for it. Another tool we utilized is Lombok (https://projectlombok.org/). This tool allowed us to write entities and spring controllers in a more efficient manner. For example, instead of having to write our own builder patterns, we could simply annotate an entity with “@Builder” and Lombok would fill in the code for us. This allowed us to think about the design over the implementation. For our views we used Apache Freemarker (https://freemarker.apache.org/). This allows us to add data to a model, and then have our views load content conditionally based on that model. We used Spring Cloud for AWS (https://spring.io/projects/spring-cloud-aws) to upload content to our S3 bucket. Alex M is very good at reading pure documentation (due to coding in Ada at work for the past year and a half), and so a lot of our learning came directly from official documentation. The most helpful tutorial was a baeldung tutorial (https://www.baeldung.com/spring-file-upload) on spring file uploading, specifically how to pass the file from our WebController to our ContentController. Lastly, we utilized bootstrap for styling our views (https://getbootstrap.com/).

All of the code that you can see in our application is original. We didn’t copy and paste from stack overflow, or any other source. All of the code that isn’t original is hidden from us behind Java Spring somewhere.

Statement on OOAD processes

The biggest issue that we faced was learning an entirely new tech stack. Oftentimes software engineers tell themselves that design is the most important part, and that implementation is almost a side venture. But when the implementations aren’t trivial, and you have to research almost everything you’re doing, it tends to get in the way of designing a good system. For us, we found that we designed without knowledge of the implementations, so our idea of good design didn’t always reflect the actual best practice. For example, we initially designed our database accesses in an odd way, but later went and redesigned things when we learned about JPA and JDBC. Things such as changing factory patterns to builders, what classes we created, etc. We found that knowing how the framework we were working with worked allowed us to design and create a better system. So overtime we understood what exactly we were making better. I think if we knew from the beginning how Java Spring worked we would have been able to accomplish 10x the amount of work. On the bright side, we got to spend a lot of time pair programming and learning about best practices within Spring. Making everything a microservice early on gave us the freedom to code in different styles as we learned new things.