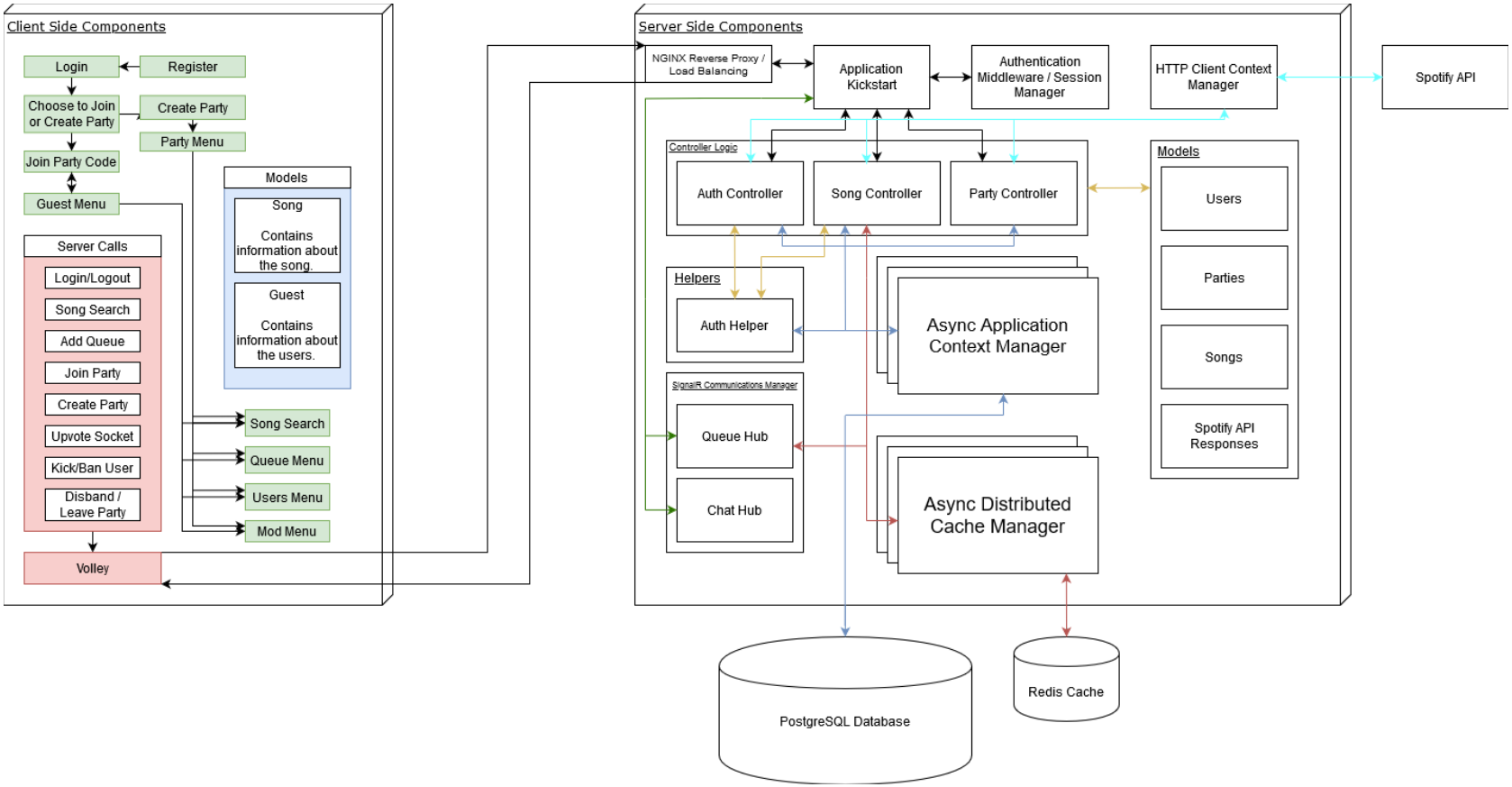
SelecTunes

JR_2

Joshua Edwards, Jack Goldsworth, Nathan Tucker, Alexander Young


Client Side Components
Login
Register
Choose to Join or Create Party
Create Party
Party Menu
Join Party Code
Guest Menu
Server Calls
Login/Logout
Song Search
Add Queue
Join Party
Create Party
Upvote Socket
Kick/Ban User
Disband / Leave Party
Models
Song
Contains information about the song.
Guest
Contains information about the users.
Song Search
Queue Menu
Users Menu
Mod Menu
Volley
Server Side Components
NGINX Reverse Proxy / Load Balancing
Application Kickstart
Authentication Middleware / Session Manager
HTTP Client Context Manager
Spotify API
Controller Logic
Auth Controller
Song Controller
Party Controller
Models
Users
Parties
Songs
Spotify API Responses
Helpers
Auth Helper
Async Application Context Manager
SignalR Communications Manager
Queue Hub
Chat Hub
Async Distributed Cache Manager
PostgreSQL Database
Redis Cache

Server Side

The requests are taken in initially by an NGINX reverse proxy, which is sent to our Application. Each request to the controllers is first passed through our Authentication Middleware, and each request must include a valid Session Cookie to be used by the application.

Once passed to the controllers, each request endpoint is managed by its respective controller, Auth, Song, and Party respectively. Each of these controllers call the Distributed Cache, which is a Redis cache to hold the song queue for each party, as well as the Application Context, the layer to interact with our PostgreSQL database holding the users, parties, and other parts of the application. Each query to the database is used to add Users, create Parties, as well as manage the two. As stated before, the Redis Cache is used to manage the song queue, and the server must be able to add songs, remove songs, and "lock in" songs, depending on its upvotes.

We have our own helper code as well to clean up the controllers, as shown by the "Helpers" block. This includes an authentication helper that updates the spotify token, abstracts the code away for banning and kicking, and updates the user tokens, if they do indeed expire. It does this by calling the Spotify API to update the Auth tokens and make it able to search

The last component of the app is the SignalR websocket component. To offer a live view of the queue as songs are being added, removed, and locked in, we have a websocket endpoint to view. Additionally, there is a chat endpoint to be added into a chat with all other party members.

Client Side:

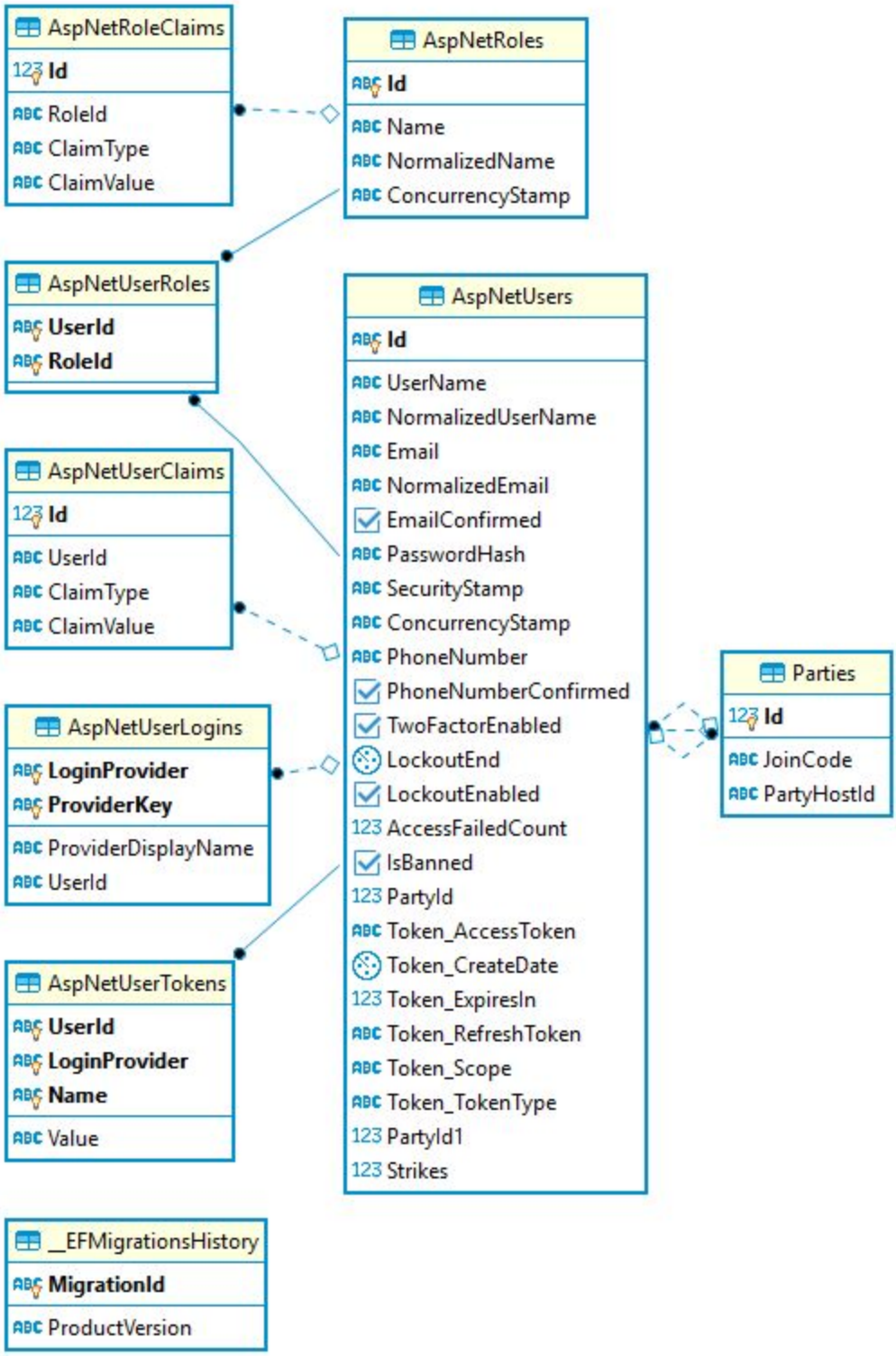
Controller:

Upon logging in, the server identifies each person as either a 'host', 'moderator', or 'guest'. Which determines which requests can be made and what changes can be enacted upon the party. Each time anyone upvotes, or downvotes a song, the server pings everyone in the party to carry out the change client-side, other things that utilize client-server communication include the song search feature, song queue view, guest list view, moderator view, which each make a JSON request of one type or another to the server which responds with the requested information and allows it to be displayed on the users screen. The song search is taken care of by the Spotify API, and allows for simple song searching that is formatted in an easy way to deserialize.

View:

Our app's front end implementation is solely for Android, so we decided to go the route of XML for layouts and styling, with Kotlin logic behind the scenes. The front end has code written to login to spotify with the credentials as a host, after that to create a "party", and many other menu options to view the current song queue, guest list, moderators, ect. The guest side on the other hand, allows guests to view the song queue, add songs to the queue, and the ability to vote songs up or down. In the background, there is code in place to correctly display song names, artists, and album arts for both the queue and the songs that are searched to be added to the queue.

Tables


AspNetRoleClaims
Id
RoleId
ClaimType
ClaimValue
AspNetRoles
Id
Name
NormalizedName
ConcurrencyStamp
AspNetUserRoles
UserId
RoleId
AspNetUsers
Id
UserName
NormalizedUserName
Email
NormalizedEmail
EmailConfirmed
PasswordHash
SecurityStamp
ConcurrencyStamp
PhoneNumber
PhoneNumberConfirmed
TwoFactorEnabled
LockoutEnd
LockoutEnabled
AccessFailedCount
IsBanned
PartyId
Token_AccessToken
Token_CreateDate
Token_ExpiresIn
Token_RefreshToken
Token_Scope
Token_TokenType
PartyId1
Strikes
AspNetUserClaims
Id
UserId
ClaimType
ClaimValue
Parties
Id
JoinCode
PartyHostId
AspNetUserLogins
LoginProvider
ProviderKey
ProviderDisplayName
UserId
AspNetUserTokens
UserId
LoginProvider
Name
Value
__EFMigrationsHistory
MigrationId
ProductVersion