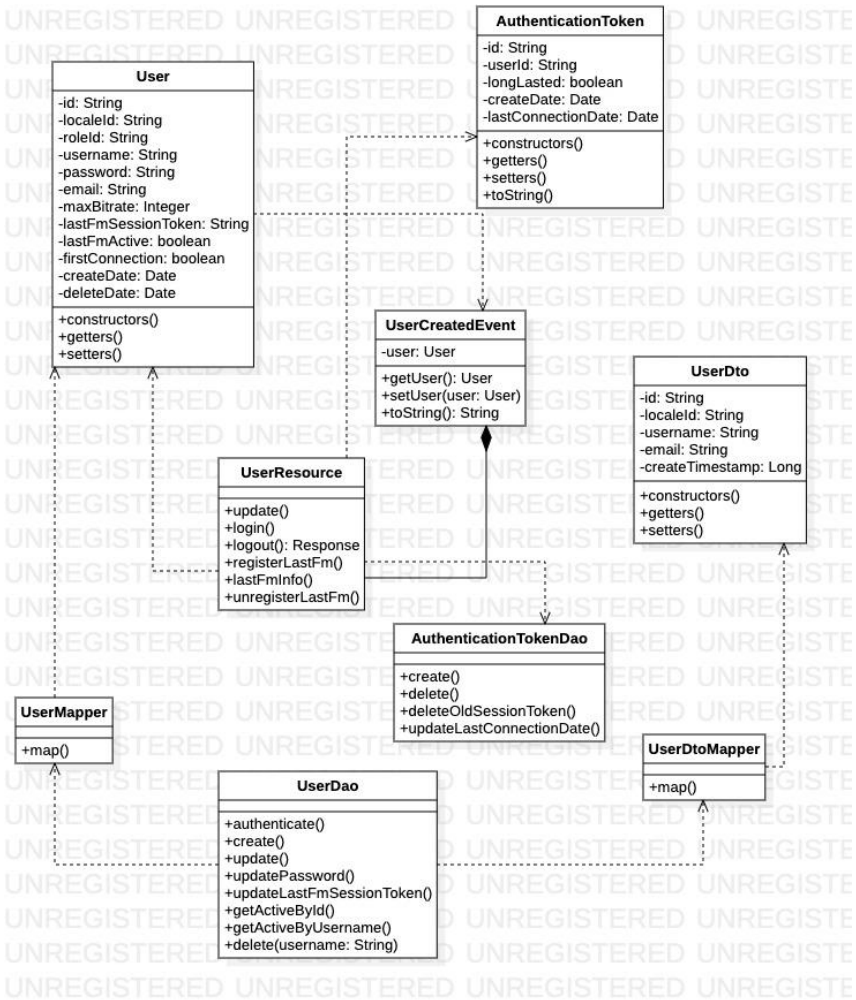
Project 1

TEAM 22
Dishant Sharma - 2022202019
Harshit Kashyap - 2022201050
Shubham Deshmukh - 2022201076
Udrasht Pal - 2022201020
Utkarsh Pathak - 2022201018

Task 1 - Mining the repository:

Created UMls and documented their functionality and behaviors.

USER MANAGEMENT



User: It's a user entity, which is using get and set functions to get and set the details of the user.

User Resource : This class keeps track of user REST resources
Resource in this is used to authenticate the user and create the user session and return the user a response(json) containing required info

Update: this update the user information
Login: this is to identify the user
Logout: this logouts the user and deletes the active session
registerLastfm: this function authenticates the user on last.fm
lastfmInfo: this gives the last.fm information about the connected user
unregisteredLastfm: Disconnects the current user from last.fm

User Dao: Its class for User Data Access object creation which further connects to the database for storing the information on it.

Authenticate: This authenticates an user
Create: this creates a new user
Update: this updates a user information
UpdatePassword: this is used to update the user password
UpdateLastFmSessionToken: this is where the user Last.fm session token is updated
getActiveById: it is used for getting the user by its ID
getActiveByUsername: It is used for getting the user by its username
getActiveByPassword: this is used to get an active user by ist password token
Delete: this deletes the user
hashPassword: this is where the password hash is calculated

User Mapper: This class maps the user details and user results to other classes.

UserCreated Events: this is where an event is raised after the creation of a user.
getUser: Get the user in the session
setUser: sets the user and the current user.
toString(): details are converted to string as required further in the process.

UserDtoMapper: This class maps the user details that are to be transferred to other classes.

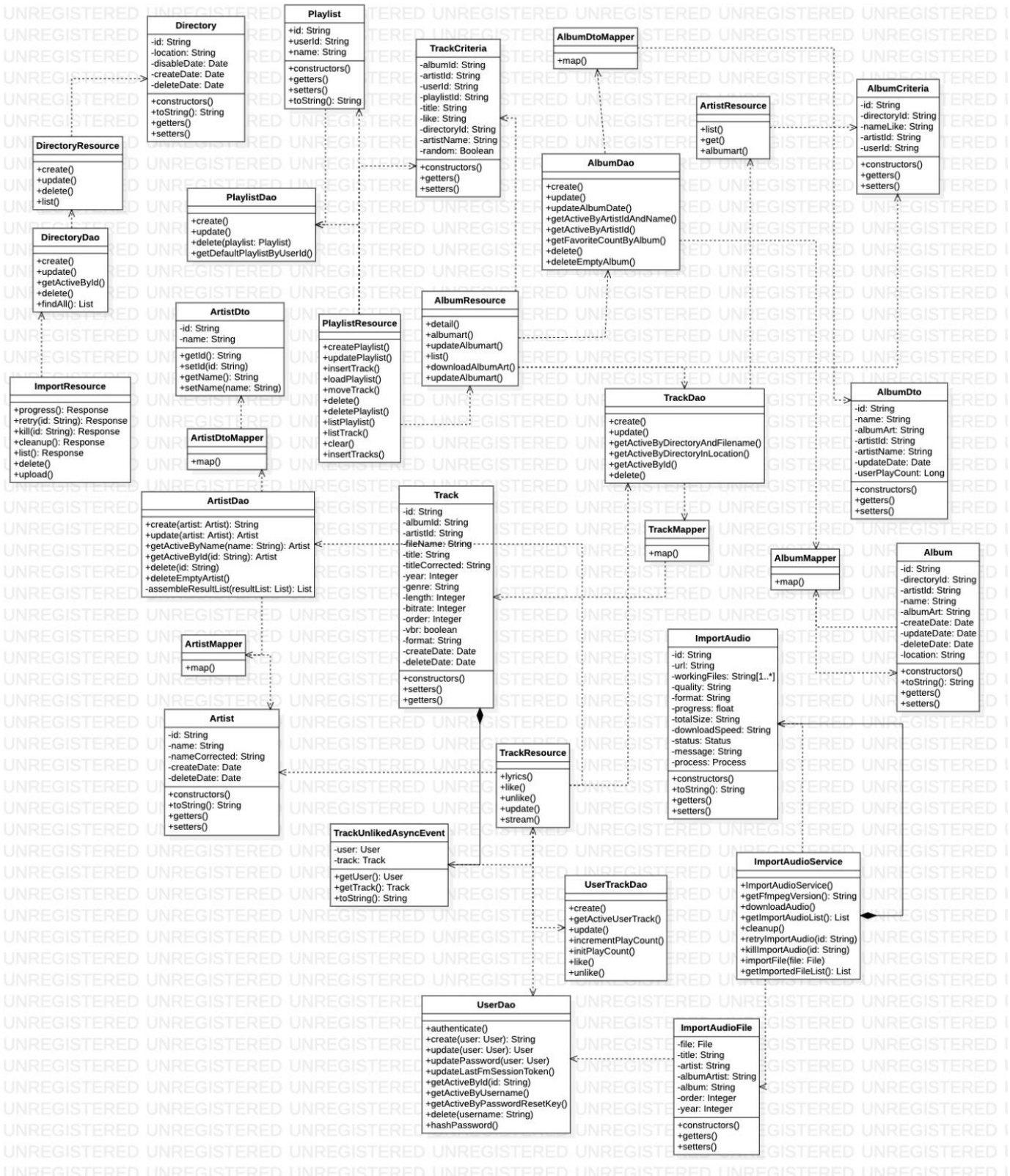
UserDto: This is the class where a data transfer object is created which is accessed by other classes as per their requirements. This class uses constructor, getter and setter functions to do this.

Authentication Token: It's a class where the entity for the authentication token is created by using get and set functions getting and setting the details as asked.

Authentication Token Dao: It's a class where a Data Object for the authentication token is created and the authentication details are then saved to the database.

Getid: this takes the id of the authentication token and returns the authentication token.
Create: this creates a new authentication token
Delete: this deletes the authentication token
deleteOldSessionToken: this deletes the short lived tokens
updateLastConnectionDate: this renews the last date when connection was established by the last authentication token

LIBRARY MANAGEMENT



Track: It's a Track entity, which is using get and set functions to get and set the details of the track for the user.

TrackResources: This class keeps track of track REST resources and return the response(json) containing required info

Lyrics: adds the lyrics in the resources
Like: add the like to the current track
Unlike: it is used to add an unlike to the current track
Update: it is used to update the details of the track.
Stream: it is where the stream to the track is returned as response(json) taking the id of the track.

TrackAsyncEvent: It capture the event for liking and unliking the track

getUser: It is used to get the user
getTrack: It is used to get the track
toString: This is used to convert details into string as per required further

TrackDao: Its class for Track Data Access object creation which further connects to the database for storing the information on it.

Create: This is used to create the new track
Update: This updates the selected track
getActiveByDirectoryAndFilename: this returns the active track by taking the file name as input
getActiveByDirectoryInLocation: this returns the active track by taking the parent directory location as input
getActiveById: this returns the active track by taking the track ID as input
Delete: this deletes the track.

TrackMapper: This class maps the track details and track results to other classes.

TrackCriteria: This class takes the data about the album, playlist, directory, artist and sets the information for the track by get and set functions.

UserTrackDao: Its class for User Track Data Access object creation which further connects to the database for storing the information on it.

Create: It creates a new user track as required
getActiveUserTrack: It is used to get the active user track or if not found it will create it if necessary
Update: It updates the userTrack
incrementPlayCount: Increments the number of times the track was played.
initPlayCount: It is used to initialize the play count for a track
Like: this is where the like is set for a track
Unlike:this is where the unlike is set for the track

User Dao: Its class for User Data Access object i created which connected to the database for storing the information on it.

Authenticate: This authenticates an user
Create: this creates a new user
Update: this updates a user information

UpdatePassword: this is used to update the user password
UpdateLastFmSessionToken: this is where the user Last.fm session token is updated
getActiveById: it is used for getting the user by its ID
getActiveByUsername: It is used for getting the user by its username
getActiveByPassword: this is used to get an active user by ist password token
Delete: this deletes the user
hashPassword: this is where the password hash is calculated

ImportAudio: This is the class where audio import progress is taken care of from the given plugins. It is done by taking the audio data and setting the information of the Audio by get and set functions.

ImportAudioService: This is the class where the Audio service is used which allows the audio to be imported from various sources and maintain their patterns and status.

getFfmpegVersion: this returns the version of ffmpeg and if not found it returns null.
downloadAudio: This function adds the URL to import from the external sources.
getImportAudioList: this returns the copied version of currently importing audio from external sources.
Cleanup: it cleans up the already imported audio from external sources.
retryImportAudio: this helps to retry a failed import from an external sources
killImportAudio: this function kill the importing audio from the the external sources
importFile: this is where the main files are imported, that can be zip of single track file.
getImportedFileList: this returns the list of imported files
Tagfile: this method tags the file and move it into the directory as per required by user

ImportAudioFile: This class is where the audio file import progress is taken care of and setting the file information in the album by getting the required information by get functions and set the values as required by set functions.

ImportResource: This class keeps track of Import REST resources and returns the response(json) containing required info.

Progress: this return a response containing the progress of the all the imports from external sources
Retry: this retries a failed import from the external sources
Kill: this kill the import from from the external sources
List: this list down the imported tracks, and suggest some tags
Delete: this deletes the imported file.
Upload: this upload a track to the imported file

Album: It's a Album entity, which is using get and set functions to get and set the details of the Album for the user.

AlbumResource: This class keeps track of album REST resources and return the response(json) containing required info
Detail: this returns the album details as response(json).
Albumart: this takes album Id and cover size as input and returns album cover as response(json)
updateAlbumart: this downloads the album cover from the URL and updates the cover.
List: this returns the list of active albums
updateAlbumArt: this downloads the album

AlbumDao:Its class for Album Data Access object creation which further connects to the database for storing the information on it.

Create: this creates a new album and returns album ID
Update: this takes the album and returns the updated album
updateAlbumDate: this updates the album date
getActiveByArtistIdAndName: this is used for getting the active album by its name
getActiveByArtist: this is used for getting the active album by artist ID
getFavoriteCountByAlbum: this returns the total number of total number of likes for the Album
Delete: This deletes the album
deleteEmptyAlbum: this deletes the album that don't have any tracks

AlbumDto:This is the class where an Album transfer object is created which is accessed by other classes as per their requirements. This class uses constructor, getter and setter functions to do this.

AlbumDtoMapper: This class maps the Album details that are to be transferred to other classes.

AlbumCriteria: This class takes the data about the directory, artist and sets the information for the track by get and set functions.

AlbumMapper: This class maps the Album details and Album results to other classes.

Artist: It's an Artist entity, which is using get and set functions to get and set the details of the Artist like name,Id,date etc..

ArtistDao: Its class for Artist Data Access object creation which further connects to the database for storing the information on it.

Create: this creates a new artist
Update: this takes the artist and returns the updated artist
getActiveByName:this is used for getting the active artist by its name
getActiveById: this is used for getting the active artist by ID
Delete: this deletes the artist by taking the artist Id as input
deleteEmptyArtist: this delete any artist that don't have any album or track
assembleResultList: this is used to assemble the query results.(no usage)

ArtistDtoMapper: This class maps the Artist details that are to be transferred to other classes.

ArtistDto: This is the class where an Artist transfer object is created which is accessed by other classes as per their requirements. This class uses constructor, getter and setter functions to do this.

Playlist: It's a playlist entity, which is either the list of current played tracks or a saved playlist with a name. It uses get and set functions to get and set the details of the playlist.

PlaylistDao: Its class for Playlist Data Access object creation which further connects to the database for storing the information on it.

Create: this creates a new playlist
Update: this updates the playlist
Delete: this deletes the playlist
getDefaultPlaylistByUserId: this is used for getting the playlist by user ID

PlaylistResource: This class keeps track of Playlist REST resources and returns the response(json) containing required info.

createPlaylist: this is used to create a named playlist by taking the name and returns response(json)
updatePlaylist: this takes the name, updates a named playlist and returns the response
insertPlaylist: this is used to insert a track in the playlist by taking playlist ID, track ID, order and returns response.(no usage)
loadPlaylist: this load a named playlist into the default playlist(nousage)
moveTrack: this is used to move the track to another position in the playlist
Delete: this removes a track from the playlist
deletePlaylist: this delete the named playlist
listyPlaylist: this returns all the named playlists
listTrack: this returns the all the tracks in the playlist
Clear: this removes all the tracks from the playlist
insertTracks: this inserts a track in the playlist

Directory: It's a directory entity, which is using get details and set functions to set the details in the directory as required.

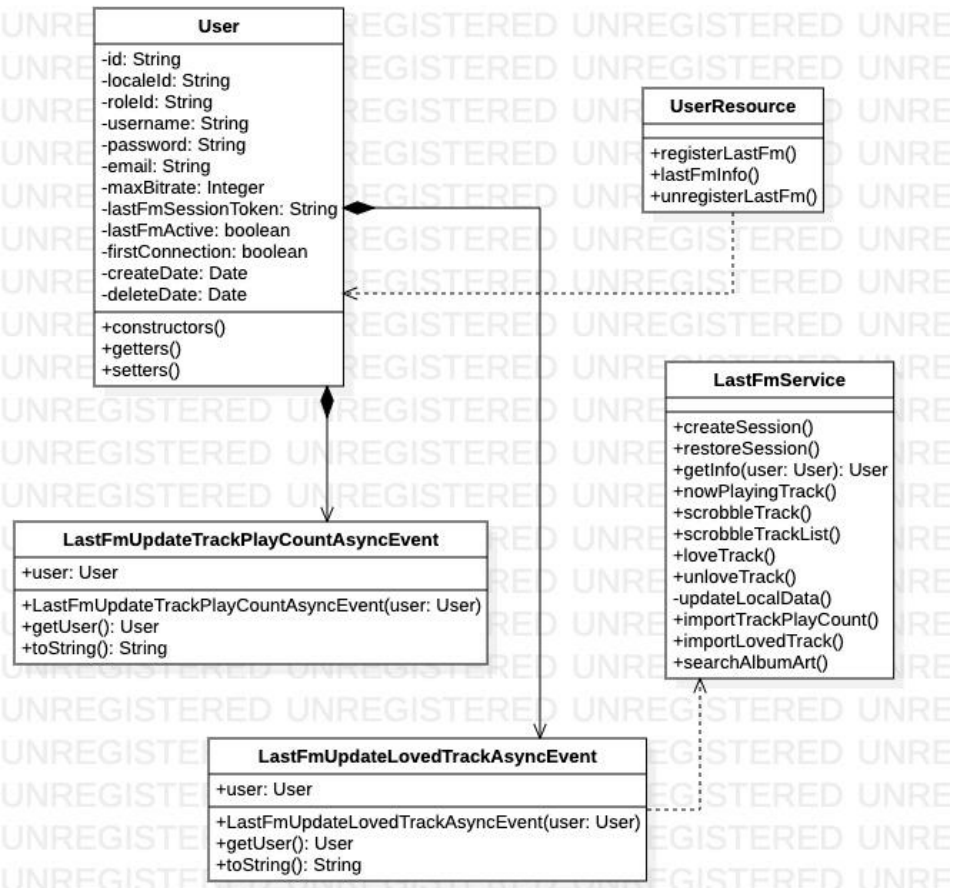
DirectoryResource: This class keeps track of Directory REST resources and returns the response(json) containing required info.

Create: this creates a new directory
Update: this updates the directory informations
Delete: this deletes the directory by taking directory id as input
List: this returns all the active directories

DirectoryDao: Its class for Directory Data Access object creation which further connects to the database for storing the information on it.

Create: this creates a new directory
Update: this updates a directory
getActiveById: this is used for getting the active directory by its ID
Delete: this delete the directory by taking Directory ID as input
findAll: this returns the list of all the directories

LAST.FM INTEGRATION



LastFmService:
This class provide services that helps to integrate the environment for creating a user session and handle the tracks and their information

Createsession: this creates a new user session
restoreSession: this restores the session from the stored session token taking user object as input and returning Last.fm session
getInfo: return the last.fm information
nowPlayingTrack: this updates the current playing track
scrobbleTrack: this scrobbles a track
scrobbleTrackList: this scrobbles a list of track
loveTrack: this is used like a rating giving the user to know the track is loved
unloveTrack:this is used like a rating giving the user to know the track is not loved
updateLocalData: this updates the local data from Last.fm corrected labels
importTrackPlayCount: this imports all track play counts from last.fm
ImportLovedTrack: this imports all the loved tracks from last.fm
searchAlbumArt: this search the album arts

LastFmUpdateTrackPlayCountAsyncEvent: this event is invoked to update the play count info, number of times the track is played. This is done by a constructor and get function

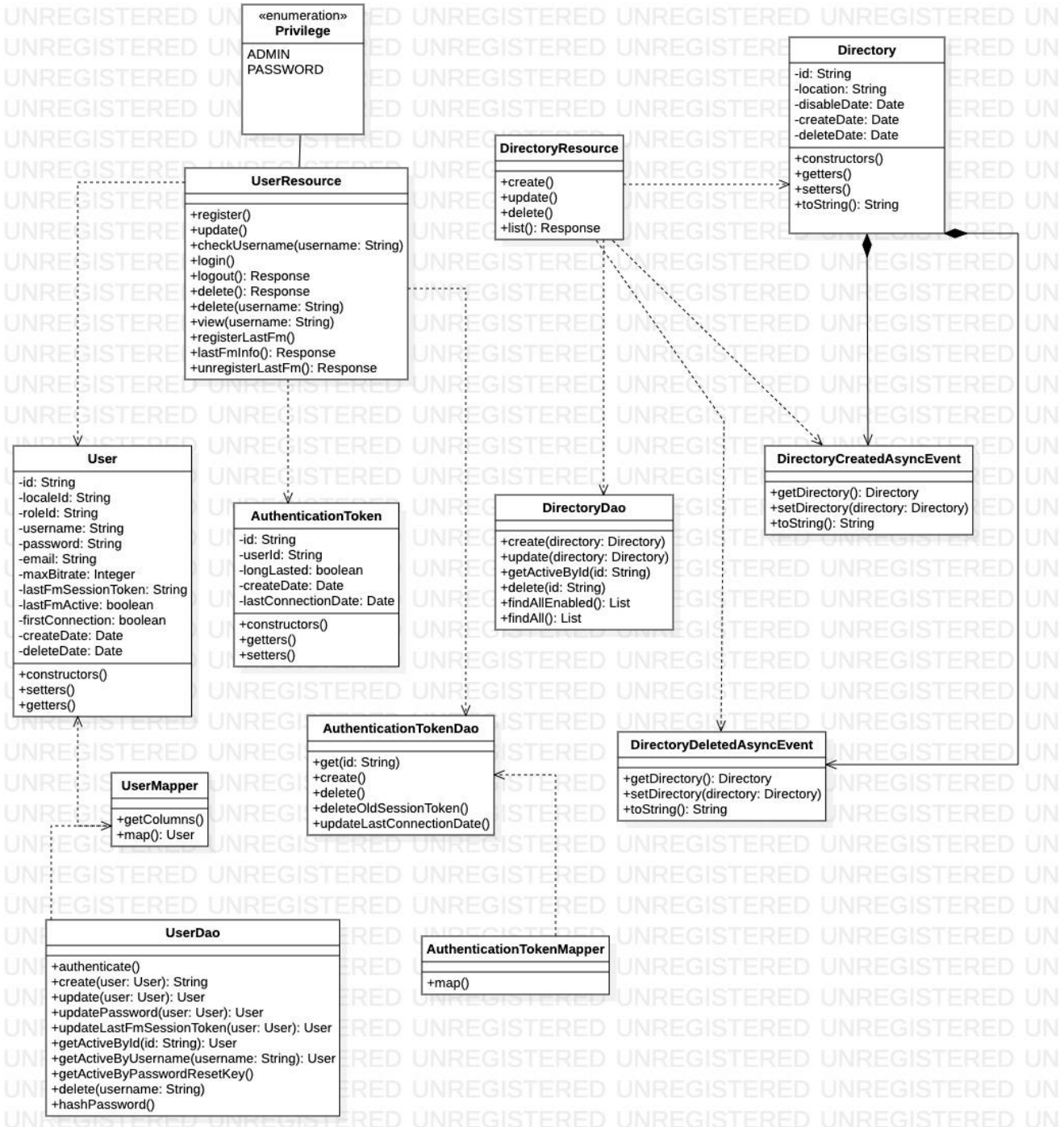
LastFmUpadteLovedTrackAsyncEvent: this event is invoked to update track-loved info,how much the track is being loved. This is done by a constructor and get function

User: It's a user entity, which is using get and set functions to get and set the details of the user.

User Resource : This class keeps track of user REST resources
Resource in this is used to authenticate the user and create the user session and return the user a response(json) containing required info

Update: this update the user information
Login: this is to identify the user
Logout: this logouts the user and deletes the active session
registerLastfm: this function authenticates the user on last.fm
lastfmInfo: this gives the last.fm information about the connected user
unregisteredLastfm: Disconnects the current user from last.fm

Administrator Features



User: It's a user entity, which is using get and set functions to get and set the details of the user.

User Resource : This class keeps track of user REST resources

Resource in this is used to authenticate the user and create the user session and return the user a response(json) containing required info

Update: this update the user information
Login: this is to identify the user
Logout: this logouts the user and deletes the active session
registerLastfm: this function authenticates the user on last.fm
lastfmInfo: this gives the last.fm information about the connected user
unregisteredLastfm: Disconnects the current user from last.fm

User Dao: Its class for User Data Access object creation which further connects to the database for storing the information on it.

Authenticate: This authenticates an user
Create: this creates a new user
Update: this updates a user information
UpdatePassword: this is used to update the user password
UpdateLastFmSessionToken: this is where the user Last.fm session token is updated
getActiveById: it is used for getting the user by its ID
getActiveByUsername: It is used for getting the user by its username
getActiveByPassword: this is used to get an active user by ist password token
Delete: this deletes the user
hashPassword: this is where the password hash is calculated

User Mapper: This class maps the user details and user results to other classes.

Authentication Token: It's a class where the entity for the authentication token is created by using get and set functions getting and setting the details as asked.

Authentication Token Dao: It's a class where a Data Object for the authentication token is created and the authentication details are then saved to the database.

Getid: this takes the id of the authentication token and returns the authentication token.
Create: this creates a new authentication token
Delete: this deletes the authentication token
deleteOldSessionToken: this deletes the short lived tokens
updateLastConnectionDate: this renews the last date when connection was established by the last authentication token

AuthenticationTokenMapper: This class maps the Authentication Token details and Authentication token results to other classes.

DBIF: this is the DBI factory where different mappers of classes are registered so that they can be connected and pass information as required by another class.
createDbi: here the mapper are created and registered
DBIF: this a private constructor(no usage).
Get: this is used to get the instance of the DBIF
Reset: this resets and drops all the objects in DBIF

Privilege(class): this class is where the admin are given privileges, such as directory controls, user management etc..
getId: this to get the Id of the privileged user
setId: this is to set the Id of the privileged user

Priviledge(enum): this allows the user to use the admin functions and allow the user to change his password

PrivilegeMapper: This class maps the Privilege details and Privilege results to other classes.

Directory: It's a directory entity, which is using get details and set functions to set the details in the directory as required.

DirectoryResource: This class keeps track of Directory REST resources and returns the response(json) containing required info.

Create: this creates a new directory
Update: this updates the directory informations
Delete: this deletes the directory by taking directory id as input
List: this returns all the active directories

DirectoryDao: Its class for Directory Data Access object creation which further connects to the database for storing the information on it.

Create: this creates a new directory
Update: this updates a directory
getActiveById: this is used for getting the active directory by its ID
Delete: this delete the directory by taking Directory ID as input
findAll: this returns the list of all the directories

DirectoryMapper: This class maps the Directory details and Directory results to other classes.

DirectoryCreatedAsyncEvent: this is an event which is invoked when a directory is created, this includes get and set functions for getting and setting the details of the Directory.

DirectoryDeletedAsyncEvent :this is an event which is invoked when a directory is deleted, this includes get and set functions for getting and setting the details of the Directory.

Strength and Weakness of System:

Strengths:

Functionality: A well-designed system should be able to perform its intended functions reliably and efficiently, meeting the needs of its users.

Observation1 : All the functions in the classes are well defined giving a clear view to the user what a function is expected to do.
Observation2 : A Good UI is built for playing and importing the track as it can abstract metadata of the track while adding on its own and it also shows the user play count and how much the track is loved, with other good features of the music player.

Security: A system with robust security measures can help protect against unauthorized access, data breaches, and other security threats.
Observation1 : All the access controls are done by the Admin, no regular user can create an account, directory, import anything unless he/she is authorized by the Admin.
Observation2: Server logs can only be viewed by the Admin

Compatibility: A system that is compatible with other software applications and platforms can improve interoperability and enhance overall usability.

Observation1 : files can be imported from external sources like youtube, personal computers or different URLs.
Observation2: many files can be imported as zip or a single track can also be imported

Weakness:

Complexity: A complex system can be difficult to design, build, and maintain, leading to increased costs, longer development times, and higher risk of errors.

Observation1: A simple task for adding a track is too complex, too many steps have to be followed for adding it.
Observation2: Lot of classes are being used and many of them have no dedicated usage in the system.

Performance: A poorly optimized system can suffer from slow performance and long response times, frustrating users and potentially leading to lower adoption rates.

Observation1: Not a User friendly UI, Too much excessive tasks may result in the user to have low interest in using the system and may result in leaving it easily.

Reliability: A system that is prone to errors and crashes can be unreliable, leading to lost data, downtime, and user frustration.
Observation1: If the Admin deletes the directory, no info is given to other users and hence leading to loss of the user data without his concern, making the system less reliable.

Assumptions:

1. There are many classes that have long get, set functions with constructors which are just represented by their names in the UML diagrams.

2. There are many classes like Base Resource, Base Dao, Base result set etc.. which implements inheritance that are not included in the UMLs. Since, we are only dealing with their subclasses in each subsystem.

TASK 2

DESIGN SMELLS

- Unfactored hierarchy and Missing hierarchy

1. **TrackLikedAsyncListener** and **TrackUnlikedAsyncListener** are performing similar type of function which is related to Track. There could be many more similar functionality in future related to track for which new classes will be made. We can solve this issue by making an interface and extending these classes from that interface.

```
public class TrackUnlikedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(TrackUnlikedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param trackUnlikedAsyncEvent New directory created event
     */
    @Subscribe
    public void onTrackLiked(final TrackUnlikedAsyncEvent trackUnlikedAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Track unliked event: " + trackUnlikedAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final User user = trackUnlikedAsyncEvent.getUser();
        final Track track = trackUnlikedAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
        if (user.getLastFmSessionToken() != null) {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.unloveTrack(user, track);
        }
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Track unliked completed in {0}", stopwatch));
        }
    }
}
```

```java
public class TrackLikedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(TrackLikedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param trackLikedAsyncEvent New directory created event
     */
    @Subscribe
    public void onTrackLiked(final TrackLikedAsyncEvent trackLikedAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Track liked event: " + trackLikedAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final User user = trackLikedAsyncEvent.getUser();
        final Track track = trackLikedAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
        if (user.getLastFmSessionToken() != null) {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.loveTrack(user, track);
        }
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Track liked completed in {0}", stopwatch));
        }
    }
}
```

2. **PlayCompletedAsyncListener** and **PlayStartedAsyncListener** are performing similar type of function which is related to playing of track. There could be many more similar functionality related to track for which new classes will be made. We can solve this issue by making an interface and extending these classes from that interface.

```java
public class PlayCompletedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(PlayCompletedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param playCompletedEvent Play completed event
     */
    @Subscribe
    public void onPlayCompleted(final PlayCompletedEvent playCompletedEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Play completed event: " + playCompletedEvent.toString());
        }

        final String userId = playCompletedEvent.getUserId();
        final Track track = playCompletedEvent.getTrack();

        TransactionUtil.handle(() -> {
            // Increment the play count
            UserTrackDao userTrackDao = new UserTrackDao();
            userTrackDao.incrementPlayCount(userId, track.getId());

            final User user = new UserDao().getActiveById(userId);
            if (user != null && user.getLastFmSessionToken() != null) {
                final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
                lastFmService.scrobbleTrack(user, track);
            }
        });
    }
}
```

```java
 */
public class PlayStartedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(PlayStartedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param playStartedEvent Play started event
     */
    @Subscribe
    public void onPlayStarted(final PlayStartedEvent playStartedEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Play started event: " + playStartedEvent.toString());
        }

        final String userId = playStartedEvent.getUserId();
        final Track track = playStartedEvent.getTrack();

        TransactionUtil.handle(() -> {
            final User user = new UserDao().getActiveById(userId);
            if (user != null && user.getLastFmSessionToken() != null) {
                final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
                lastFmService.nowPlayingTrack(user, track);
            }
        });
    }
}
```

3. **LastFmUpdateLovedTrackAsyncListener** and **LastFmUpdateTrackPlayCountAsyncListener** are performing similar type of function which is related to lastfm.There could be many more similar functionality related to track for which new classes will be made. We can solve this issue by making an interface and extending these classes from that interface.

```java
public class LastFmUpdateLovedTrackAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(LastFmUpdateLovedTrackAsyncListener.class);

    /**
     * Process the event.
     *
     * @param lastFmUpdateLovedTrackAsyncEvent Update loved track event
     */
    @Subscribe
    public void onLastFmUpdateLovedTrack(final LastFmUpdateLovedTrackAsyncEvent lastFmUpdateLovedTrackAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Last.fm update loved track event: " + lastFmUpdateLovedTrackAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final User user = lastFmUpdateLovedTrackAsyncEvent.getUser();

        TransactionUtil.handle(() -> {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.importLovedTrack(user);
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Last.fm update loved track event completed in {0}", stopwatch));
        }
    }
}
```

```java
public class LastFmUpdateTrackPlayCountAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(LastFmUpdateTrackPlayCountAsyncListener.class);

    /**
     * Process the event.
     *
     * @param lastFmUpdateTrackPlayCountAsyncEvent Update track play count event
     */
    @Subscribe
    public void onLastFmUpdateTrackPlayCount(final LastFmUpdateTrackPlayCountAsyncEvent lastFmUpdateTrackPlayCountAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Last.fm update track play count event: " + lastFmUpdateTrackPlayCountAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final User user = lastFmUpdateTrackPlayCountAsyncEvent.getUser();

        TransactionUtil.handle(() -> {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.importTrackPlayCount(user);
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Last.fm update track play count event completed in {0}", stopwatch));
        }
    }
}
```

4. **DirectoryCreatedAsyncListener** and **DirectoryDeletedAsyncListener** are performing similar type of function which is related to directory.There could be many more similar functionality related to track for which new classes will be made. We can solve this issue by making an interface and extending these classes from that interface.

```java
public class DirectoryDeletedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(DirectoryDeletedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param directoryDeletedAsyncEvent New directory deleted event
     */
    @Subscribe
    public void onDirectoryDeleted(final DirectoryDeletedAsyncEvent directoryDeletedAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Directory deleted event: " + directoryDeletedAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final Directory directory = directoryDeletedAsyncEvent.getDirectory();

        TransactionUtil.handle(() -> {
            // Stop watching the directory
            AppContext.getInstance().getCollectionWatchService().unwatchDirectory(directory);

            // Remove directory from index
            CollectionService collectionService = AppContext.getInstance().getCollectionService();
            collectionService.removeDirectoryFromIndex(directory);
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Collection updated in {0}ms", stopwatch));
        }
    }
}
```

```java
public class DirectoryDeletedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(DirectoryDeletedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param directoryDeletedAsyncEvent New directory deleted event
     */
    @Subscribe
    public void onDirectoryDeleted(final DirectoryDeletedAsyncEvent directoryDeletedAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Directory deleted event: " + directoryDeletedAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final Directory directory = directoryDeletedAsyncEvent.getDirectory();

        TransactionUtil.handle(() -> {
            // Stop watching the directory
            AppContext.getInstance().getCollectionWatchService().unwatchDirectory(directory);

            // Remove directory from index
            CollectionService collectionService = AppContext.getInstance().getCollectionService();
            collectionService.removeDirectoryFromIndex(directory);
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Collection updated in {0}ms", stopwatch));
        }
    }
}
```

- Imperative Abstraction

1. **DirectoryCreatedAsyncEvent** and **DirectoryDeletedAsyncEvent** have same attributes and behaviour. There are multiple classes unecessary for similar functionality. We can have one common class for both the classes.

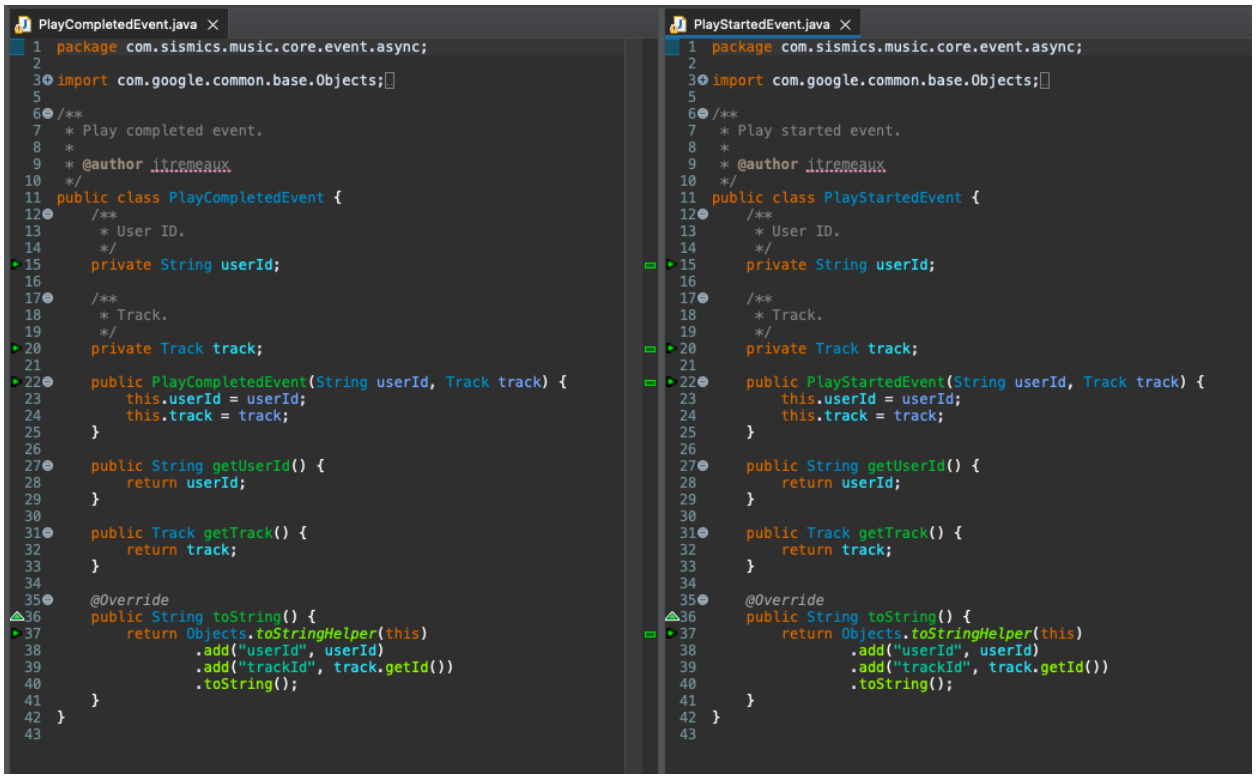
DirectoryCreatedAsyncEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * New directory created event.
 *
 * @author jtremeaux
 */
public class DirectoryCreatedAsyncEvent {
    /**
     * New directory.
     */
    private Directory directory;

    /**
     * Getter of directory.
     *
     * @return directory
     */
    public Directory getDirectory() {
        return directory;
    }

    /**
     * Setter of directory.
     *
     * @param directory directory
     */
    public void setDirectory(Directory directory) {
        this.directory = directory;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("directory", directory)
                .toString();
    }
}
```

DirectoryDeletedAsyncEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Directory deleted event.
 *
 * @author jtremeaux
 */
public class DirectoryDeletedAsyncEvent {
    /**
     * New directory.
     */
    private Directory directory;

    /**
     * Getter of directory.
     *
     * @return directory
     */
    public Directory getDirectory() {
        return directory;
    }

    /**
     * Setter of directory.
     *
     * @param directory directory
     */
    public void setDirectory(Directory directory) {
        this.directory = directory;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("directory", directory)
                .toString();
    }
}
```

2. **PlayCompletedEvent** and **PlayStartedEvent** have same attributes and behaviour. There are multiple classes unecessary for similar functionality. We can have one common class for both the classes.



PlayCompletedEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Play completed event.
 *
 * @author jtremeaux
 */
public class PlayCompletedEvent {
    /**
     * User ID.
     */
    private String userId;

    /**
     * Track.
     */
    private Track track;

    public PlayCompletedEvent(String userId, Track track) {
        this.userId = userId;
        this.track = track;
    }

    public String getUserId() {
        return userId;
    }

    public Track getTrack() {
        return track;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("userId", userId)
                .add("trackId", track.getId())
                .toString();
    }
}
```

PlayStartedEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Play started event.
 *
 * @author jtremeaux
 */
public class PlayStartedEvent {
    /**
     * User ID.
     */
    private String userId;

    /**
     * Track.
     */
    private Track track;

    public PlayStartedEvent(String userId, Track track) {
        this.userId = userId;
        this.track = track;
    }

    public String getUserId() {
        return userId;
    }

    public Track getTrack() {
        return track;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("userId", userId)
                .add("trackId", track.getId())
                .toString();
    }
}
```

3. **LastFmUpdateLovedTrackAsyncEvent** and **LastFmUpdateTrackPlayCountAsyncEvent** have same attributes and behaviour. There are multiple classes unecessary for similar functionality. We can have one common class for both the classes.

LastFmUpdateLovedTrackAsyncEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Last.fm update loved tracks event.
 *
 * @author itremeaux
 */
public class LastFmUpdateLovedTrackAsyncEvent {
    /**
     * User.
     */
    private User user;

    public LastFmUpdateLovedTrackAsyncEvent(User user) {
        this.user = user;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .toString();
    }
}
```

LastFmUpdateTrackPlayCountAsyncEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Last.fm update track play count event.
 *
 * @author itremeaux
 */
public class LastFmUpdateTrackPlayCountAsyncEvent {
    /**
     * User.
     */
    private User user;

    public LastFmUpdateTrackPlayCountAsyncEvent(User user) {
        this.user = user;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .toString();
    }
}
```

4. **TrackLikedAsyncEvent** and **TrackUnlikedAsyncEvent** have same attributes and behaviour. There are multiple classes unecessary for similar functionality. We can have one common class for both the classes.

TrackLikedAsyncEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Track liked event.
 *
 * @author itremeaux
 */
public class TrackLikedAsyncEvent {
    /**
     * Originating user.
     */
    private User user;

    /**
     * Liked track.
     */
    private Track track;

    public TrackLikedAsyncEvent(User user, Track track) {
        this.user = user;
        this.track = track;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    /**
     * Getter of track.
     *
     * @return track
     */
    public Track getTrack() {
        return track;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .add("track", track)
                .toString();
    }
}
```

TrackUnlikedAsyncEvent.java

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Track unliked event.
 *
 * @author itremeaux
 */
public class TrackUnlikedAsyncEvent {
    /**
     * Originating user.
     */
    private User user;

    /**
     * Liked track.
     */
    private Track track;

    public TrackUnlikedAsyncEvent(User user, Track track) {
        this.user = user;
        this.track = track;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    /**
     * Getter of track.
     *
     * @return track
     */
    public Track getTrack() {
        return track;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .add("track", track)
                .toString();
    }
}
```

- Repeated code of Logger
  Logger Services are used before and after a transaction for many listener events. Therefore there is a lot of repeated code.

```
*/
public class TrackLikedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(TrackLikedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param trackLikedAsyncEvent New directory created event
     */
    @Subscribe
    public void onTrackLiked(final TrackLikedAsyncEvent trackLikedAsyncEvent) throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Track liked event: " + trackLikedAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final User user = trackLikedAsyncEvent.getUser();
        final Track track = trackLikedAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
        if (user.getLastFmSessionToken() != null) {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.loveTrack(user, track);
        }
        });

        if (log.isInfoEnabled()) {
            log.info(MessageFormat.format("Track liked completed in {0}", stopwatch));
        }
    }
}
```

  - Dead code and reference of code

music-core/src/main/java/com/sismics/music/core/service/collection/CollectionService.java

```
302     public void updateScore() {
303 //      AlbumDao albumDao = new AlbumDao();
304 //      List<AlbumDto> albumList = albumDao.findByCriteria(new AlbumCriteria());
305 //      for (AlbumDto albumDto : albumList) {
306 //          Integer score = albumDao.getFavoriteCountByAlbum(albumDto.getId());
307 //
308 //          Album album = new Album();
309 //          album.setId(albumDto.getId());
310 //          album.setScore(score);
311 //
312 //          albumDao.updateScore(album);
313 //      }
314     }
315 302 }
```

music-core/src/main/java/com/sismics/music/core/listener/async/DirectoryCreatedAsyncListener.java

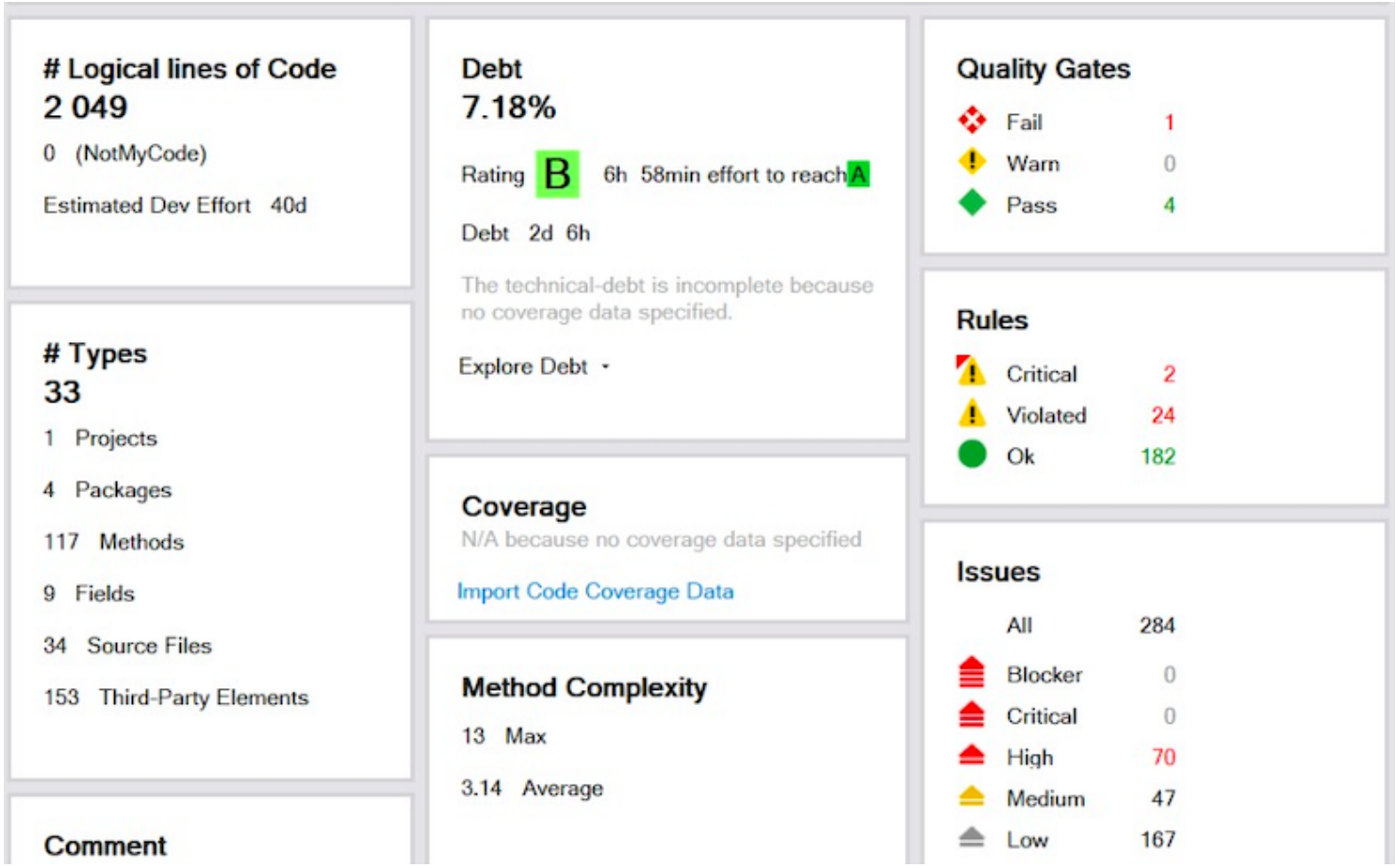
```
45 45            // Watch new directory
46 46            AppContext.getInstance().getCollectionWatchService().watchDirectory(directory);
47 47
48               // Update the scores
49               collectionService.updateScore();
50 48        });
51 49
```

music-core/src/main/java/com/sismics/music/core/listener/async/CollectionReindexAsyncListener.java

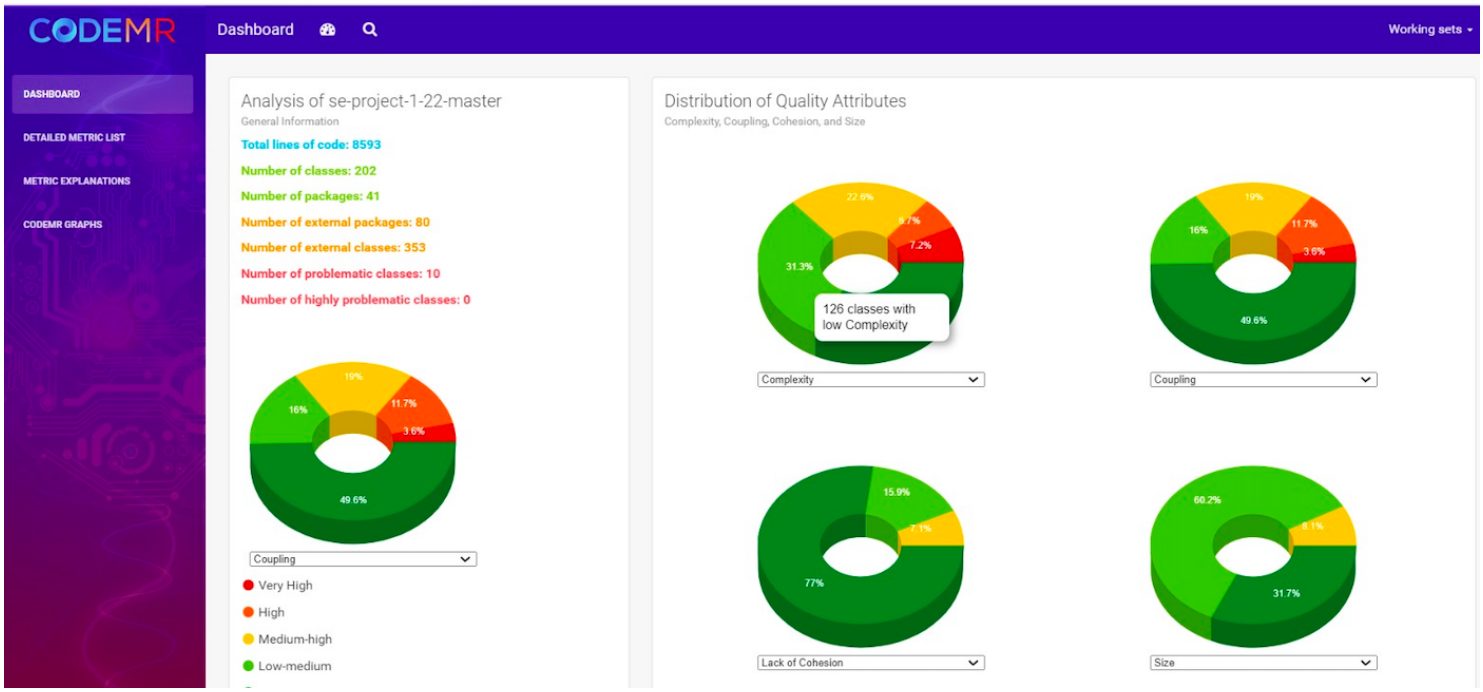
```
   @@ -39,8 +39,7 @@ public class CollectionReindexAsyncListener {
39 39            CollectionService collectionService = AppContext.getInstance().getCollectionService();
40 40            collectionService.reindex();
41 41
42               // Update the scores
43               collectionService.updateScore();
   42
44 43        });
45 44
46 45        if (log.isInfoEnabled()) {
```

CODE METRICS

Code Metrics for music-web before refactoring



CODE MR ANALYSIS

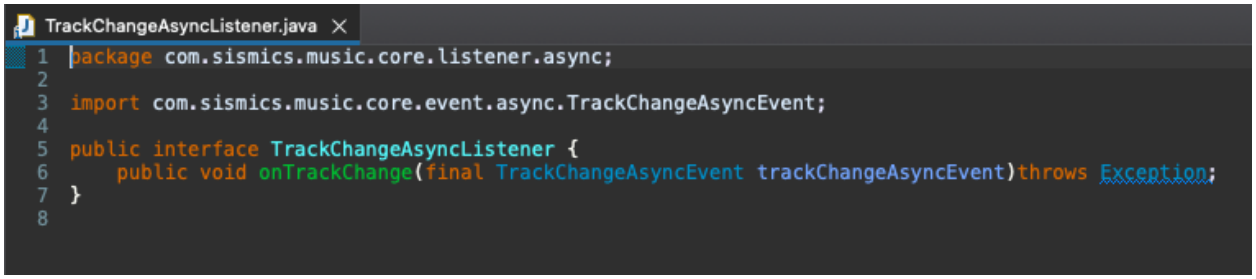


TASK 3

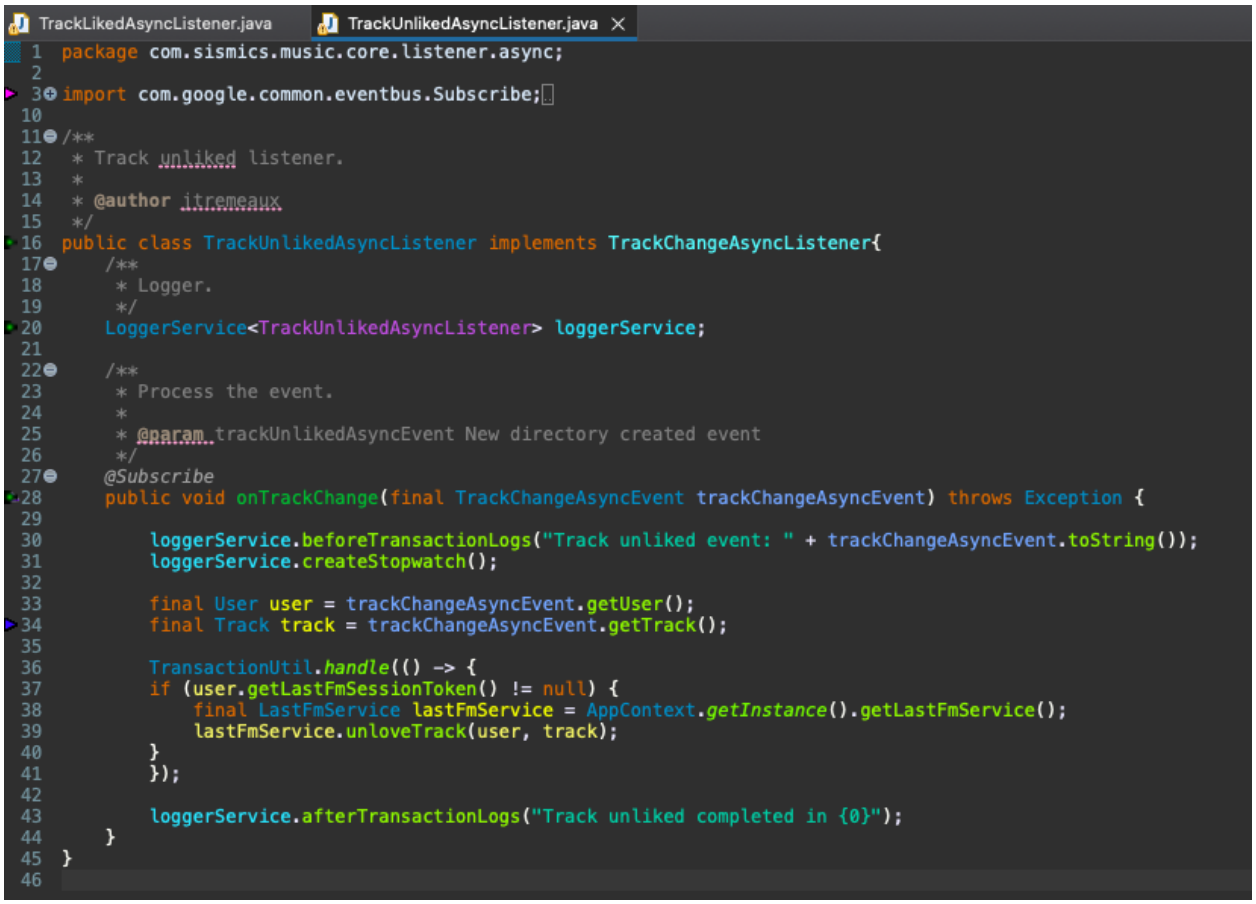
REFACTORING CODE SMELLS

- Unfactored hierarchy and Missing hierarchy

To resolve this issue , created an interface **TrackChanegAsyncListener** related to Track and all the classes which implement this interface will be implementing the function in them. For example



```java
package com.sismics.music.core.listener.async;

import com.sismics.music.core.event.async.TrackChangeAsyncEvent;

public interface TrackChangeAsyncListener {
    public void onTrackChange(final TrackChangeAsyncEvent trackChangeAsyncEvent)throws Exception;
}
```

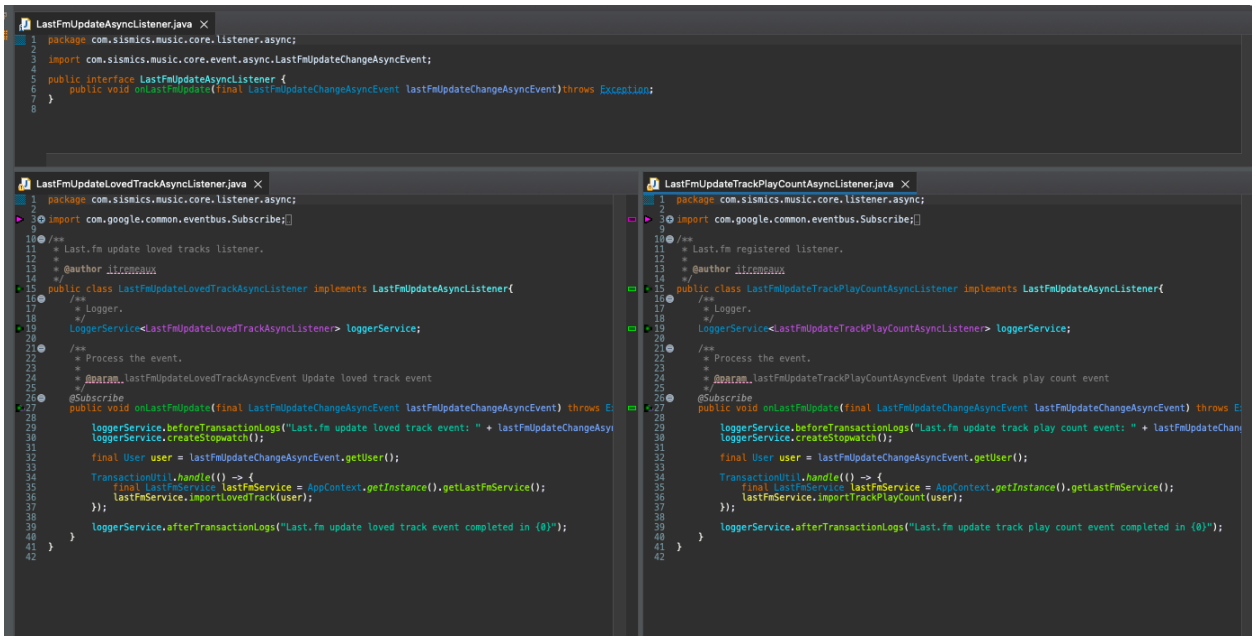


```java
package com.sismics.music.core.listener.async;

import com.google.common.eventbus.Subscribe;

/**
 * Track unliked listener.
 *
 * @author itremeaux
 */
public class TrackUnlikedAsyncListener implements TrackChangeAsyncListener{
    /**
     * Logger.
     */
    LoggerService<TrackUnlikedAsyncListener> loggerService;

    /**
     * Process the event.
     *
     * @param trackUnlikedAsyncEvent New directory created event
     */
    @Subscribe
    public void onTrackChange(final TrackChangeAsyncEvent trackChangeAsyncEvent) throws Exception {

        loggerService.beforeTransactionLogs("Track unliked event: " + trackChangeAsyncEvent.toString());
        loggerService.createStopwatch();

        final User user = trackChangeAsyncEvent.getUser();
        final Track track = trackChangeAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
        if (user.getLastFmSessionToken() != null) {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.unloveTrack(user, track);
        }
        });

        loggerService.afterTransactionLogs("Track unliked completed in {0}");
    }
}
```

TrackLikedAsyncListener.java

```java
package com.sismics.music.core.listener.async;

import com.google.common.eventbus.Subscribe;

/**
 * Track liked listener.
 *
 * @author jtremeaux
 */
public class TrackLikedAsyncListener implements TrackChangeAsyncListener {
    /**
     * Logger.
     */
    LoggerService<TrackLikedAsyncListener> loggerService;
    /**
     * Process the event.
     *
     * @param trackLikedAsyncEvent New directory created event
     */

    @Subscribe
    public void onTrackChange(final TrackChangeAsyncEvent trackChangeAsyncEvent) throws Exception {

        loggerService.beforeTransactionLogs("Track liked event: " + trackChangeAsyncEvent.toString());
        loggerService.createStopwatch();

        final User user = trackChangeAsyncEvent.getUser();
        final Track track = trackChangeAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
        if (user.getLastFmSessionToken() != null) {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.loveTrack(user, track);
        }
        });

        loggerService.afterTransactionLogs("Track liked completed in {0}");
    }
}

```

Similar type of refactoring is done for LastFm , Directory and Player respectively.



LastFmUpdateAsyncListener.java

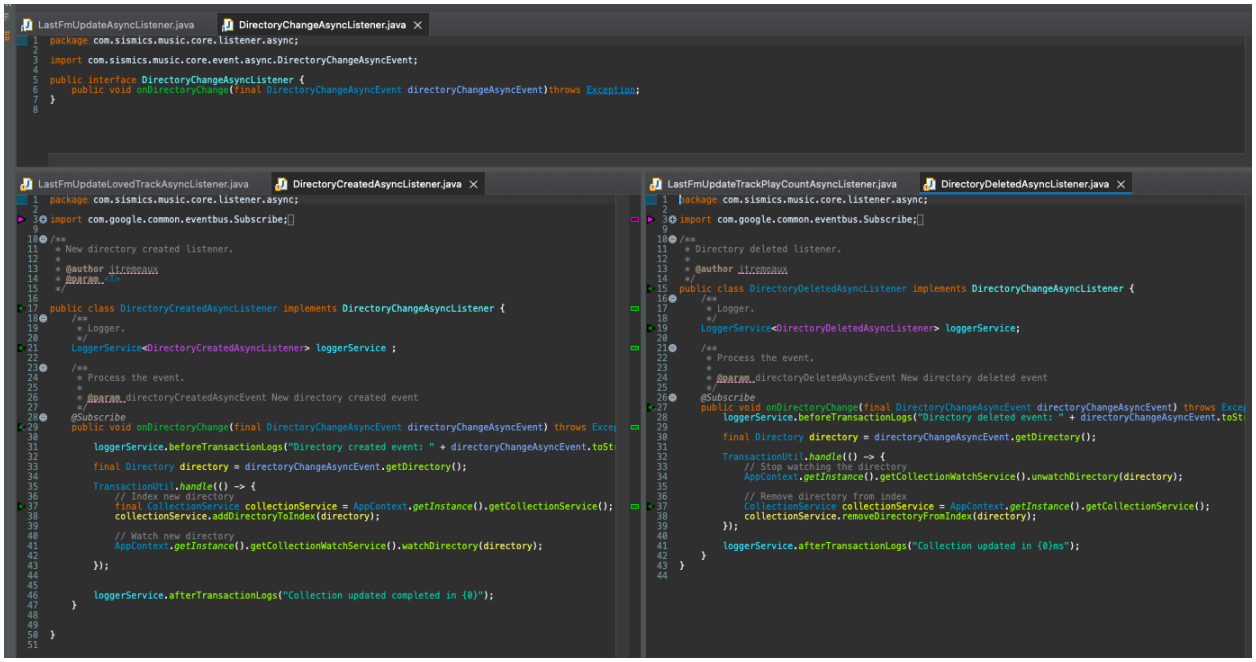
```java
package com.sismics.music.core.listener.async;

import com.sismics.music.core.event.async.LastFmUpdateChangeAsyncEvent;

public interface LastFmUpdateAsyncListener {
    public void onLastFmUpdate(final LastFmUpdateChangeAsyncEvent lastFmUpdateChangeAsyncEvent)throws Exception;
}
```

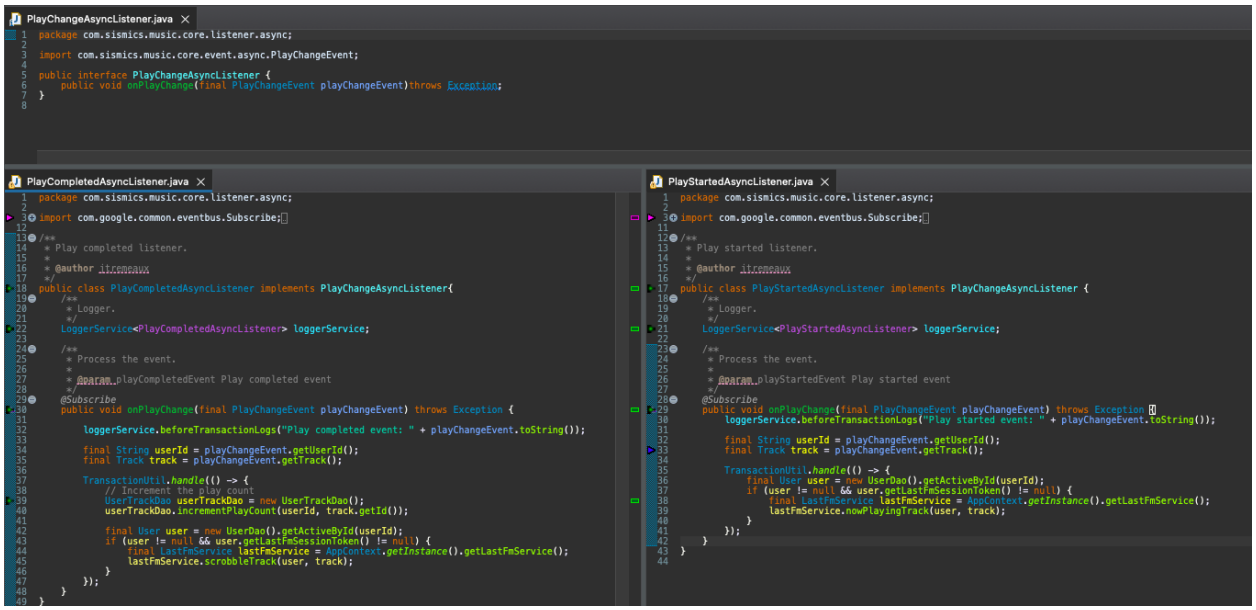
LastFmUpdateLovedTrackAsyncListener.java

```java
package com.sismics.music.core.listener.async;

import com.google.common.eventbus.Subscribe;

/**
 * Last.fm update loved tracks listener.
 *
 * @author jtremeaux
 */
public class LastFmUpdateLovedTrackAsyncListener implements LastFmUpdateAsyncListener{
    /**
     * Logger.
     */
    LoggerService<LastFmUpdateLovedTrackAsyncListener> loggerService;

    /**
     * Process the event.
     *
     * @param lastFmUpdateLovedTrackAsyncEvent Update loved track event
     */
    @Subscribe
    public void onLastFmUpdate(final LastFmUpdateChangeAsyncEvent lastFmUpdateChangeAsyncEvent) throws E

        loggerService.beforeTransactionLogs("Last.fm update loved track event: " + lastFmUpdateChangeAsy
        loggerService.createStopwatch();

        final User user = lastFmUpdateChangeAsyncEvent.getUser();

        TransactionUtil.handle(() -> {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.importLovedTrack(user);
        });

        loggerService.afterTransactionLogs("Last.fm update loved track event completed in {0}");
    }
}
```

LastFmUpdateTrackPlayCountAsyncListener.java

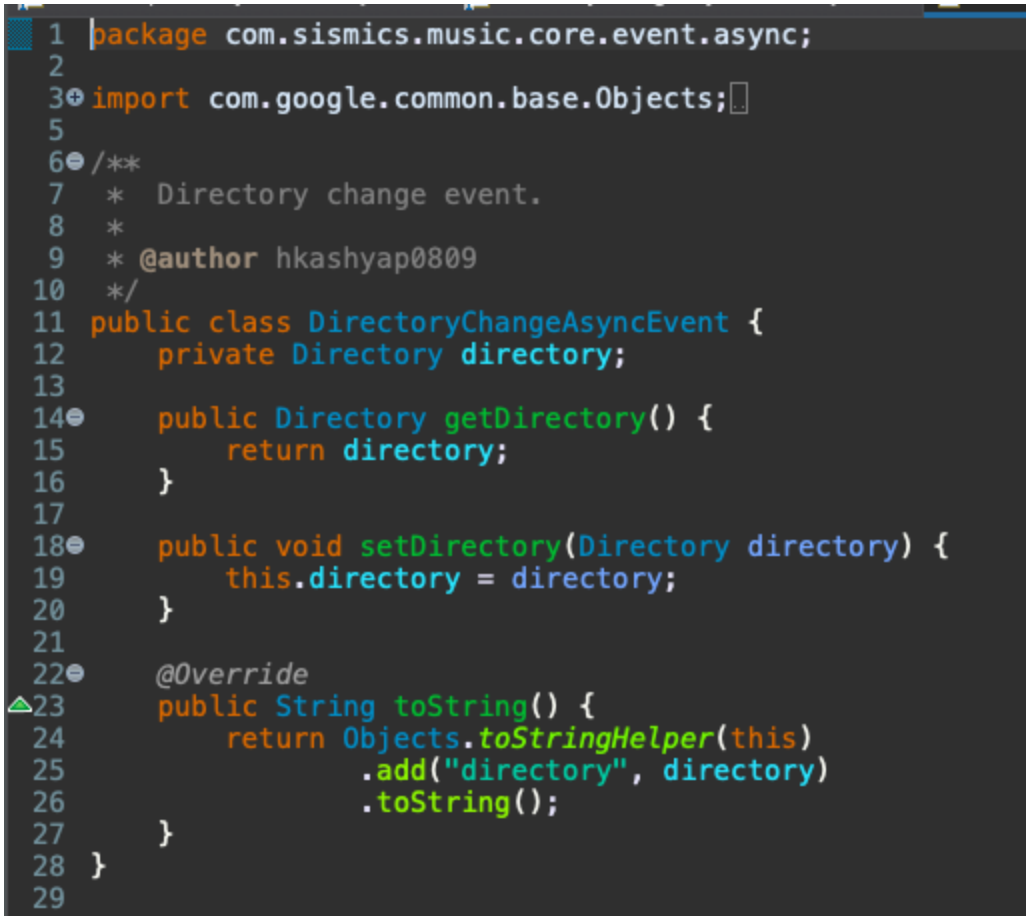
```java
package com.sismics.music.core.listener.async;

import com.google.common.eventbus.Subscribe;

/**
 * Last.fm registered listener.
 *
 * @author jtremeaux
 */
public class LastFmUpdateTrackPlayCountAsyncListener implements LastFmUpdateAsyncListener{
    /**
     * Logger.
     */
    LoggerService<LastFmUpdateTrackPlayCountAsyncListener> loggerService;

    /**
     * Process the event.
     *
     * @param lastFmUpdateTrackPlayCountAsyncEvent Update track play count event
     */
    @Subscribe
    public void onLastFmUpdate(final LastFmUpdateChangeAsyncEvent lastFmUpdateChangeAsyncEvent) throws E

        loggerService.beforeTransactionLogs("Last.fm update track play count event: " + lastFmUpdateChan
        loggerService.createStopwatch();

        final User user = lastFmUpdateChangeAsyncEvent.getUser();

        TransactionUtil.handle(() -> {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.importTrackPlayCount(user);
        });

        loggerService.afterTransactionLogs("Last.fm update track play count event completed in {0}");
    }
}
```





- **Imperative Abstraction**

1. To resolve this issue created one single class (**DirectoryChangeAsyncEvent**) for both the event classes **DirectoryCreatedAsyncEvent** and **DirectoryDeletedAsyncEvent.**



```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 *  Directory change event.
 *
 * @author hkashyap0809
 */
public class DirectoryChangeAsyncEvent {
    private Directory directory;

    public Directory getDirectory() {
        return directory;
    }

    public void setDirectory(Directory directory) {
        this.directory = directory;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("directory", directory)
                .toString();
    }
}

```

Similar kind of refactoring is done in another classes

- **PlayChangeEvent** for **PlayCompletedEvent** and **PlayStartedEvent**

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Play change event.
 *
 * @author hkashyap0809
 */
public class PlayChangeEvent {
    /**
     * User ID.
     */
    private String userId;

    /**
     * Track.
     */
    private Track track;

    public PlayChangeEvent(String userId, Track track) {
        this.userId = userId;
        this.track = track;
    }

    public String getUserId() {
        return userId;
    }

    public Track getTrack() {
        return track;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("userId", userId)
                .add("trackId", track.getId())
                .toString();
    }
}

```

- **LastFmUpdateChangeAsyncEvent** for **LastFmUpdateLovedTrackAsyncEvent** and **LastFmUpdateTrackPlayCountAsyncEvent**

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;

/**
 * Last.fm update loved tracks event.
 *
 * @author hkashyap0809
 */
public class LastFmUpdateChangeAsyncEvent {
    /**
     * User.
     */
    private User user;

    public LastFmUpdateChangeAsyncEvent(User user) {
        this.user = user;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .toString();
    }

}
```

- **TrackChangeAsyncEvent** for **TrackLikedAsyncEvent** and **TrackUnlikedAsyncEvent**

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;


/**
 * Track change event.
 *
 * @author hkashyap0809
 */
public class TrackChangeAsyncEvent {
    /**
     * Originating user.
     */
    private User user;

    /**
     * Liked track.
     */
    private Track track;

    public TrackChangeAsyncEvent(User user, Track track) {
        this.user = user;
        this.track = track;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    /**
     * Getter of track.
     *
     * @return track
     */
    public Track getTrack() {
        return track;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .add("track", track)
                .toString();
    }
}

```

- Repeated code of Logger

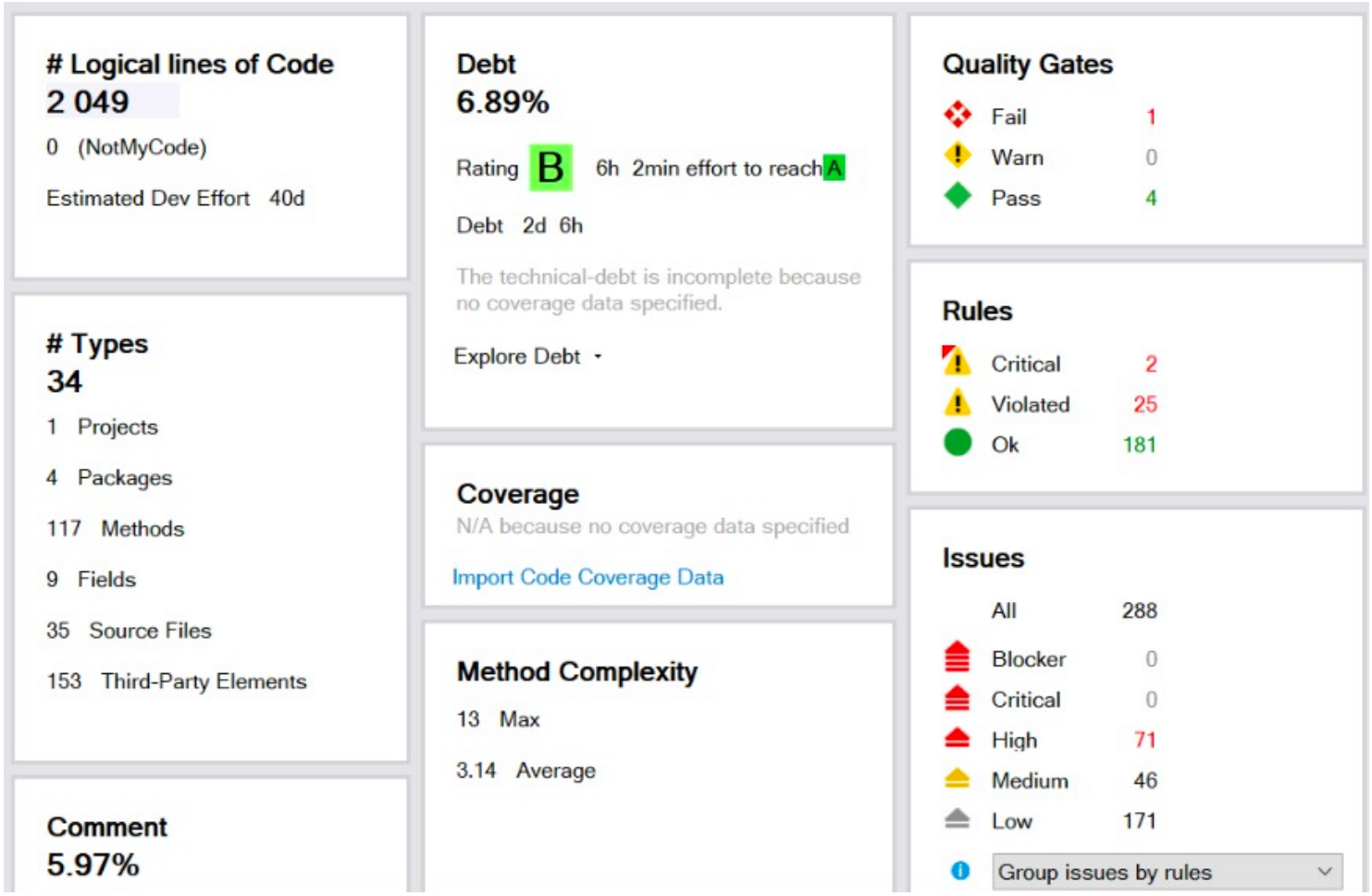
Created one logger service class to implement function for before and after transaction logs.
The functions are called whenever required.

```java
package com.sismics.music.core.listener.async;

import java.text.MessageFormat;

public class LoggerService<T> {

    private final Logger log = LoggerFactory.getLogger(this.getClass());
    private Stopwatch stopwatch;

    public void beforeTransactionLogs(String message) {
        if(log.isInfoEnabled()) {
            log.info(message);
        }

    }

    public void createStopwatch() {
        stopwatch = Stopwatch.createStarted();
    }

    public void afterTransactionLogs(String message) {
        if(log.isInfoEnabled()) {
            log.info(MessageFormat.format(message, stopwatch));
        }

    }
}

```

```java
import com.google.common.eventbus.Subscribe;

/**
 * Track liked listener.
 *
 * @author itremeaux
 */
public class TrackLikedAsyncListener implements TrackChangeAsyncListener {
    /**
     * Logger.
     */
    LoggerService<TrackLikedAsyncListener> loggerService;
    /**
     * Process the event.
     *
     * @param trackLikedAsyncEvent New directory created event
     */

    @Subscribe
    public void onTrackChange(final TrackChangeAsyncEvent trackChangeAsyncEvent) throws Exception {

        loggerService.beforeTransactionLogs("Track liked event: " + trackChangeAsyncEvent.toString());
        loggerService.createStopwatch();

        final User user = trackChangeAsyncEvent.getUser();
        final Track track = trackChangeAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
        if (user.getLastFmSessionToken() != null) {
            final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
            lastFmService.loveTrack(user, track);
        }
        });

        loggerService.afterTransactionLogs("Track liked completed in {0}");
    }
}

```

CODE METRICS AFTER REFACTORING

Code metrics after refactoring music-web



CODE MR ANALYSIS AFTER REFACTORING