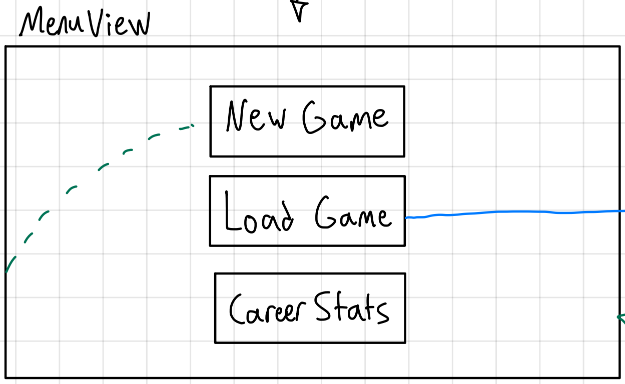
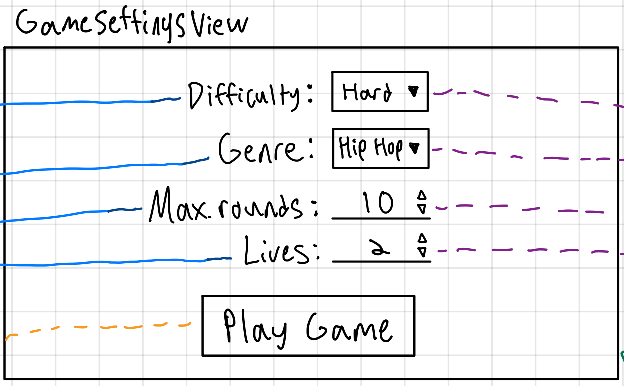
Start program



Only enabled if most recently created game is not finished (i.e. game.getFinishedDateTime() returns null)

* Note that the functionality relating to Load Game and Career Stats are not (directly) covered here. See last page for additional info.



Allowed Values

Easy (4 option multiple choice),
Medium (2 option multiple choice),
Hard (type in answer)

Hip Hop, Country, Pop, Rock

$d \in \mathbb{Z},\ 1 \le d \le 10$

$d \in \mathbb{Z},\ 1 \le d \le \min(\text{max rounds}, 10)$

* Can all be hard coded in to the view for now

GameSettingsState (for GameSettings ViewModel)

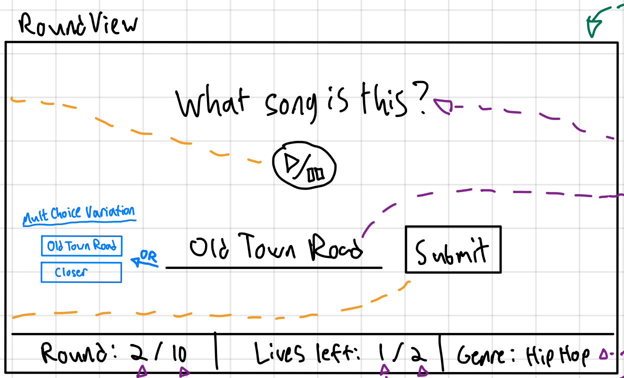
- String difficulty
- String genre
- int maxRounds
- int initialLives

updates every time user picks new value

CreateGame action

- Send game settings (i.e. contents of GameSettingsState) to interactor
- If most recent game was not finished, mark as finished (i.e. set finished datetime to now). Save with DAO.
- Create Game entity using retrieved settings.
- Create Round entity using info from Game object, inject in to game as the current round, and use DAO to save this new game.
- Update RoundViewModel (via RoundState) with game and current round info.
- Change to RoundView screen (i.e. start the game)

Make a method in RoundFactory (e.g. Round createFromGame(Game game){...})



RoundState (for Round ViewModel)

- int gameId
- String promptText
- String userAnswer
- String genre
- int initialLives
- int currentLives
- int maxRounds
- int currentRoundNumber
- List<String> multipleChoiceOptions

updates every key stroke (when typing) or every option press (when mult. choice)

If null, render single text input.
Else, display a button for each option.

ToggleAudio action

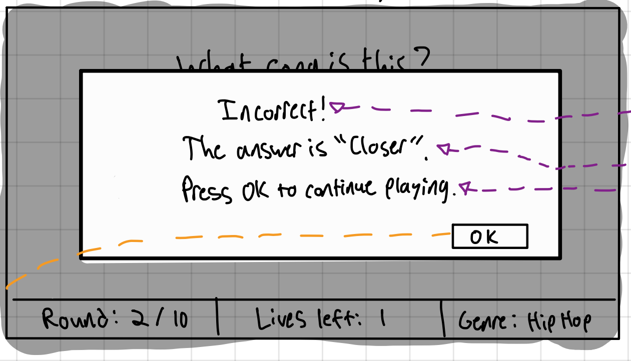
- Send gameId to interactor
- get game object from DAO using gameId
- Access current round's song audio
- If audio is playing? Stop audio. Else, play audio.

Ideally shouldn't block the main thread

SubmitAnswer action

- Send gameId and userAnswer to interactor
- get game object from DAO using gameId
- get current round from game and check if userAnswer is correct
- Is answer correct? Increase game score. Else, decrease current lives in game.
- Is game over? Mark game as finished. Else, use RoundFactory to create new Round and inject in to game as the current round.
- Save game object with DAO.
- Update PostRoundViewModel (via PostRoundState) with gameId, correct answer, correctness message (i.e. "Incorrect!" or "Correct!"), and next steps message (i.e. "Game is over! Press ok to finish game." or "Press ok to continue playing.")
- Show PostRound popup informing the user of their correctness and what happens next.

RoundView with PostRound dialog



PostRoundState (for PostRound ViewModel)

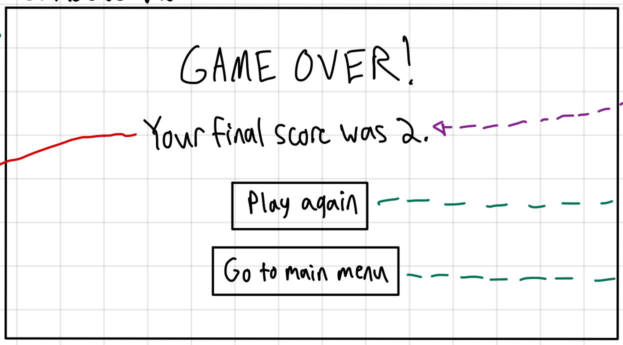
- int gameId
- String correctnessMessage
- String correctAnswer
- String nextStepsMessage

Could replace these with booleans as well and then let the view decide what to put based on their truth values

HandlePostRound action

- Send gameId to interactor
- get Game object from DAO using gameId
- Is game playing still? (i.e. game.isGameOver() == false)
  - ↳ Clear PostRoundState values and update PostRoundViewModel
  - ↳ Update RoundViewModel (via RoundState) with current round info.
  - ↳ We are already on RoundView so no need to switch screens (just make sure changes to RoundViewModel were observed)
- Else, game is over.
  - ↳ Get score from game.
  - ↳ Update GameOverViewModel (via GameOverState) with score.
  - ↳ Change to GameOverView screen

GameOverView



GameOverState (for GameOver ViewModel)

- int gameScore

We could also scrap the idea of score and say how many the user got right (e.g. instead say "You answered 2/4 questions correctly (50%).")
↳ generate statistics for GameOverState instead of score

ENTITIES

Game (interface)

getters
- int getId()
- String getDifficulty()
- String getGenre()
- int getMaxRounds()
- int getInitialLives()
- int getScore()
- int getCurrentLives()
- int getRoundsPlayedCount() → includes current round
- Round getCurrentRound()
- List<Round> getAllRounds()
- LocalDateTime getCreatedDateTime()
- LocalDateTime getFinishedDateTime()

setters
- void setCurrentLives(int lives)
- void setScore(int score)
- void setCurrentRound(Round round) → Should still keep the previous rounds intact
- void setFinishedDateTime(LocalDateTime finishedDateTime)

- boolean isGameOver()

CommonGame implements Game

Round (interface)

getters
- String getQuestion()
- Song getSong()
- String getCorrectAnswer()
- String getUserAnswer()

setters
- void setUserAnswer(String userAnswer)

- boolean isAnswerCorrect(String userAnswer)

TextInputRound implements Round
MultipleChoiceRound implements Round

Song (interface)

getters
- String getTitle()
- String getArtist()
- PlayableAudio getAudio()

CommonSong implements Song

PlayableAudio (interface)

getters
- String getPath() → i.e. preview_url or file location

- void play()
- void stop()
- boolean isPlaying()

OnlineMP3PlayableAudio implements PlayableAudio
FileMP3PlayableAudio implements PlayableAudio

Notes about Load Game Functionality

Clarifications on how the load game functionality would work in Royce's eyes:

- Prerequisites for loading:
  - ↳ The only existing game that is a candidate for being loaded is the most recently created one (i.e. game.getCreatedDateTime() is closest to the current date time).
  - ↳ This candidate game can only be loaded if it is not marked as "finished" (i.e. game.getFinishedDateTime() returns null)
- The actual loading functionality:
  - ↳ get loadable game entity from DAO using criteria listed above
  - Update RoundViewModel (via RoundState) with game and current round info
  - ↳ Change screen to the RoundView

Since we already "save" games in the actions described by the previous pages, the loading part can conceptually be as simple as the process described here.

However, we could also have it so that any unfinished game can be loadable, opposed to just the most recent one. This would be slightly more involved since we would need a view displaying all of the loadable games and then we load the one that the user chooses. Saying that, this is definetly doable if we have time.

Notes about Statistics (Career Stats) functionality

We should sit down together and better define what statistics we want to show. This will determine how involved this will be

Some food for thought:

- ↳ We could have a Statistics class that accepts a Game object in its constructor and has various methods that return different stats about the game
- ↳ Do we explicitly save statistics to the persistence layer or do we just generate them on the fly whenever we need them?