Multiplayer mode for ReadiRacer

Background information

Why do we need this page

Product design for Readiracer

The multiplayer mode can be further broken down into 3 different modes:

- **Multiplayer - Class -** Where the student plays the game against other students from his class.
- **Multiplayer - School -** Where the student plays the game against other students from his school.
- **Multiplayer - World -** Where the student plays the game against other students from anywhere in the world (any school, any class, any country)

In future releases, we will add the ability to add more multiplayer modes where 3P educators and teachers will be able to set up a game mode by specifying the game mode name, game mode icon, students who can play the game, the duration for which the game mode is active and the default spelling list to be used for the game. Gameplay mechanics for any of the multiplayer modes will be the same except that the students who see these custom modes and play them would be limited to the selected students (by country, school, class, or any custom student selection).

The student would be able to join a lobby to select which group they want to play.

- Lobby design
    - Wait time
    - Number of player limitation per game
    - Allow a student to invite other classmates
- Categorize and grouping student's accuracy to allow them to play against each other in auto-match mode.
- Selecting a car (design and color) for individual personalization TBD
- Ranking/Result pages: Class, School, Country, World TBD

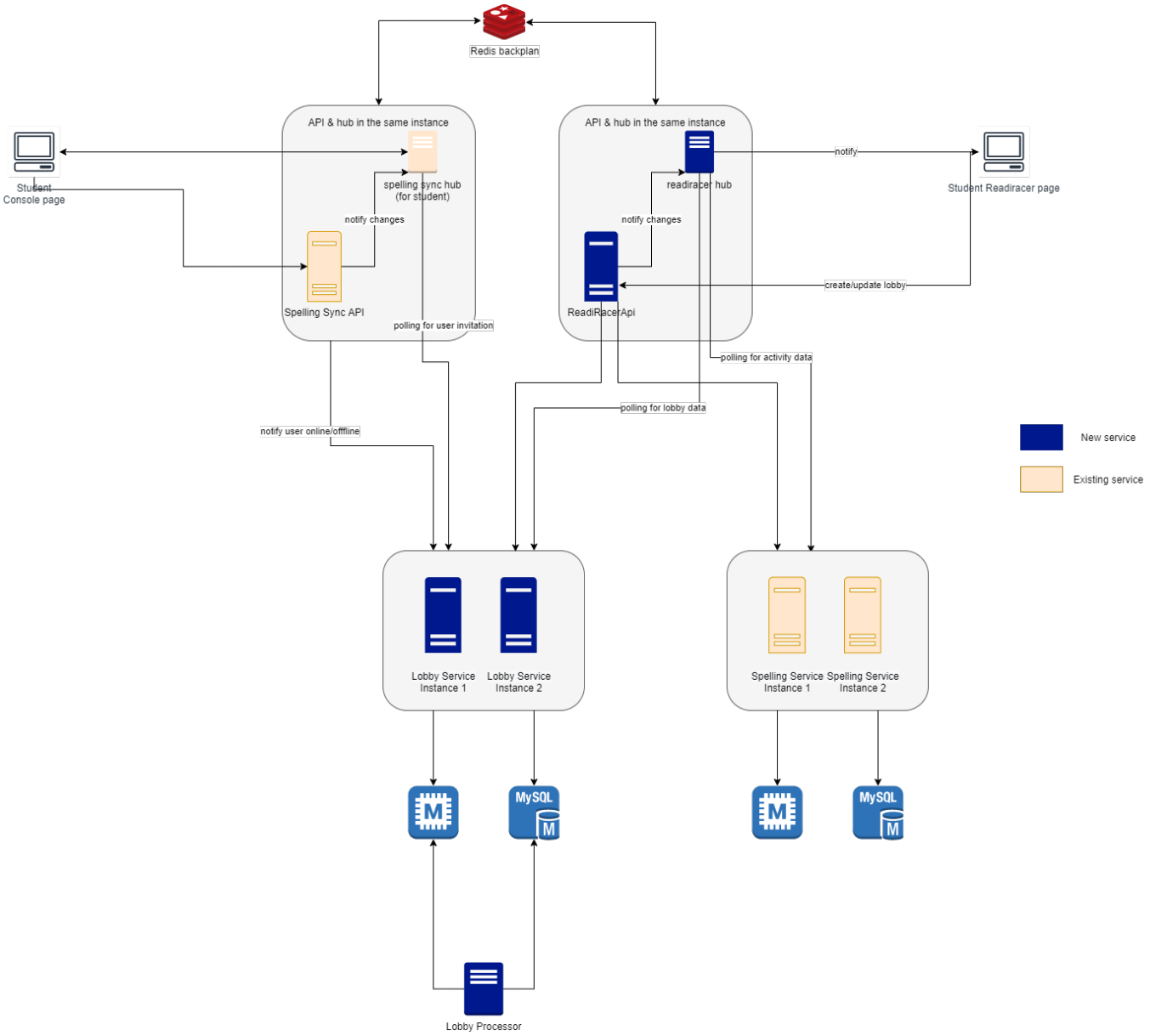
High-Level System Architecture

Issue

There is a couple of issues in the current implementation of live-test

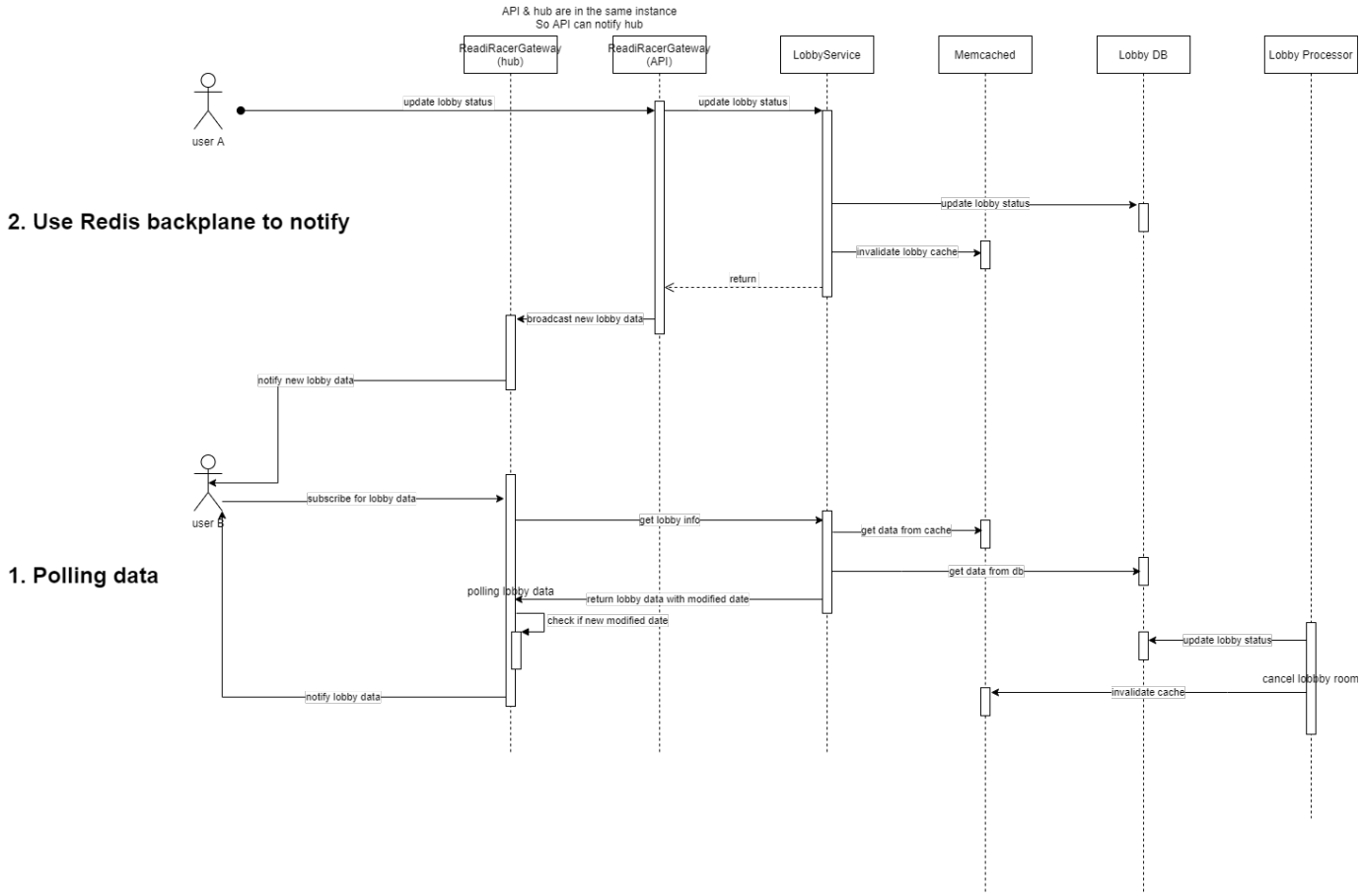
- The data is synchronized amongst hub instances. Triggered based on user interaction or on signalR connect/disconnect
    - Hard to debug and replicate the issue
    - Difficult to maintain/extend the current features
    - Data is replicated on each hub higher memory consumption especially for ReadiRacer since there is a lot of games is played compare to live-test
    - When a portion of data is missing in one of the hubs, there is no way to update this data
- Hubs are role-based, hard to introduce a new hub for readiracer
    - Online user hub for the teacher role (live-test manager)
    - spelling sync hub for the student role (notifications, live-test recipients)
    - Now we have ReadiRacer with a student as a host and auto-matching games

The solution



Typical flow of how a user retrieves a lobby notification when another user changes lobby status, there are 2 ways to notify to user

- Polling for new data from each hub, there is a few seconds delay between each query
- Notify the client using Redis backplane, for a faster response time. If the notification does not reach the client, the data eventually query by polling jobs



ReadiRacer Gateway - API

Verb	Endpoint	Description
POST	lobby/	Create new lobby room
GET	lobby/{lobbyId}	
PUT	lobby/{lobbyId}	Update lobby status
POST	lobby/{lobbyId}/invitations	Send an invitation to another user
PUT	lobby/{lobbyId}/invitations/{invitationId}	accept/deny an invitation
POST	lobby/{lobbyId}/resources	user selects a car color

ReadiRacer Gateway - hub

When the user connects to a hub, the client should specify which data expected to retrieve

Method	Description

`` Subcribe(scopes: Scope[]) ``` Scope enum: UserNotification, Lobby, StudentList, LobbyResource, ActivityAttempts	Subcribe for notification of (user notification, lobby, student list, lobby resource, activity attempts) Server will start polling for data for these registered scope
`` UnSubcribe(scopes: Scope[]) ```	Remove these resources from polling and notification when move to another UI
`` JoinActivity(lobbyId: Guid) ```	Join activity, change user status to busy
`` LeaveActivity(lobbyId: Guid) ```	leave activity, update user status

Lobby Service - API

Endpoint	Description	Request body	Response

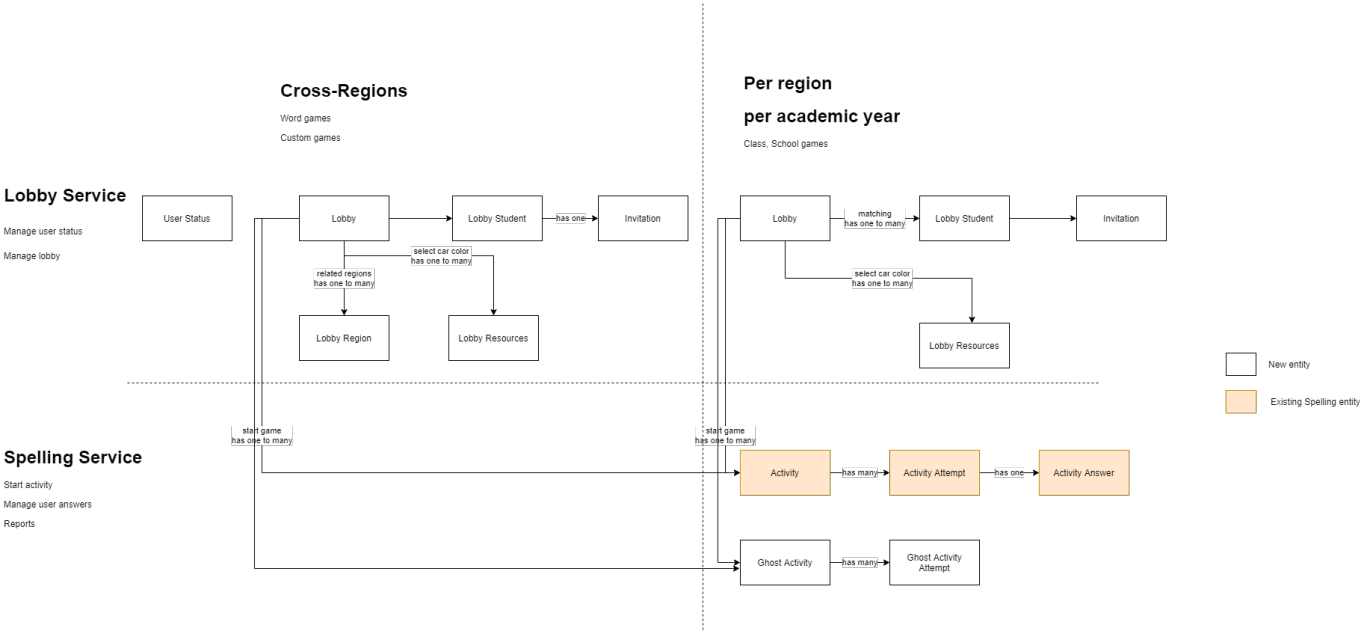
POST hubInstance/{instanceId}/users/	• Update new user status • Update busy status if user is participating in a “live-activity” • Update modified date of all records of this hub instance	`[{ userProfileId: int, classId: int, schoolId: int, status: UserStatus, onlineActivityType: int }]` `UserStatus`: Online, Offline `onlineActivityType` either live-test or readiracer	
GET users/online	Get number of users online that update recently		`[{ scope: string, nUsers: int }]` Scope enum: classmate, school, world

POST users/{executingUserProfileId} /lobby	Create new room	{ hostProfileId: int?, classId: int, schoolId: int, activityType: int, activityMode: int, configurations: Configuration } `Configuration` object { timer: int, timerStartTimeUTC: DateTime, scope: Scope }	{ lobbyId: Guid, hostProfileId: int?, classId: int, schoolId: int, activityType: int, activityMode: int, status: LobbyStatus, configurations: Configuration, resources: Resource[] } Resource { resourceType: int, resourceScope: string, userProfileId: int }

PUT users/{executingUserProfileId} /lobby/{lobbyId}	Update room status, trigger timer	<pre>{ status: LobbyStatus }</pre> status enum: created, matching, progressing, cancel, complete	
POST users/{executingUserProfileId} /lobby/{lobbyId}/invitations	Send an invitation to another user. Auto accept if the user is offline (ghost player)		
PUT users/{executingUserProfileId} /lobby/{lobbyId}/invitations/{invitationId}	accept/deny an invitation	<pre>{ invitationStatus: InvitationStatus }</pre> InvitationStatus: Pending, Accepted, Denied	
POST users/{executingUserProfileId} /lobby/{lobbyId}/resources	user select a car color. Car now only a type of resource	<pre>{ resourceType: ResourceType, resourceOrder: int, resourceStatus: ResourceStatus }</pre> ResourceType <pre>enum ResourceType { Car = 1, ... }</pre>	`Resource` object return `error code` if the resource is no longer available anymore
PUT users/{executingUserProfileId} /lobby/{lobbyId}/join	Join activity, update user status		

PUT users/{executingUserProfileId} /lobby/{lobbyId}/leave	Leave activity, update user status		

Entity Relationship



Number of Online users & user status

Approach 1: notify user connectivity periodically and data persist to DB

Data is sent from each hub and stored in DB, there is a job in the processor to clean up old data

How to capture the number of online users

hub

- periodically send NEW list online users to LobbyService

lobby service

- API: insert/update list of online users with new `ModifiedDateUtc`
- API: get the number of users online from Memcached or DB

lobby processor

- schedule a job to clean up records that haven't updated `ModifiedDateUtc` for a long time
- schedule a job to recalculate the number of online users and update Memcached

```
CREATE TABLE `online_users` (
  `HubInstanceId` GUID NOT NULL,
  `UserProfileId` INT(10) NOT NULL,
  ...
  `ModifiedDateUtc` DATETIME NOT NULL,
  PRIMARY KEY (`HubInstanceId`, `UserProfileId`)
) ENGINE=INNODB DEFAULT CHARSET=utf8mb4;
```

Advantages

- No race-condition among hubs since data is stored per `HubInstanceId`
- If a hub is restarted or going offline, the relevant records with the same `HubInstanceId` will be cleanup
- If the same user connects to multiple hubs, the API can distinct

Disadvantages

- heavy DB job to update/insert new online users records

How to capture user status

User status is determined based on

- data recorded from table `online_users`

Approach 2: notify the number of users online periodically

Each hub calculates the number of online users and only sends the calculated results. The data is stored in Memcached and uses **Cas operation** to solve race-condition problems.

How to capture the number of online users

Advantages

- No race-condition among hubs since data is stored per `HubInstanceId`
- Small DB footprint since we only store a few records per hub (1 record per class, per school)
- Can also store student id in JSON serialized for more accurate calculation
- If a hub is restarted or going offline, the relevant records with the same `HubInstanceId` will be cleanup
- Since the data relatively small, we can use **Cas operation on Memcached** to avoid race-condition and store data directly into `Memcache` https://github.com/enyim/EnyimMemcached/blob/485b5fe60a2f4383ff73b3cdb7ce4117a9aff4bc/Enyim.Caching/MemcachedClient.cs#L280
- or https://github.com/memcached/memcached/wiki/Commands#cas

Disadvantages

- If the same user connects to multiple hubs (multiple tabs), the API can not distinguish correctly

How to capture user status

User status is stored on Memcached only and each hub will manage different status and keep `modifiedDate` up-to-date. Status change uses Cas operation to prevent race-condition

Spelling Sync hub - connect to a student on console page and other activities

- Manages user status when it's `online`
- Update `modifiedDate` periodically on `online` status

ReadiRacer hub

- Change status to `busy` to indicate the user is playing readiracer game
- Change status to `leave-game` to indicate that the hub no longer maintain the user status, Spelling Sync hub can change status from `leave-game` to `online` again and continue to maintain the status
- Update `modifiedDate` periodically on `busy` status

How to manage user invitation

Polling for the new invitation, query data from `lobby` and `lobby_invitations`

How to manage user's resource, the student selects a car in the lobby

Advantages

- No race-condition among hubs since data locked on DB level
- This setup of primary keys makes sure that there is only one resource assigned to a single user.

```
CREATE TABLE `lobby_resources` (
  `LobbyId` GUID NOT NULL,
  `UserProfileId` INT(10) NOT NULL,
  `ResourceType` INT(10) NOT NULL,
  `ResourceScope` INT(10) NOT NULL
  ...
  PRIMARY KEY (`LobbyId`, `UserProfileId`, `ResourceType`)
) ENGINE=INNODB DEFAULT CHARSET=utf8mb4;
```