IMS project - Software Engineering Application Design Document[1]

Contents

Chapter 1

Use Cases

Actors:

- Participant – an experiment participant. Uses a smart watch.
- Operator – an experiment conductor. Uses the control console.
- Admin – the main experiment conductor.

1.1 Operator login

Actor: Operator

Description: An operator who is not currently logged in will enter their credentials to access the system.

Preconditions:

- The operator has a valid account in the system.
- The system is online and accessible.

Postconditions:

- The operator is successfully logged into the system.

Basic Flow:

1. The operator navigates to the login page.
2. The system prompts login credentials.
3. The operator enters their credentials.
4. The system verifies the credentials and logs the operator in.
5. The operator is redirected to the main dashboard.

Alternative Flows:

- **Invalid credentials:** The system notifies the operator of incorrect login credentials and prompts them to try again.
- **System offline:** The system displays an error message indicating that login is unavailable.

1.2 Admin can add or remove an operator from the system

Actor: Admin

Description: The admin can manage operator accounts by adding or removing them from the system.

Preconditions:

- The admin is logged in.
- The system is online.

Postconditions:

- The operator account is successfully added or removed.

Basic Flow (Adding an Operator):

1. The admin navigates to the operator management page.
2. The admin selects "Add Operator."
3. The system prompts for operator details.
4. The admin enters the details and submits.
5. The system creates the operator account and confirms the action.

Basic Flow (Removing an Operator):

1. The admin navigates to the operator management page.
2. The admin selects an operator and chooses "Remove."
3. The system removes the operator and confirms the action.

Alternative Flows:

- **Operator already exists (while adding):** The system notifies the admin that the operator account already exists.
- **Operator not found (while removing):** The system notifies the admin that the operator account does not exist.

1.3 Operators can add or remove a participant from the system

Actor: Operator

Description: An operator can manage participants by adding or removing them from the system.

Preconditions:

- The operator is logged in.
- The system is online.

Postconditions:

- The participant is successfully added or removed.

Basic Flow (Adding a Participant):

1. The operator navigates to the participant management page.
2. The operator selects "Add Participant."
3. The system prompts for participant details.
4. The operator enters the details and submits.
5. The system registers the participant and confirms the action.

Basic Flow (Removing a Participant):

1. The operator navigates to the participant management page.
2. The operator selects a participant and chooses "Remove."
3. The system removes the participant and confirms the action.

Alternative Flows:

- **Participant already exists (while adding):** The system notifies the operator that the participant already exists.
- **Participant not found (while removing):** The system notifies the operator that the participant account does not exist.

1.4 Operator can create a lobby

Actor: Operator

Description: An operator can create a lobby for experiments.

Preconditions:

- The operator is logged in.
- The system is online.

Postconditions:

- A new lobby is created and available for adding sessions.

Basic Flow:

1. The operator navigates to the lobby management page.
2. The operator selects "Create Lobby."
3. The system prompts for lobby details.
4. The operator enters the details and submits.
5. The system creates the lobby and confirms the action.

Alternative Flows:

- **One of the added participants became unavailable:** The system notifies the operator that the participant is not available.

1.5 Operator can add or remove sessions of a lobby

Actor: Operator

Description: The operator can manage sessions within a lobby.

Preconditions:

- The operator is logged in.
- The system is online.
- The lobby exists.

Postconditions:

- The session is successfully added, removed, or updated.

Basic Flow (Adding a Session):

1. The operator selects a lobby.
2. The operator chooses "Add Session."
3. The system prompts for session details.
4. The operator enters details and submits.
5. The system creates the session and confirms the action.

Basic Flow (Removing a Session):

1. The operator selects a lobby.
2. The operator chooses a session and selects "Remove."
3. The system asks for confirmation.
4. The operator confirms.
5. The system removes the session and confirms the action.

Alternative Flows:

- **Invalid input (while adding):** The system notifies the operator that the participant is not available.
- **The session doesn't exist in the backend (while removing):** The system notifies the operator that the session does not exist.

1.6 Operator can view information from the database

Actor: Operator

Description: The operator can access and view stored information from the database.

Preconditions:

- The operator is logged in.
- The system is online.

Postconditions:

- The requested information is displayed.

Basic Flow:

1. The operator navigates to the requested category to view.
2. The system retrieves and displays the requested data.

Alternative Flows:

- **Database connection issues:** The system notifies the operator that there has been an error while fetching the data.

1.7 Participants can participate in an experiment

Actor: Participant

Description: Participants take part in an experiment that consists of multiple stages.

Preconditions:

- The participant is registered.
- The participant is assigned to an experiment.

Postconditions:

- The participant completes the experiment.

Basic Flow:

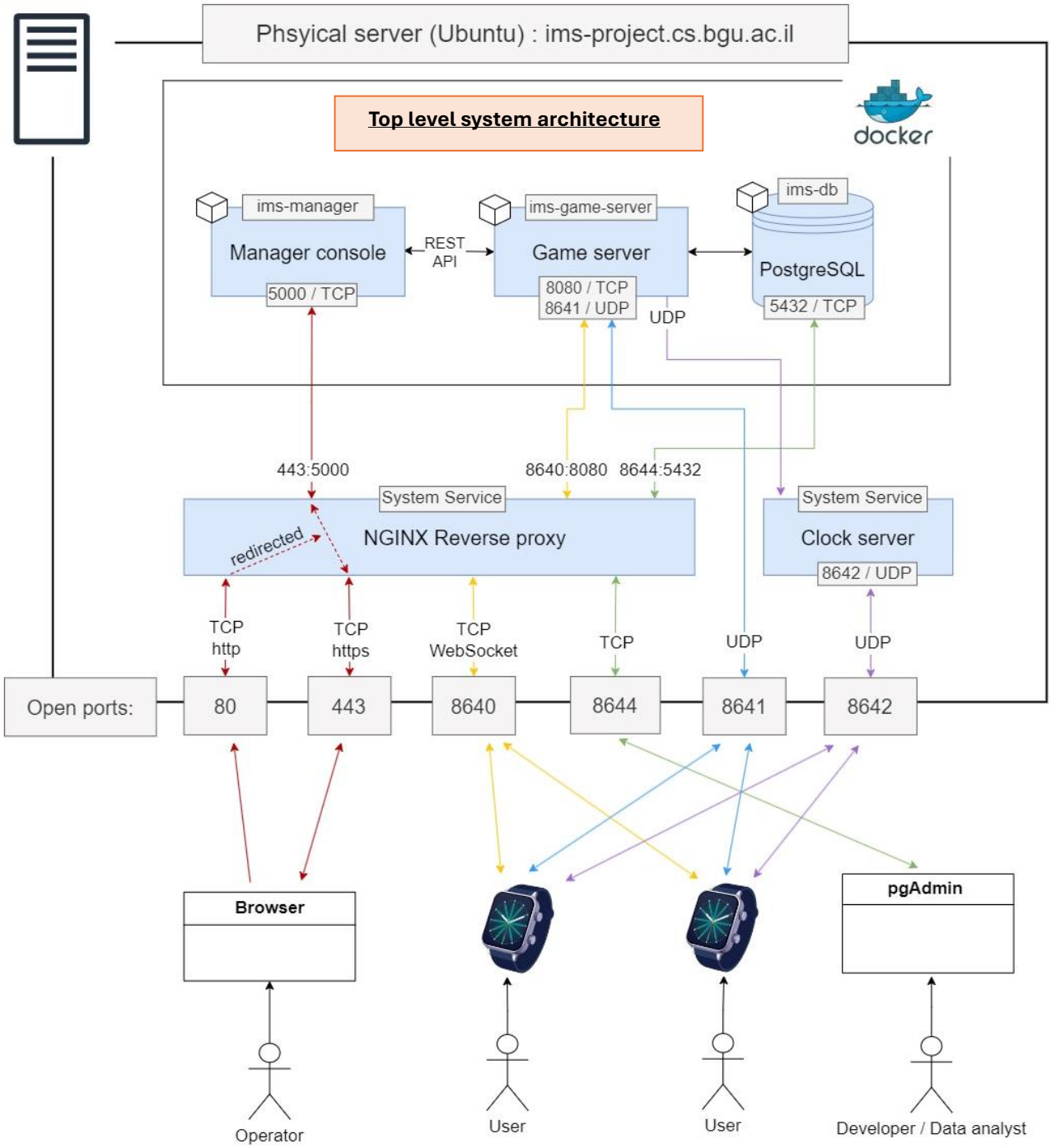
1. The participant logs in the watch with their assigned id.
2. The system redirects to the assigned experiment lobby.
3. The system provides instructions.
4. The participant performs the required tasks.
5. The system records data.
6. The session concludes, and the system sends the data to the server.

Alternative Flows:

- **The session was stopped in the middle:** The system marks the session as incomplete, and the participant is notified the session has ended.

Chapter 2

System Architecture



Game Server REST API Documentation

Internal Server URL

http://ims-game-server:8080

The REST API architecture

The manager console communicates with the game server via a REST API. The watch app also uses the '/data' endpoint to send the session data to the game server.

The following REST API documentation shows the architecture of the REST API, written in markdown.

Response Format

All responses from the API adhere to the following structure:

```
{
  "message": "string?",
  "success": "boolean",
  "payload": "[string]?"
}
```

Note: Not all fields are present in every response. fields marked with `?` can be missing

- `message` - provides a message to show to the client if there is any. Also can be used as an error message if the success field is `false`.
- `success` - indicates whether the operation was successful `READ: this field always exists`
- `payload` - if applicable, holds a list of data that is returned from the server. If the expected data is otherwise not a simple string, it will be a serialized json object that needs to be parsed.

Endpoints

1. GET `/auth`

Description

This endpoint is used for user authentication.
It validates the credentials provided in the `Authorization` header and returns a `Bearer` token that is required for subsequent requests.

Request

- **Headers**:
  - `Authorization : Basic <base64>`
    where `<base64>` - base64 encoded string of `username:password`

Return value

the success field will be true if the credentials are valid, and false otherwise.
If the credentials are valid, the `payload` field in the response json will hold a list with a single string, which is the token.

2. POST `/manager`

Description

Handles all game server requests

Request

- **Headers**:
  - `Authorization: Bearer <token>` - Required for all requests.
- **Body**:

```
{
  "type": "string",
  "playerId": "string?",
  "lobbyId": "string?",
  "gameType": "string?",
  "duration": "integer?",
  "sessionId": "string?",
  "sessionIds": ["string?"]
}
```

Note: Not all fields are present in every request. fields marked with `?` can be missing

Fields:

`type` (required):

Specifies the type of operation.
Possible values:

- `get_online_player_ids`
- `get_lobbies`
- `get_lobby`
- `create_lobby`
- `remove_lobby`
- `join_lobby`
- `leave_lobby`
- `start_experiment`
- `end_experiment`
- `create_session`
- `remove_session`
- `get_sessions`
- `change_sessions_order`

`playerId` (situational):

ID of a player involved in the operation.

`lobbyId` (situational):

ID of the target lobby.

`gameType` (situational):

The game type (e.g. `water_ripples`).

`duration` (situational):

The duration of the game in seconds.

`sessionId` (situational):

The ID of the session.

`sessionIds` (situational):

A list of session IDs.

Request types and their required fields:

`get_online_player_ids`

Required fields:

- None

Description: Retrieves the list with ids of online players. No additional fields are required for this request type.

Return value:

```
{
  "success": true,
  "payload": [
    "player1",
    "player2",
    "player3"
  ]
}
```

`get_lobbies`

Required fields:

- None

Description:
Retrieves the list of available lobbies. No additional fields are required for this request type.

Return value:

```
{
  "success": true,
  "payload": [
    {
      "lobbyId": "lobby1",
      "state": "waiting",
      "gameType": "gameTypeA",
      "duration": 60,
      "syncWindowLength": 5000,
      "syncThreshold": 50,
      "players": ["player1", "player2"],
      "readyStatus": ["false", "true"]
    },
    {
      "lobbyId": "lobby2",
      "state": "playing",
      "gameType": "gameTypeB",
      "duration": 60,
      "syncWindowLength": 4000,
      "syncThreshold": 100,
      "players": ["player3"],
      "readyStatus": ["true"]
    }
  ]
}
```

`get_lobby`

Required fields:

- `lobbyId`

Description:
This request type is used to get the details of a specific lobby. The `lobbyId` must be provided to identify the lobby.

Return value:

```
{
  "success": true,
  "payload": [
    {
      "lobbyId": "lobby1",
      "state": "waiting",
      "gameType": "gameTypeA",
      "duration": 60,
      "syncWindowLength": 5000,
      "syncThreshold": 50,
      "players": ["player1", "player2"],
      "readyStatus": ["false", "true"]
    }
  ]
}
```

`create_lobby`

Required fields:

- `gameType`

Description:
This request type is used to create a new lobby. The `gameType` field must be provided to specify the type of game to be played in the new lobby.

Return value:

```
{
  "success": true,
  "payload": ["newLobbyId"]
}
```

`remove_lobby`

Required fields:

- `lobbyId`

Description: This request type is used to remove a specific lobby. The `lobbyId` must be provided to identify the lobby to be removed.

Return value:

```
{
  "success": true
}
```

`join_lobby`

Required fields:

- `lobbyId`
- `playerId`

Description:
This request type is used to join an existing lobby. Both the `lobbyId` (to identify the target lobby) and the `playerId` (to identify the player joining the lobby) are required.

Return value:

```
{
  "success": true
}
```

`leave_lobby`

Required fields:

- `lobbyId`
- `playerId`

Description:
This request type is used when a player leaves a lobby. Both `lobbyId` (to identify the lobby) and `playerId` (to identify the player leaving the lobby) are required.

Return value:

```
{
  "success": true
}
```

`start_experiment`

Required fields:

- `lobbyId`

Description:
This request type starts the experiment in a specific lobby. The `lobbyId` must be provided to identify the lobby where the game should start.

Return value:

```
{
  "success": true
}
```

`end_experiment`

Required fields:

- `lobbyId`

Description:
This request type ends the experiment in a specific lobby. The `lobbyId` must be provided to identify the lobby where the game should end.

Return value:

```
{
  "success": true
}
```

`create_session`

Required fields:

- `lobbyId`
- `gameType`
- `duration`
- `syncWindowLength`
- `syncThreshold`

Description: This request type is used to create a new session.

Return value:

```
{
  "success": true,
  "payload": ["newSessionId"]
}
```

`remove_session`

Required fields:

- `lobbyId`
- `sessionId`

Description: This request type is used to remove a specific session from a lobby.

Return value:

```
{
  "success": true
}
```

`get_sessions`

Required fields:

- `lobbyId`

Description: This request type is used to get the list of sessions in a specific lobby.

Return value:

```
{
  "success": true,
  "payload": [
    {
      "sessionId": "session1",
      "gameType": "gameTypeA",
      "duration": 60,
      "syncWindowLength": 5000,
      "syncThreshold": 50
    },
    {
      "sessionId": "session2",
      "gameType": "gameTypeB",
      "duration": 120,
      "syncWindowLength": 4000,
      "syncThreshold": 100
    }
  ]
}
```

`change_sessions_order`

Required fields:

- `lobbyId`
- `sessionIds`

Description: This request type is used to change the order of sessions in a lobby.

Return value:

```
{
  "success": true
}
```

---

3. POST `/data`

Description

This endpoint is used to send session events after a session is finished

Request

- **Body**:

```
{
  "sessionId": "id"
  "events": ["serializedEventsList"]
}
```

Response

```
{
  "success": true,
}
```

4. POST `/operators/{action}`

Description

This endpoint manages user operations such as adding or removing operators. The specific action is determined by the `action` path variable, which can be either `add` or `remove`.

Request

- **Path Variables**:
  - `action` (required): The action to perform.
    - **Possible values**:
      - `add` : Add a new operator.
      - `remove` : Remove an existing operator.
- **Body**:

  ```
  {
    "userId": "string",
    "password": "string"
  }
  ```

  Required Fields:

  - `userId` : The unique identifier of the user. Must be in lowercase.
  - `password` : The password of the user. (Required only for the `add` action.)

  Password Requirements (for `add` action):

  - At least 8 characters.
  - At least one uppercase letter.
  - At least one lowercase letter.
  - At least one digit.
  - May contain special characters (`!@#$%^&*()-=_+[]{};:<>?/\~|`).

Response

- **Success**:

  ```
  {
    "message": "User added successfully",
    "success": true
  }
  ```

  or

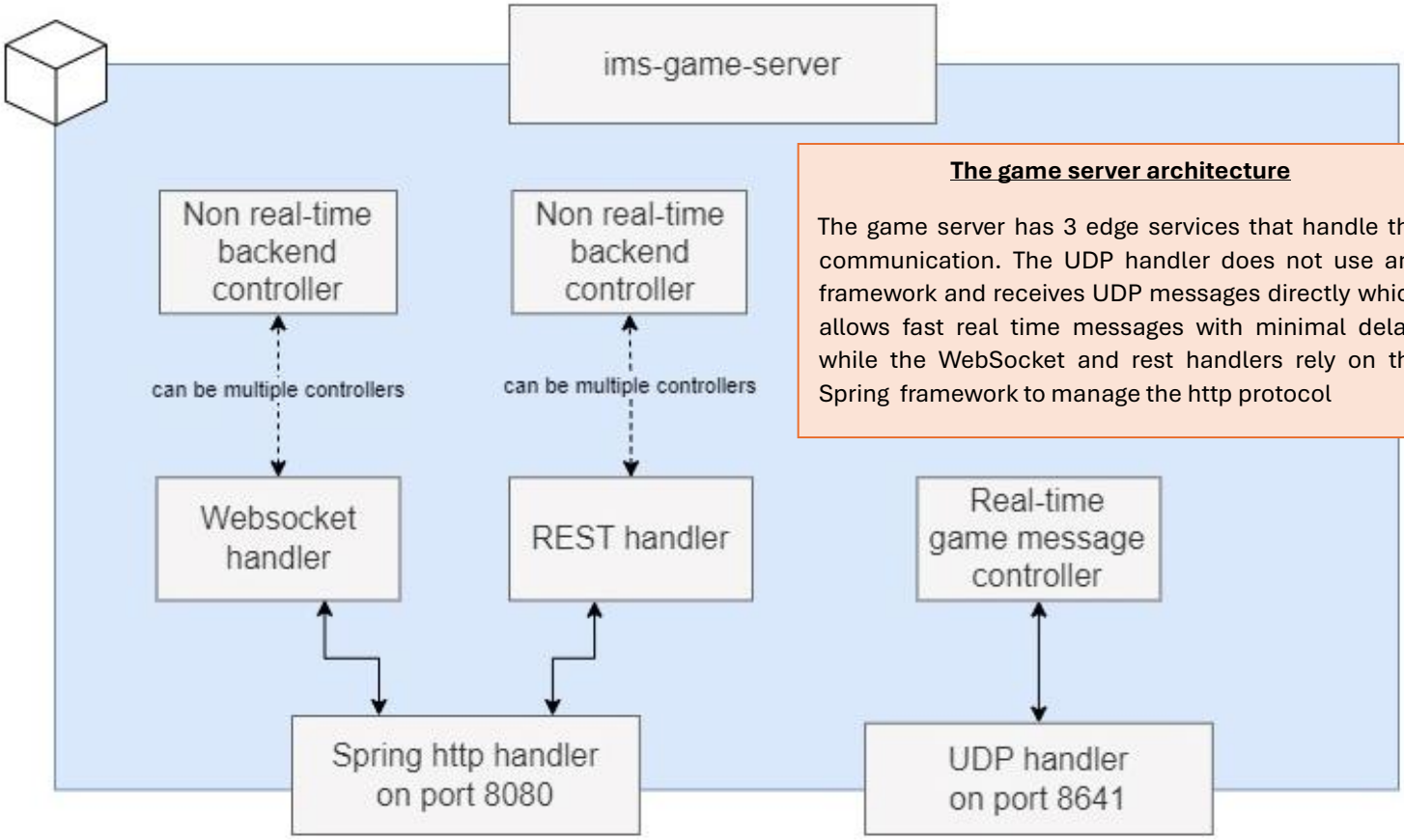
  ```
  {
    "message": "User removed successfully",
    "success": true
  }
  ```

- **Failure**:

```
{
  "message": "string",
  "success": false
}
```

Examples of failure messages:

- `"Invalid action"`
- `"User already exists"`
- `"Password does not meet the requirements"`
- `"User not found"`



The game server architecture

The game server has 3 edge services that handle the communication. The UDP handler does not use any framework and receives UDP messages directly which allows fast real time messages with minimal delay, while the WebSocket and rest handlers rely on the Spring framework to manage the http protocol

Chapter 3

Data Model

3.1 Description of Data Objects

Our system has several entities, some are volatile, and some are persistent.

The following business entities are volatile state-oriented:

1) Lobby:

holds state of the next session (that will be started in the future), information about participants that are in the lobby, whether they are currently in a session or not and whether any sessions are configured for the lobby.

It has its own id, which is unique while the system is running (resets every time the system is restarted)

It holds two participant ids which signify that they are both in the lobby, and they cannot be in a different lobby. It also holds the ready status of each participant.

The information it stores about the pending session is the game type, the game duration, sync window length and sync tolerance.

The lobby object is also used to synchronize the state of the watch app with the server

2) Game:

The Game entity is used to run a session currently being played.

It holds both participants' handler (an object that can be used to send messages to the participants), handles the game messages from the watches and handles the game logic appropriately.

The game entity is extended by each respective game mode (water ripples, wine glasses etc) to override default functionality where needed

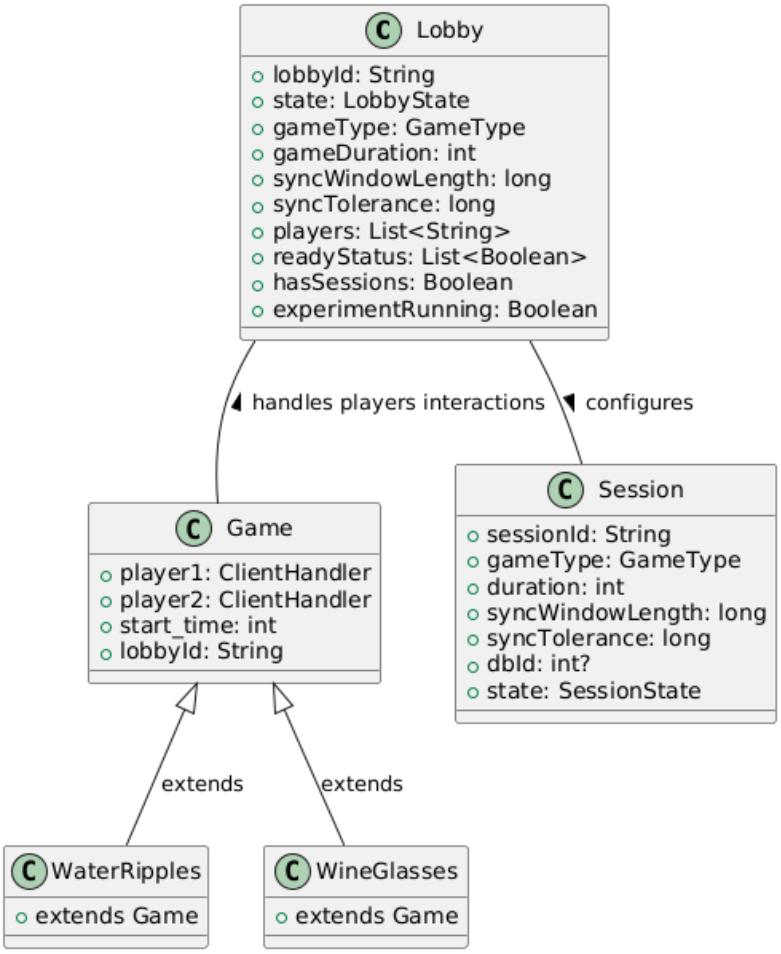
3) Session:

The session data class is used to store the data for every session in an experiment. This object is used by the lobby to configure it.

Session maps almost identically to the database.

3.2 Data Objects Relationships

This class diagram captures the main data objects in our project as described above in a compact way:

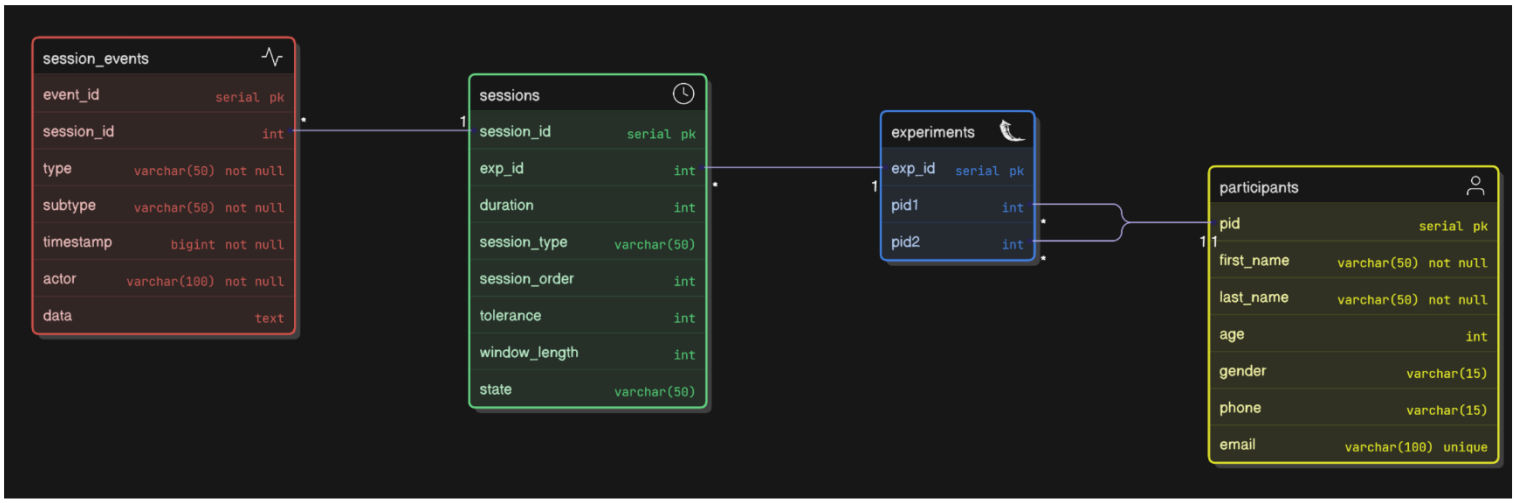


3.3 Databases

- The tables in our database are:
    - Participants - represents the experiment participants signed up.

- Experiments – represents the assignment of two participants to be together in an experiment. Each experiment has two participants.
- Sessions – represents the sessions that are part of an experiment. Each experiment has some number of sessions.
- Session Events - represents the saved data for a session from a participant's watch.

- Transactions in the code:
  - Creation of an experiment with sessions information – creating an experiment affects the experiments table. If the participant ids provided are valid and exist in participants table, the operation should be successful. Then, adding each session to the sessions table occurs.
  - Sending session events from a watch – sending the session events happens as a batch transaction, as the number of session events sent at once is high. The batch transaction consists of insert statements for every event session into the session events table.

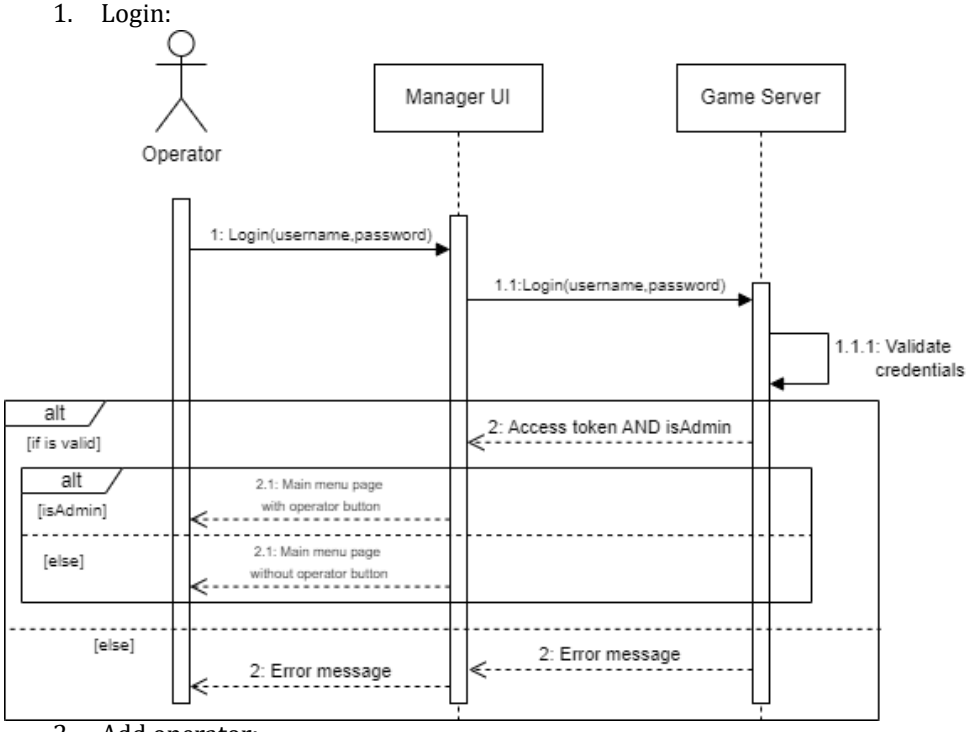


Chapter 4

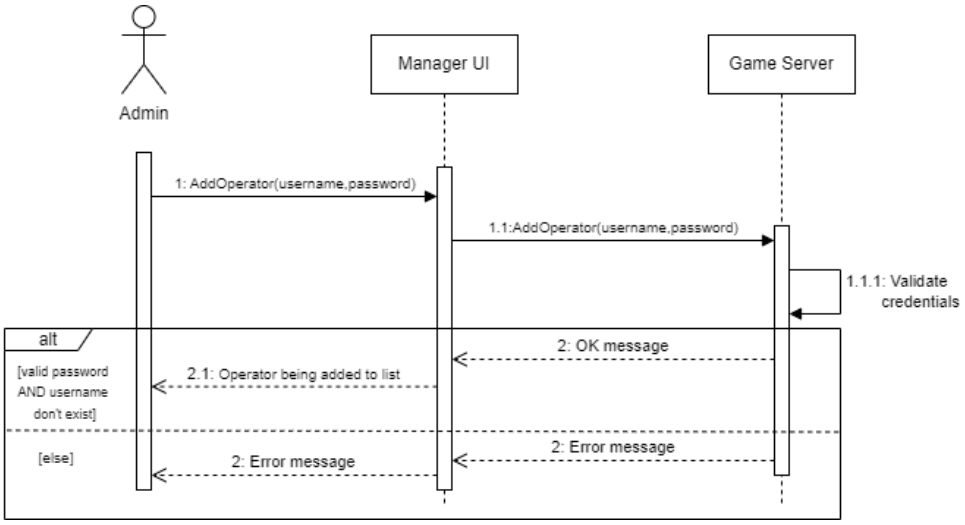
Behavioral Analysis

4.1 Sequence Diagrams

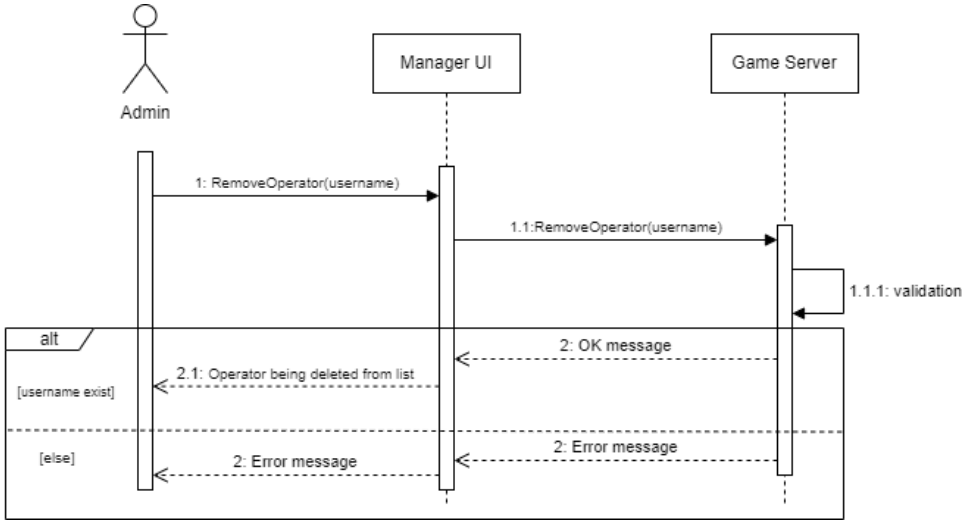
1. Login:



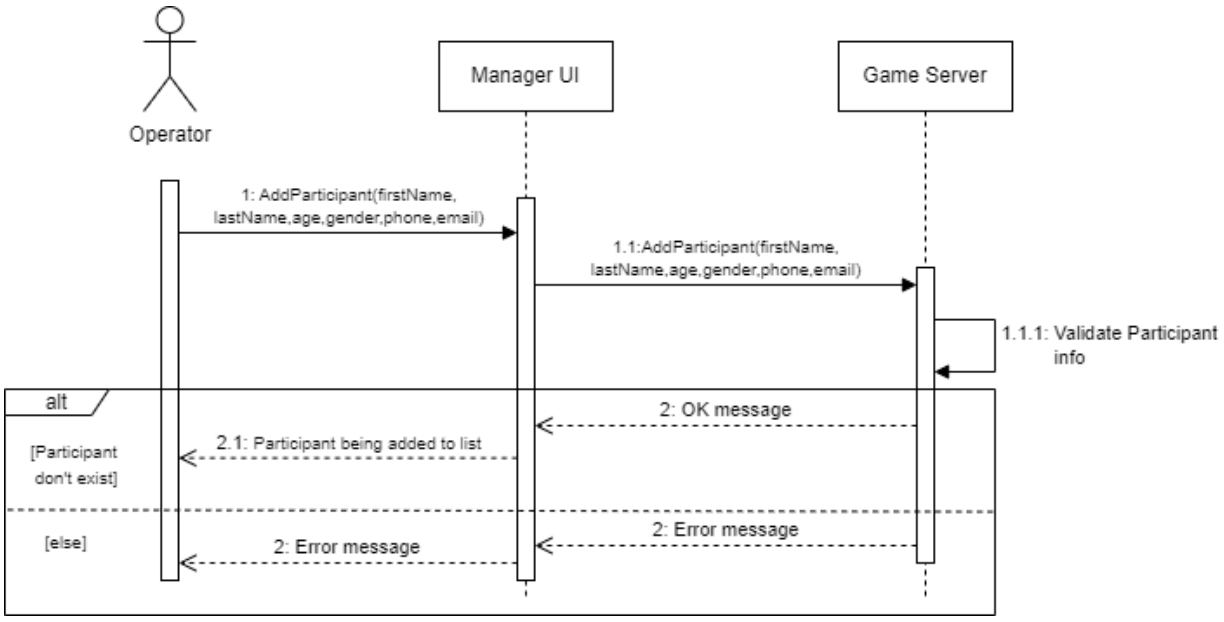
2. Add operator:



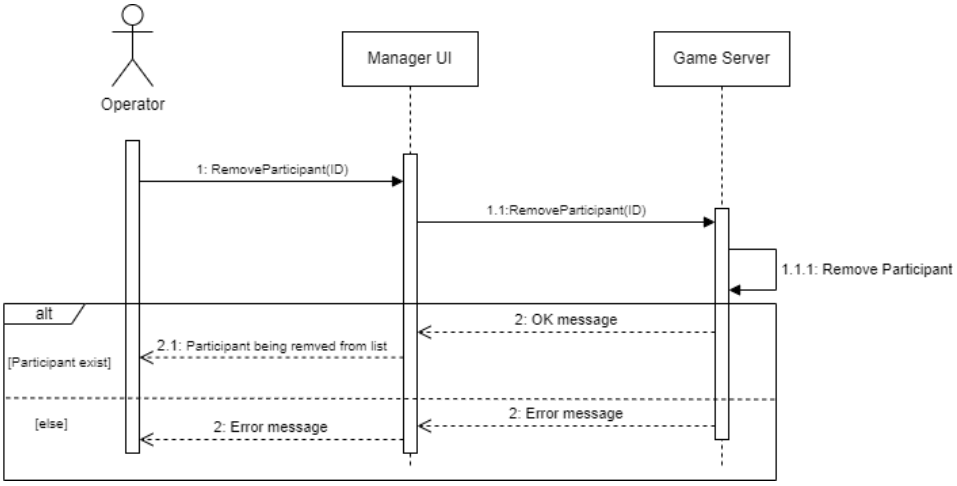
3. Remove operator:



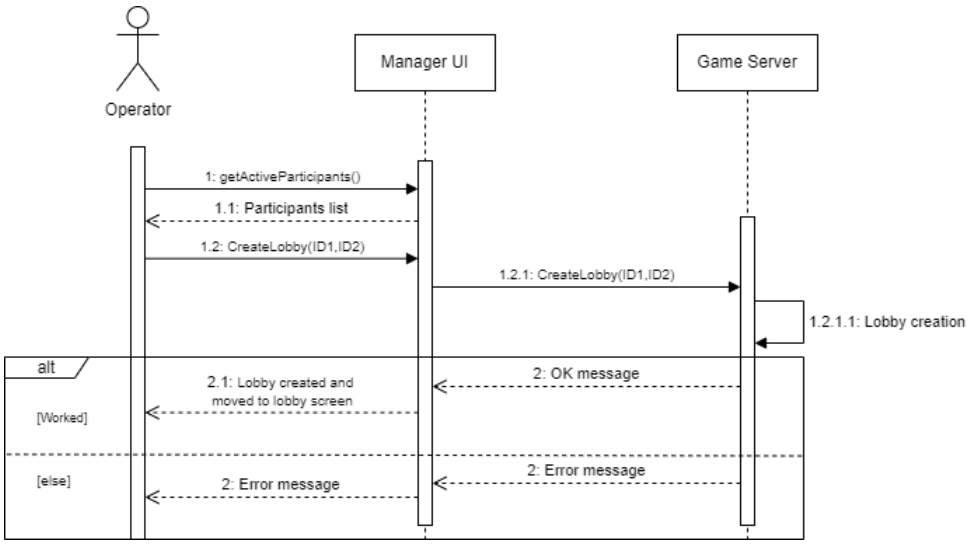
4. Add participant:



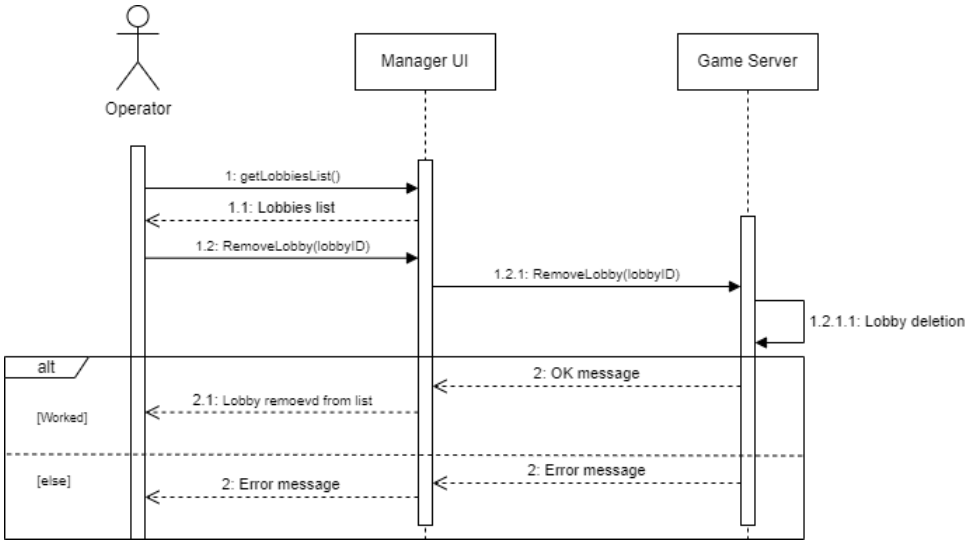
5. Remove participant



6. Create lobby:

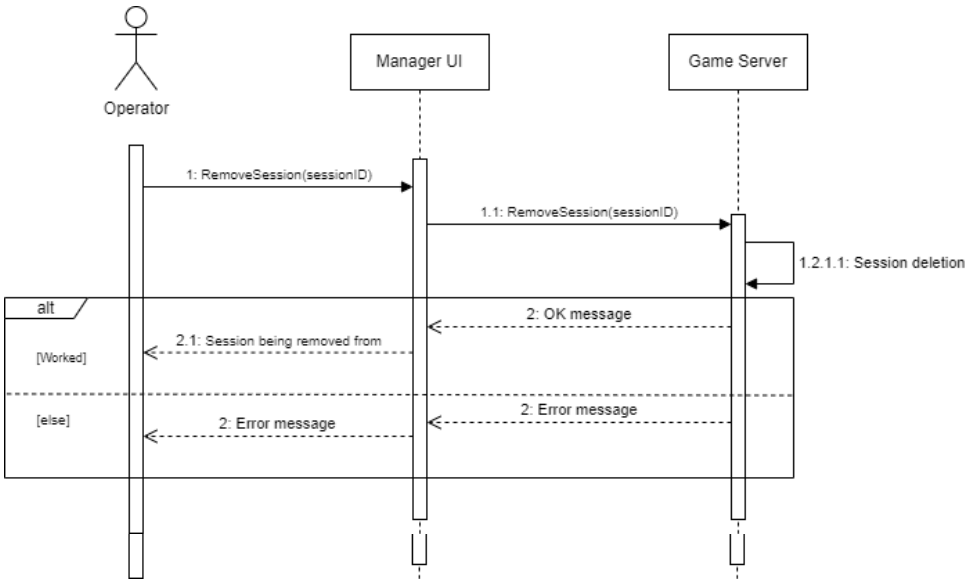


7. Remove lobby:

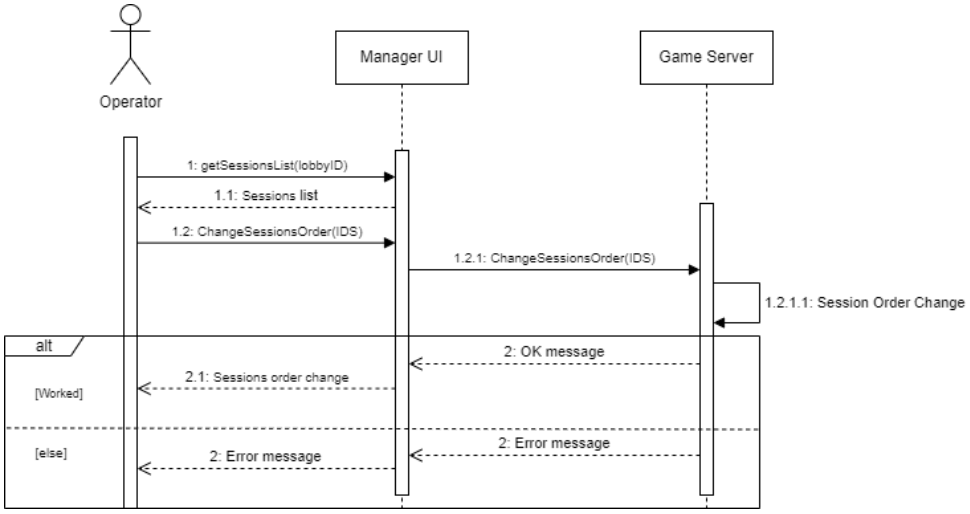


8. Add session:

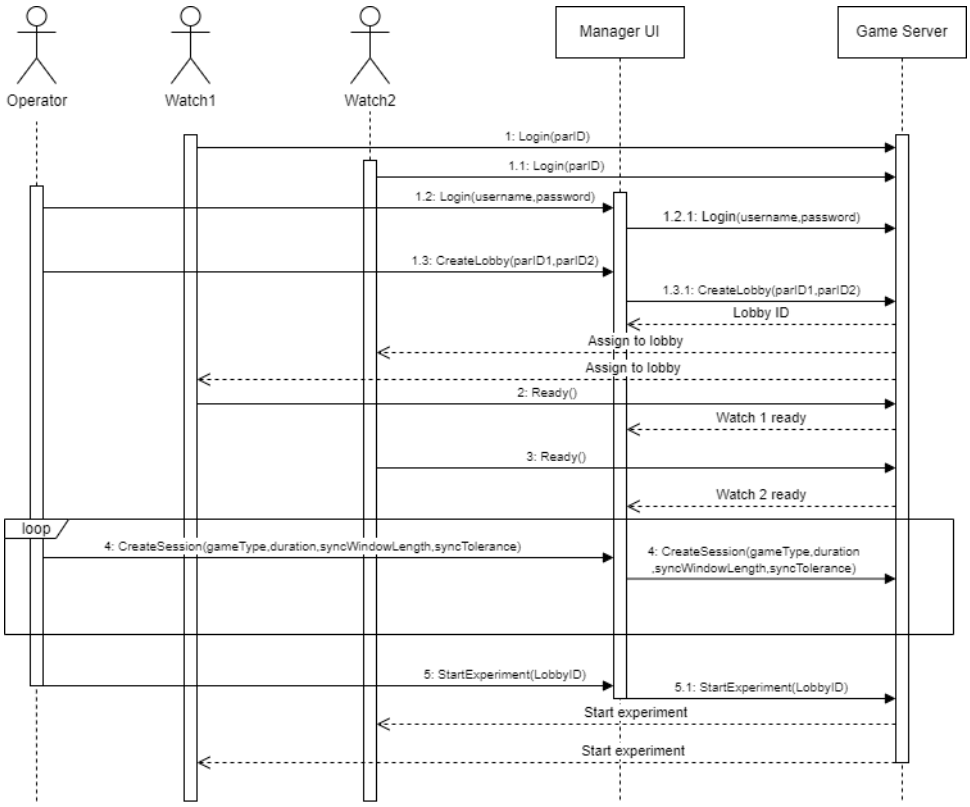
9. Remove session:



10. Change session order:



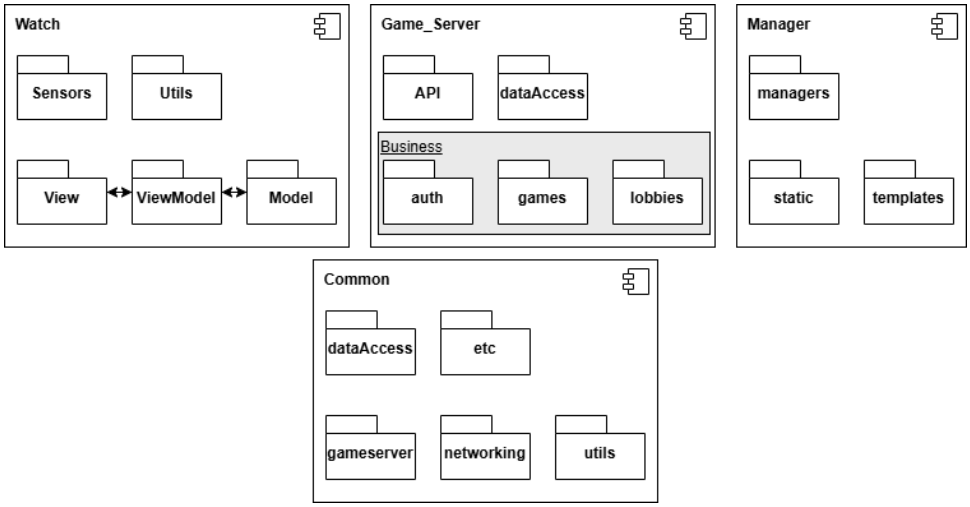
11. Start experiment


Operator
Watch1
Watch2
Manager UI
Game Server
1: Login(parID)
1.1: Login(parID)
1.2: Login(username,password)
1.2.1: Login(username,password)
1.3: CreateLobby(parID1,parID2)
1.3.1: CreateLobby(parID1,parID2)
Lobby ID
Assign to lobby
Assign to lobby
2: Ready()
Watch 1 ready
3: Ready()
Watch 2 ready
loop
4: CreateSession(gameType,duration,syncWindowLength,syncTolerance)
4: CreateSession(gameType,duration
,syncWindowLength,syncTolerance)
5: StartExperiment(LobbyID)
5.1: StartExperiment(LobbyID)
Start experiment
Start experiment

Chapter 5

Object-Oriented Analysis

5.1 Packages



Our project is divided into 4 main parts: watch, game server, manager and common.

Watch

The watch side contains the code for the application that ran on the smartwatches.
Sensors - all the code for collecting and storing data from the physiological and spatial sensors.
Utils - Includes general data processing and storage departments.
Model – Part of the MVVM architecture. Contains the logic and data handling.
ViewModel – Connect the Model and the View.
View – the front side of the application.

Game Server

The server who manages all the communication between the components of the system.
API – As described before on the System Architecture section.
DataAccess – Responsible for all communication with the database.
Auth - Manages authentication and security for administrators.
Games - Manage experiments, sessions, and actions in real time.
Lobbies - Establishing a lobby for trial participants, configuring details for the experiment.

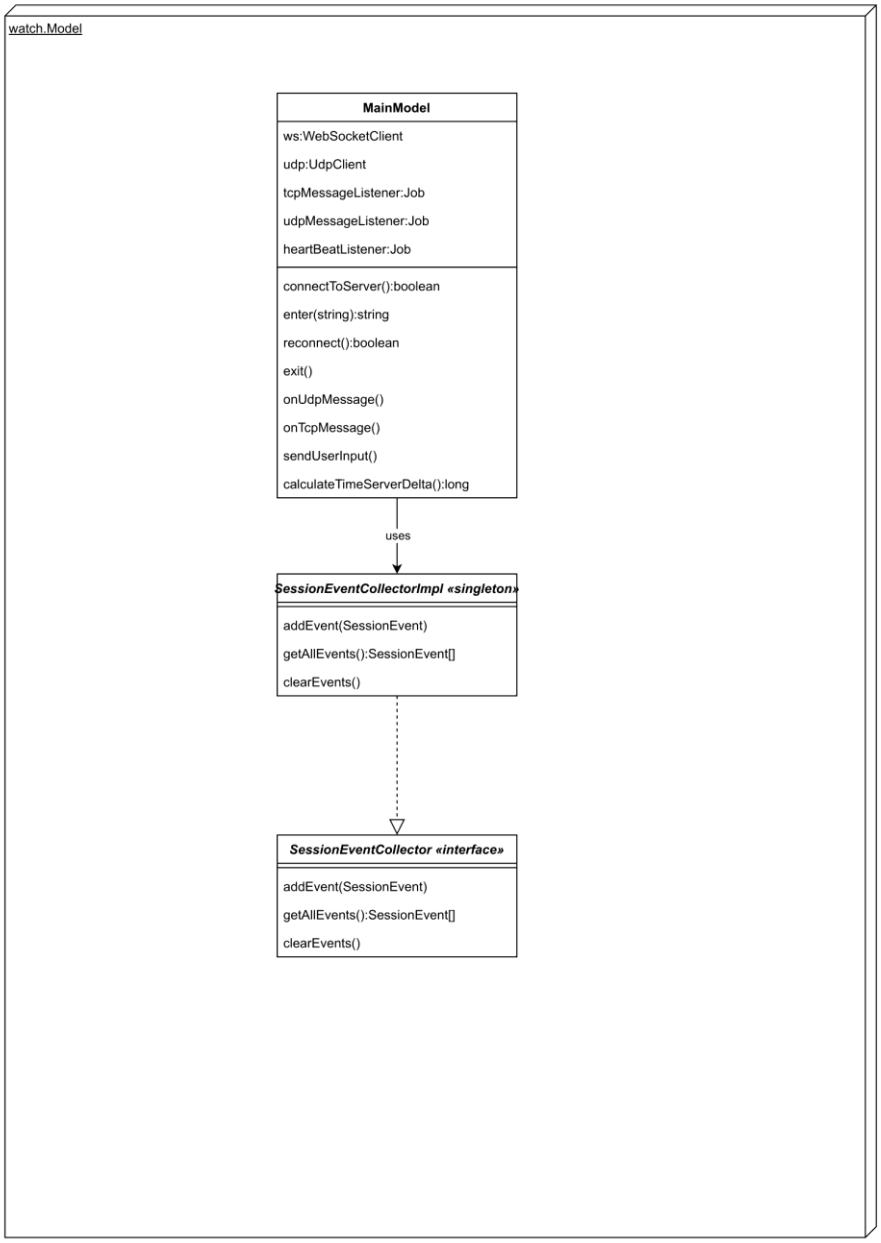
Manager

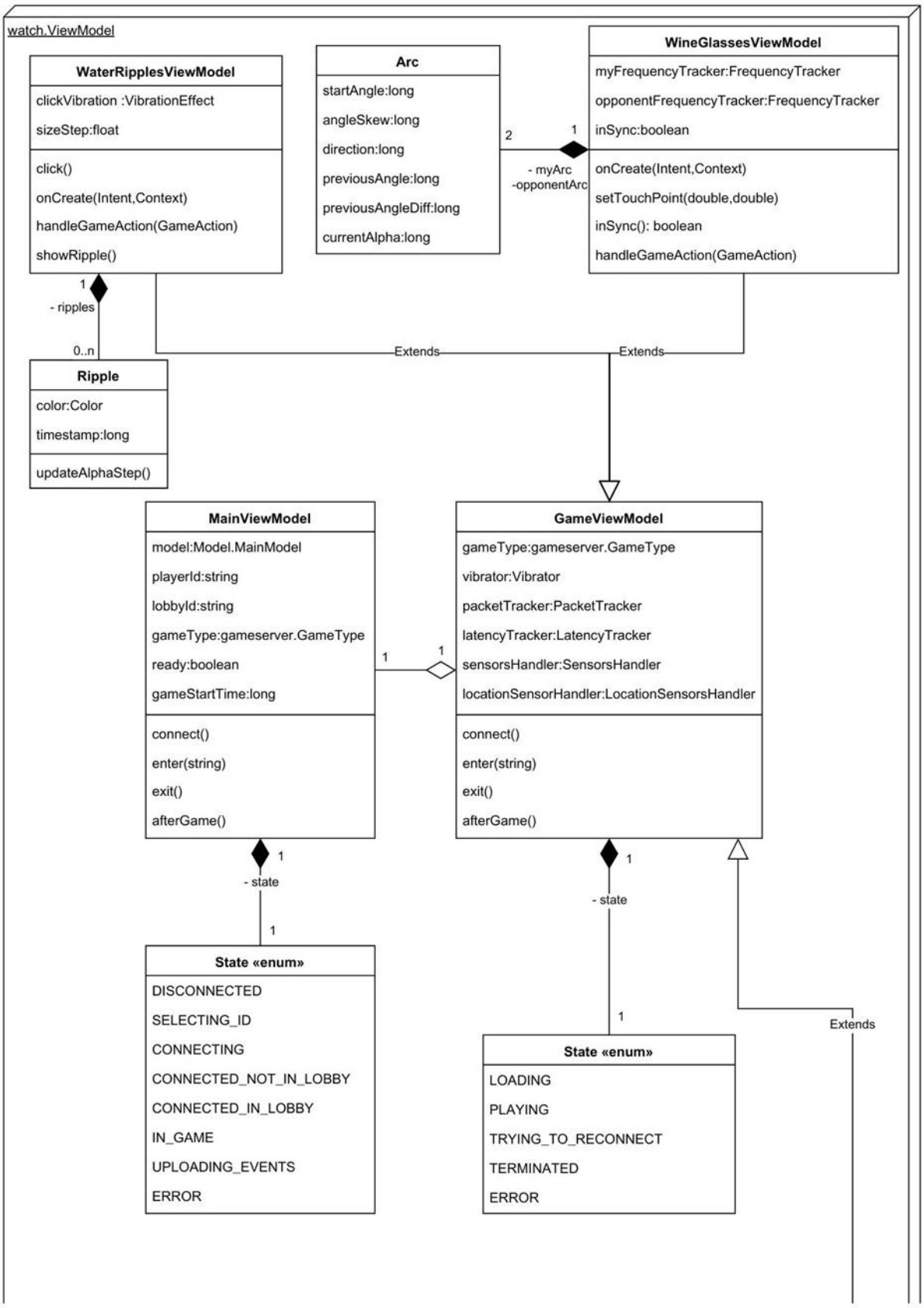
All the code for the management console ran as a website. From the console, administrators manage the experiment and establish communication with the system.

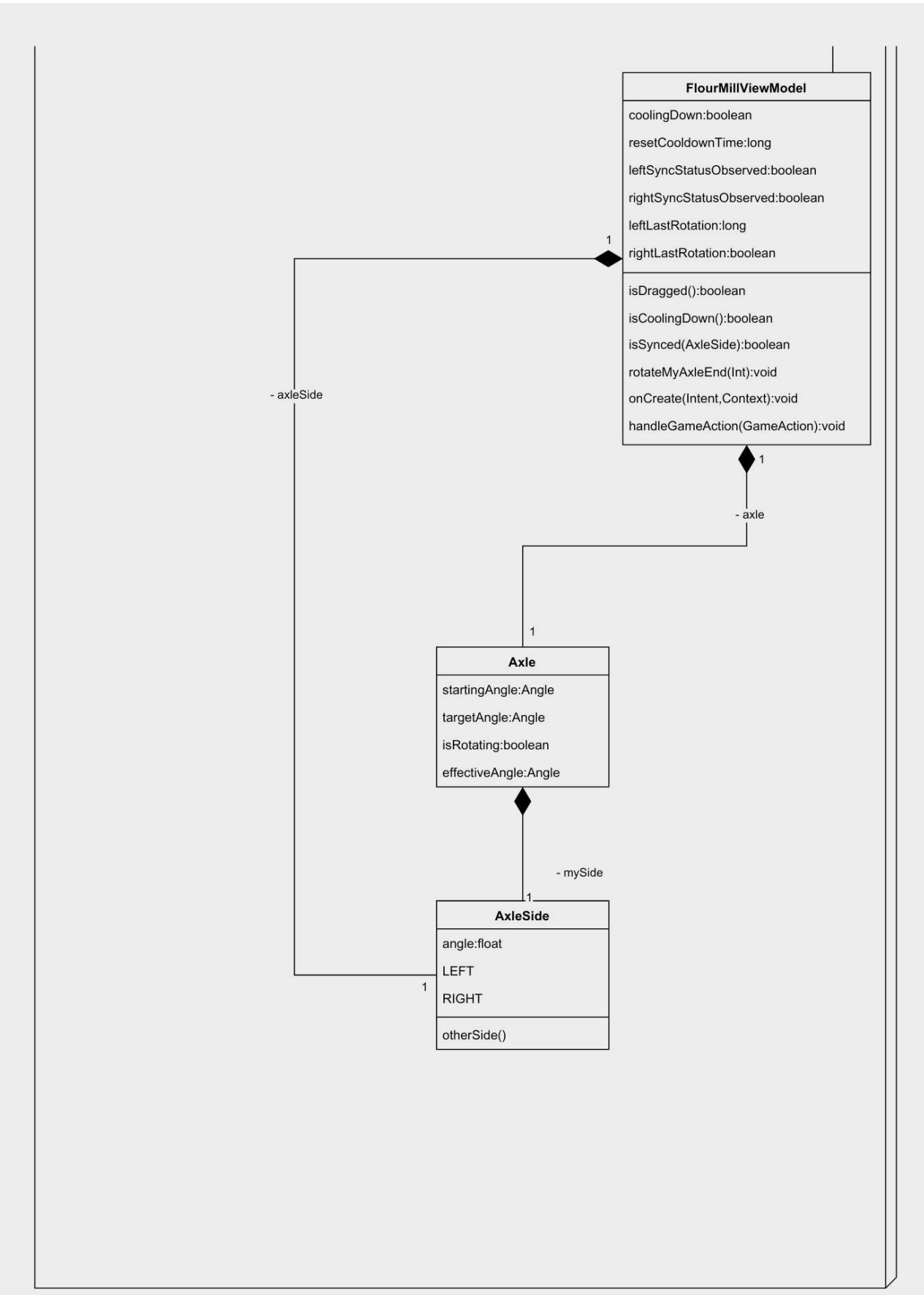
Common

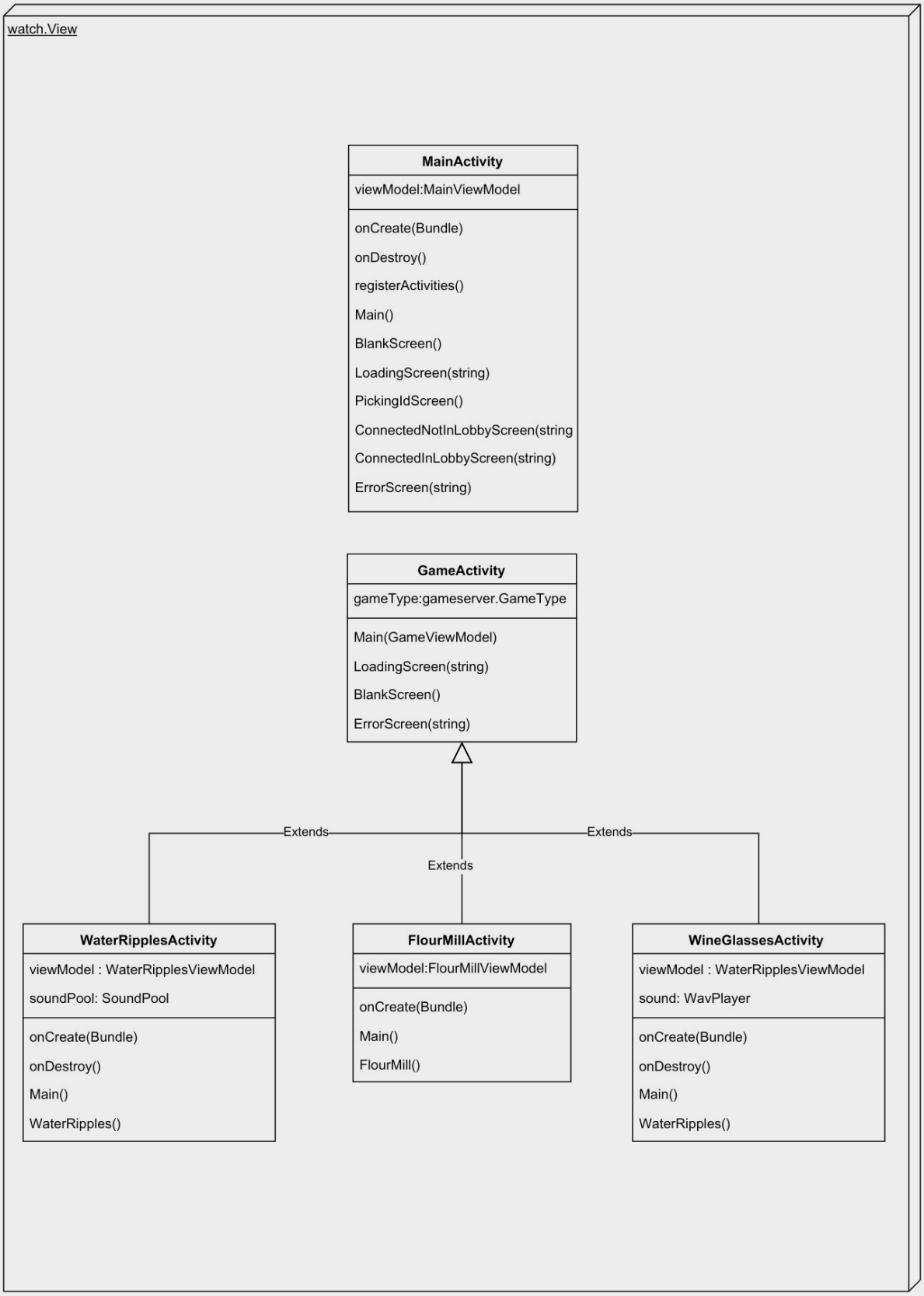
All classes that are common to all project components. The classes in the package are intended for alignment between all developers, and for defined and clear communication.

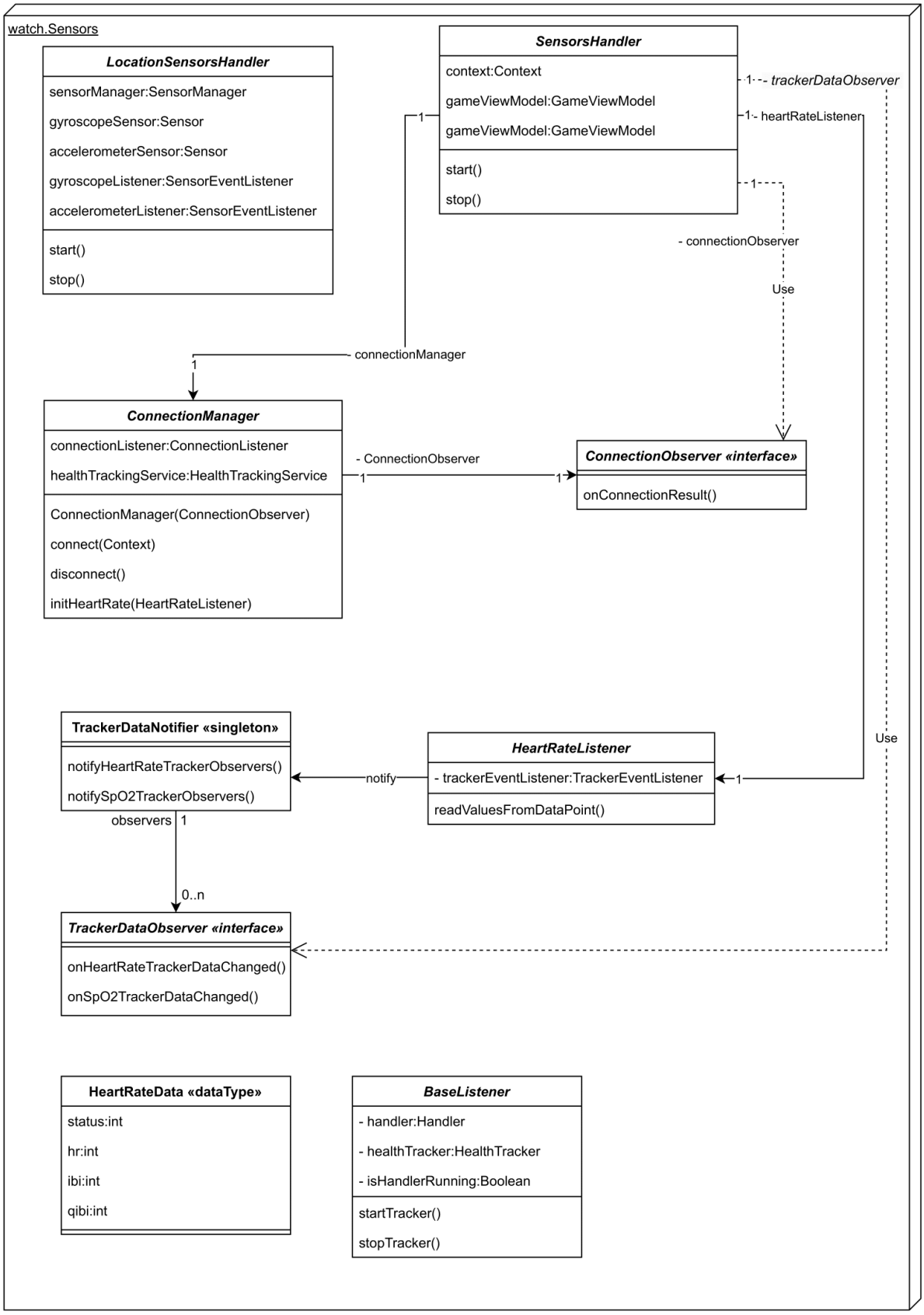
5.2 Class Diagram

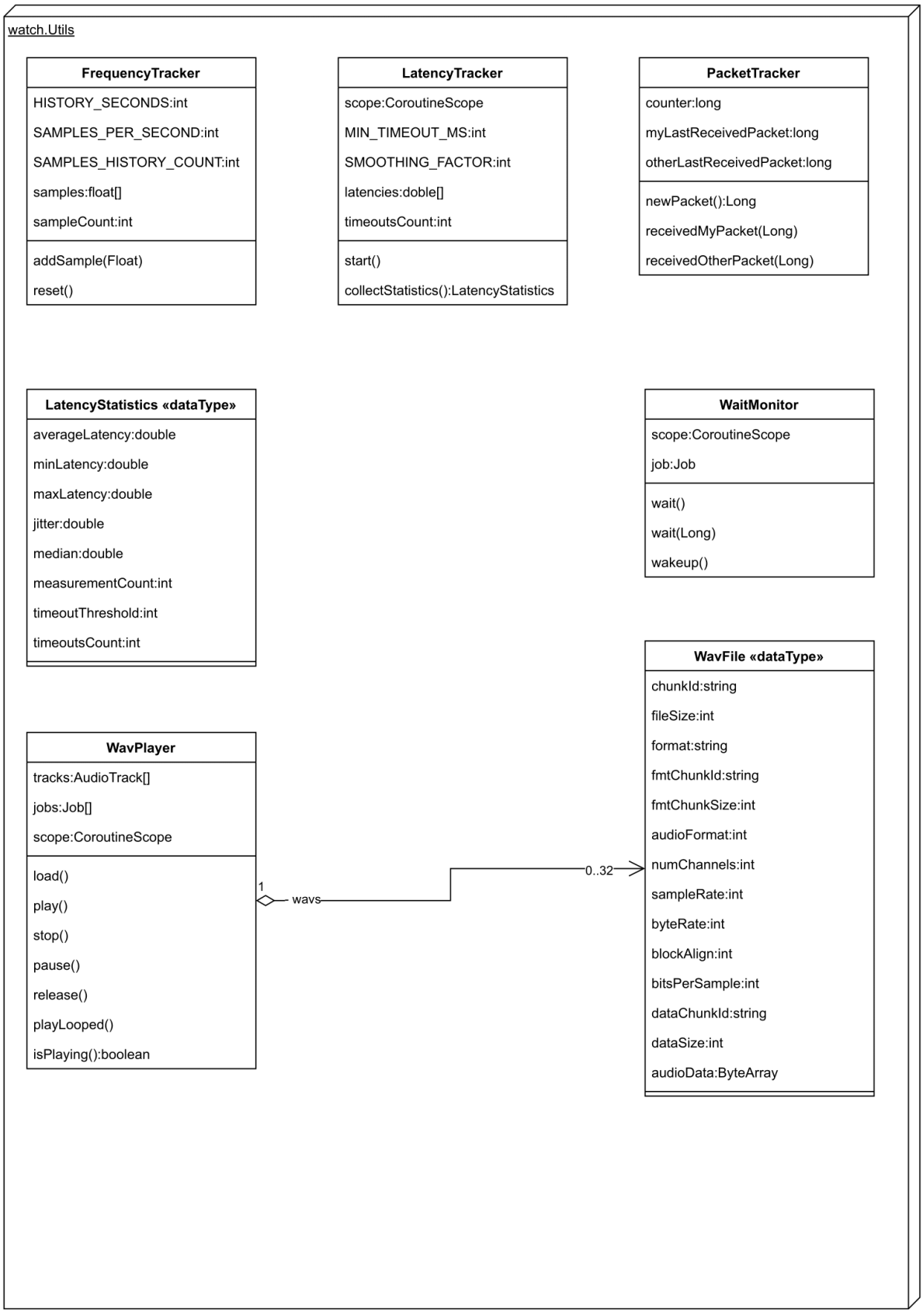


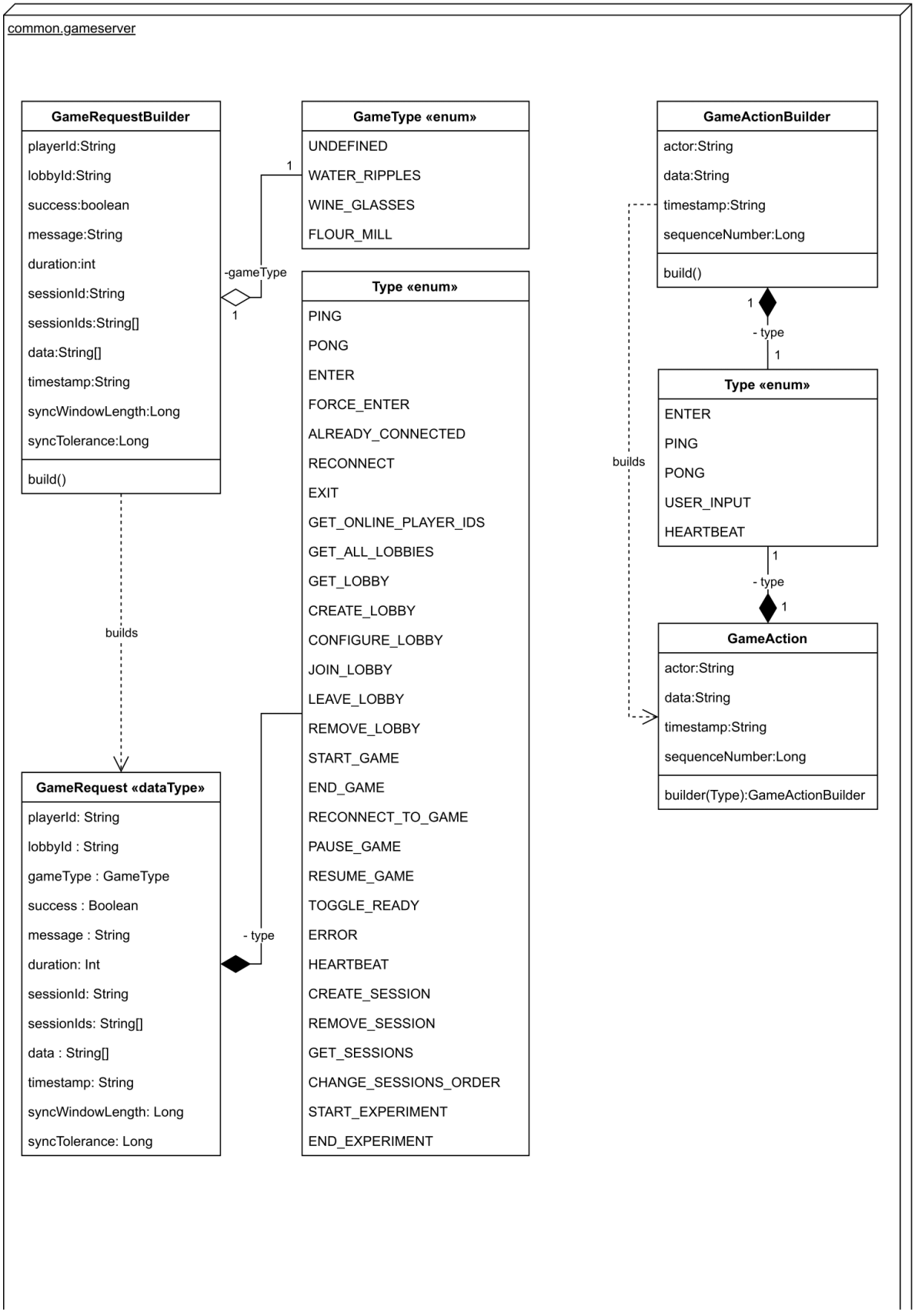

watch.ViewModel
WaterRipplesViewModel
clickVibration :VibrationEffect
sizeStep:float
click()
onCreate(Intent,Context)
handleGameAction(GameAction)
showRipple()
Arc
startAngle:long
angleSkew:long
direction:long
previousAngle:long
previousAngleDiff:long
currentAlpha:long
2
1
- myArc
-opponentArc
WineGlassesViewModel
myFrequencyTracker:FrequencyTracker
opponentFrequencyTracker:FrequencyTracker
inSync:boolean
onCreate(Intent,Context)
setTouchPoint(double,double)
inSync(): boolean
handleGameAction(GameAction)
1
- ripples
0..n
Ripple
color:Color
timestamp:long
updateAlphaStep()
Extends
Extends
MainViewModel
model:Model.MainModel
playerId:string
lobbyId:string
gameType:gameserver.GameType
ready:boolean
gameStartTime:long
connect()
enter(string)
exit()
afterGame()
1
1
GameViewModel
gameType:gameserver.GameType
vibrator:Vibrator
packetTracker:PacketTracker
latencyTracker:LatencyTracker
sensorsHandler:SensorsHandler
locationSensorHandler:LocationSensorsHandler
connect()
enter(string)
exit()
afterGame()
1
- state
1
1
- state
1
State «enum»
DISCONNECTED
SELECTING_ID
CONNECTING
CONNECTED_NOT_IN_LOBBY
CONNECTED_IN_LOBBY
IN_GAME
UPLOADING_EVENTS
ERROR
1
State «enum»
LOADING
PLAYING
TRYING_TO_RECONNECT
TERMINATED
ERROR
Extends

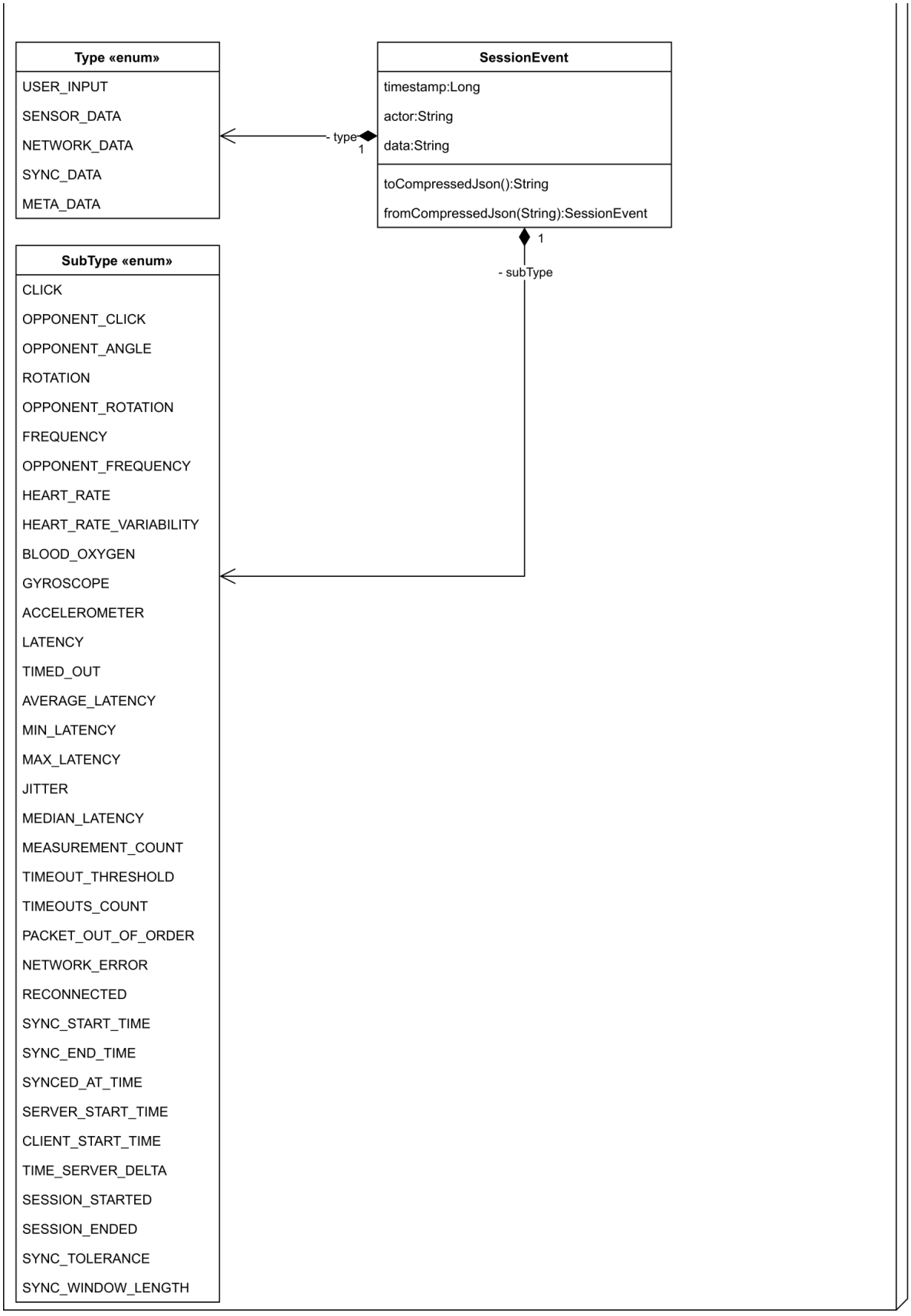

FlourMillViewModel
coolingDown:boolean
resetCooldownTime:long
leftSyncStatusObserved:boolean
rightSyncStatusObserved:boolean
leftLastRotation:long
rightLastRotation:boolean
isDragged():boolean
isCoolingDown():boolean
isSynced(AxleSide):boolean
rotateMyAxleEnd(Int):void
onCreate(Intent,Context):void
handleGameAction(GameAction):void
1
- axleSide
1
- axle
1
Axle
startingAngle:Angle
targetAngle:Angle
isRotating:boolean
effectiveAngle:Angle
- mySide
1
AxleSide
angle:float
LEFT
RIGHT
otherSide()
1

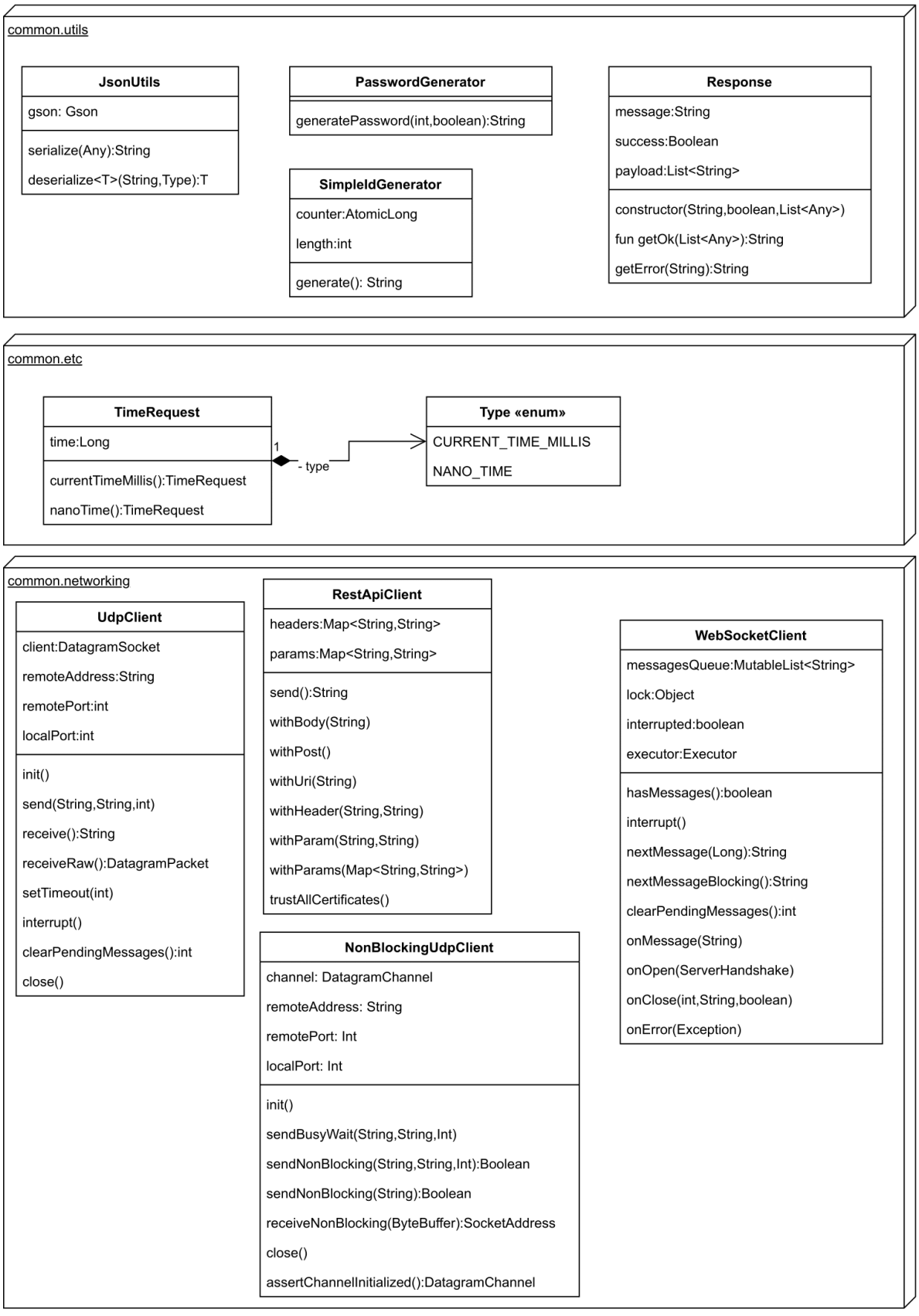

watch.View
MainActivity
viewModel:MainViewModel
onCreate(Bundle)
onDestroy()
registerActivities()
Main()
BlankScreen()
LoadingScreen(string)
PickingIdScreen()
ConnectedNotInLobbyScreen(string
ConnectedInLobbyScreen(string)
ErrorScreen(string)
GameActivity
gameType:gameserver.GameType
Main(GameViewModel)
LoadingScreen(string)
BlankScreen()
ErrorScreen(string)
Extends
Extends
Extends
WaterRipplesActivity
viewModel : WaterRipplesViewModel
soundPool: SoundPool
onCreate(Bundle)
onDestroy()
Main()
WaterRipples()
FlourMillActivity
viewModel:FlourMillViewModel
onCreate(Bundle)
Main()
FlourMill()
WineGlassesActivity
viewModel : WaterRipplesViewModel
sound: WavPlayer
onCreate(Bundle)
onDestroy()
Main()
WaterRipples()

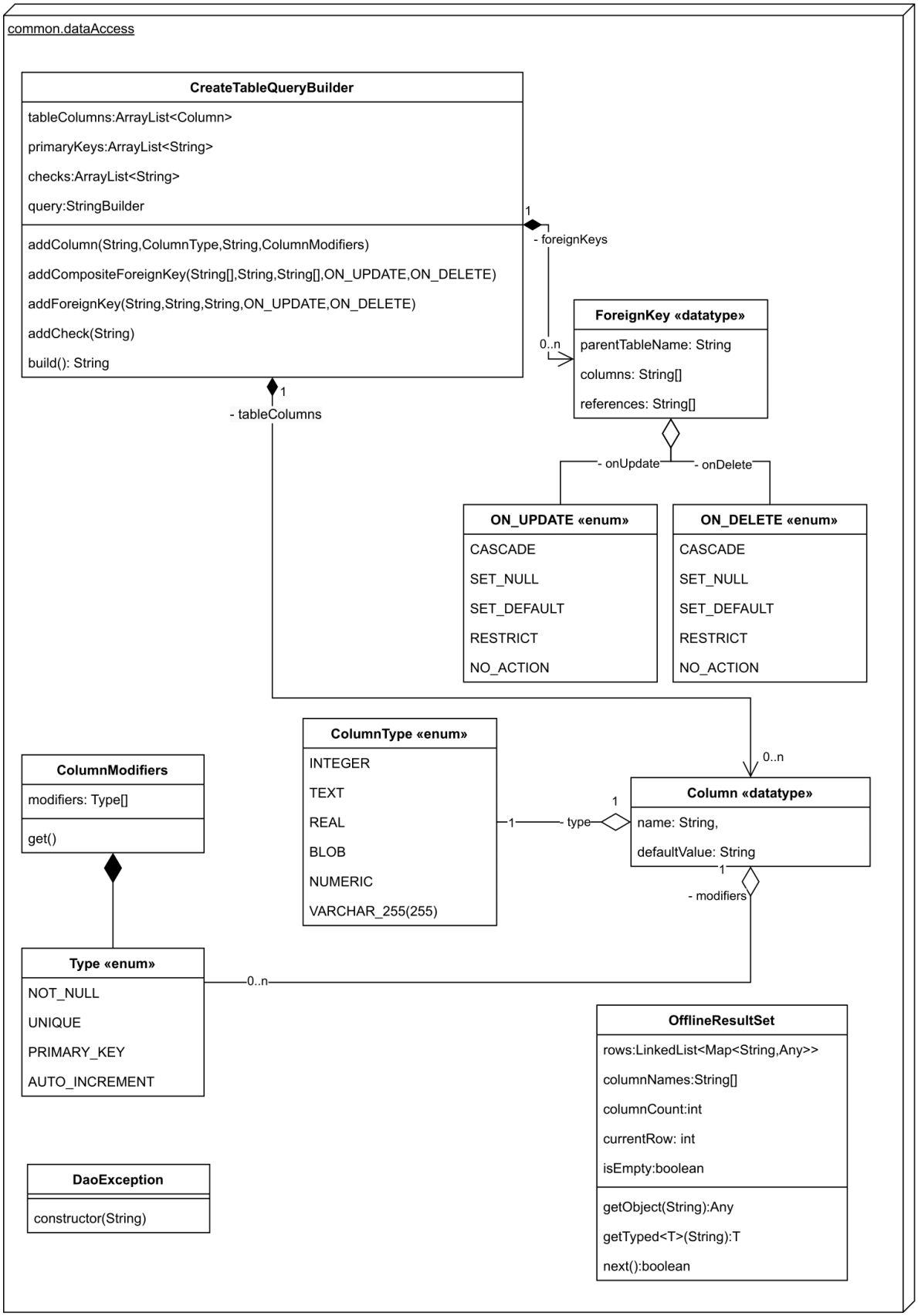

watch.Sensors
LocationSensorsHandler
sensorManager:SensorManager
gyroscopeSensor:Sensor
accelerometerSensor:Sensor
gyroscopeListener:SensorEventListener
accelerometerListener:SensorEventListener
start()
stop()
SensorsHandler
context:Context
gameViewModel:GameViewModel
gameViewModel:GameViewModel
start()
stop()
1
- trackerDataObserver
1
- heartRateListener
1
- connectionObserver
Use
connectionManager
1
ConnectionManager
connectionListener:ConnectionListener
healthTrackingService:HealthTrackingService
ConnectionManager(ConnectionObserver)
connect(Context)
disconnect()
initHeartRate(HeartRateListener)
- ConnectionObserver
1
1
ConnectionObserver «interface»
onConnectionResult()
Use
TrackerDataNotifier «singleton»
notifyHeartRateTrackerObservers()
notifySpO2TrackerObservers()
notify
HeartRateListener
- trackerEventListener:TrackerEventListener
readValuesFromDataPoint()
1
observers
1
0..n
TrackerDataObserver «interface»
onHeartRateTrackerDataChanged()
onSpO2TrackerDataChanged()
HeartRateData «dataType»
status:int
hr:int
ibi:int
qibi:int
BaseListener
- handler:Handler
- healthTracker:HealthTracker
- isHandlerRunning:Boolean
startTracker()
stopTracker()

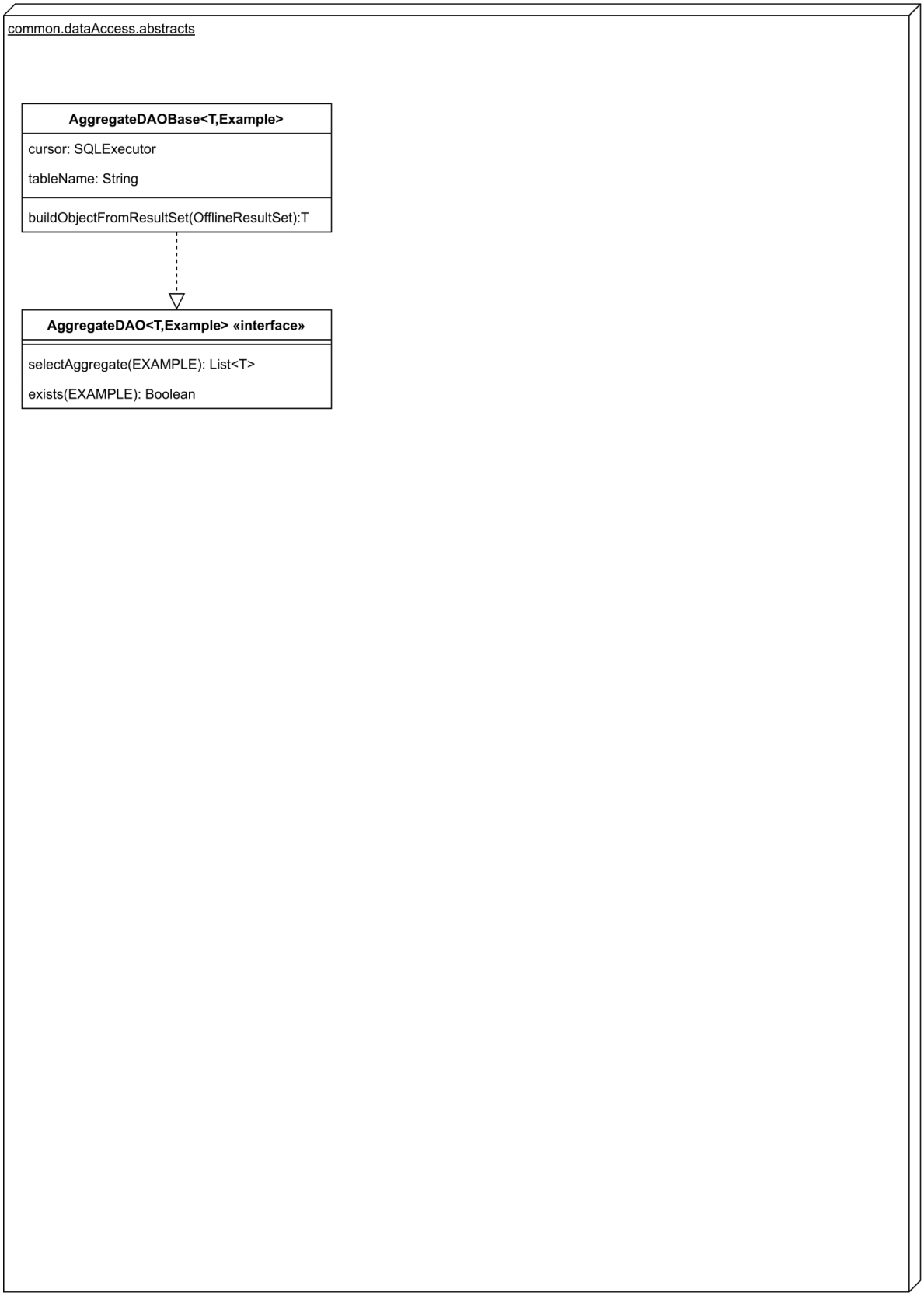

watch.Utils
FrequencyTracker
HISTORY_SECONDS:int
SAMPLES_PER_SECOND:int
SAMPLES_HISTORY_COUNT:int
samples:float[]
sampleCount:int
addSample(Float)
reset()
LatencyTracker
scope:CoroutineScope
MIN_TIMEOUT_MS:int
SMOOTHING_FACTOR:int
latencies:doble[]
timeoutsCount:int
start()
collectStatistics():LatencyStatistics
PacketTracker
counter:long
myLastReceivedPacket:long
otherLastReceivedPacket:long
newPacket():Long
receivedMyPacket(Long)
receivedOtherPacket(Long)
LatencyStatistics «dataType»
averageLatency:double
minLatency:double
maxLatency:double
jitter:double
median:double
measurementCount:int
timeoutThreshold:int
timeoutsCount:int
WaitMonitor
scope:CoroutineScope
job:Job
wait()
wait(Long)
wakeup()
WavFile «dataType»
chunkId:string
fileSize:int
format:string
fmtChunkId:string
fmtChunkSize:int
audioFormat:int
numChannels:int
sampleRate:int
byteRate:int
blockAlign:int
bitsPerSample:int
dataChunkId:string
dataSize:int
audioData:ByteArray
WavPlayer
tracks:AudioTrack[]
jobs:Job[]
scope:CoroutineScope
load()
play()
stop()
pause()
release()
playLooped()
isPlaying():boolean
1
wavs
0..32

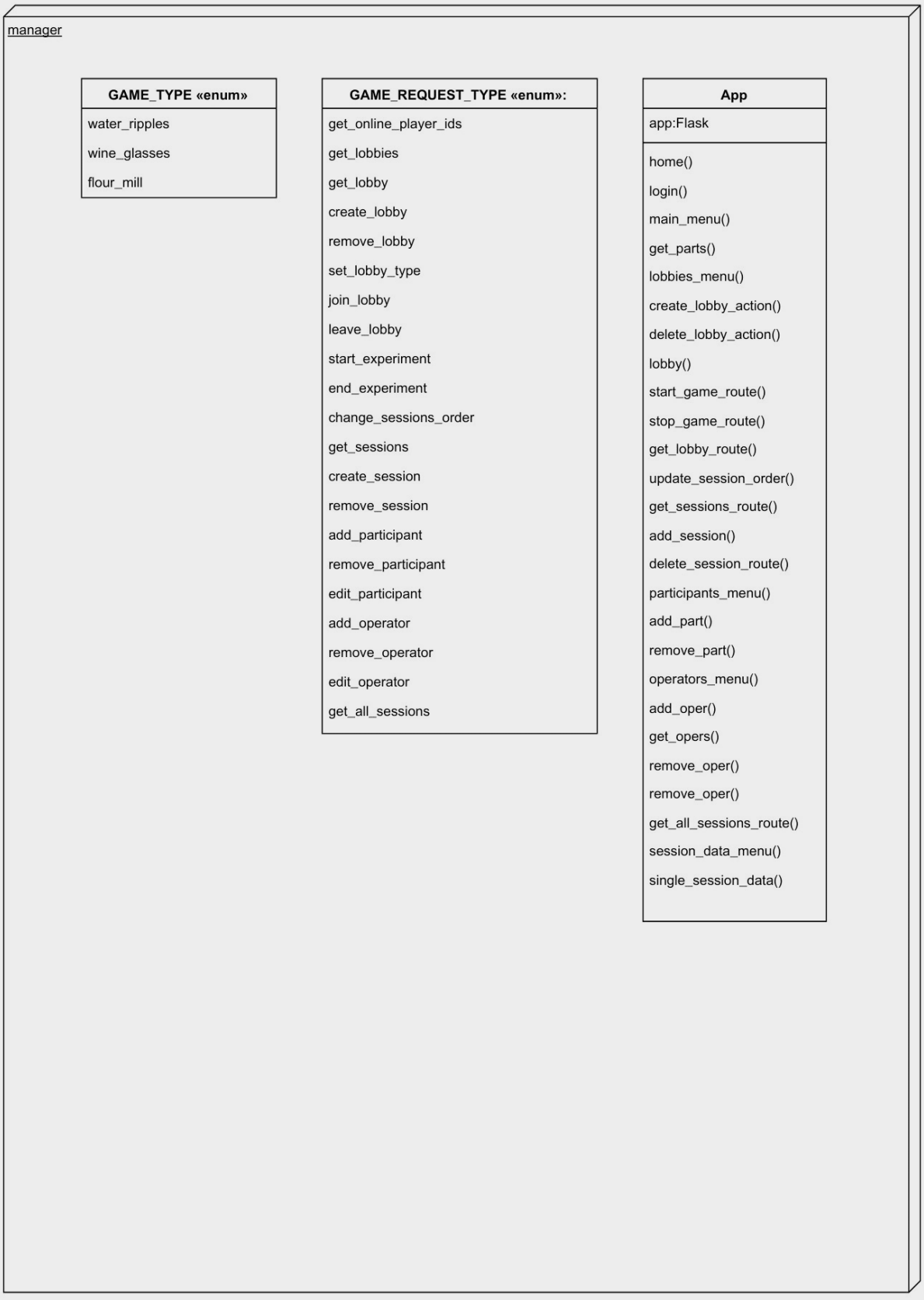

common.gameserver
GameRequestBuilder
playerId:String
lobbyId:String
success:boolean
message:String
duration:int
sessionId:String
sessionIds:String[]
data:String[]
timestamp:String
syncWindowLength:Long
syncTolerance:Long
build()
GameType «enum»
UNDEFINED
WATER_RIPPLES
WINE_GLASSES
FLOUR_MILL
1
-gameType
1
Type «enum»
PING
PONG
ENTER
FORCE_ENTER
ALREADY_CONNECTED
RECONNECT
EXIT
GET_ONLINE_PLAYER_IDS
GET_ALL_LOBBIES
GET_LOBBY
CREATE_LOBBY
CONFIGURE_LOBBY
JOIN_LOBBY
LEAVE_LOBBY
REMOVE_LOBBY
START_GAME
END_GAME
RECONNECT_TO_GAME
PAUSE_GAME
RESUME_GAME
TOGGLE_READY
ERROR
HEARTBEAT
CREATE_SESSION
REMOVE_SESSION
GET_SESSIONS
CHANGE_SESSIONS_ORDER
START_EXPERIMENT
END_EXPERIMENT
builds
GameRequest «dataType»
playerId: String
lobbyId : String
gameType : GameType
success : Boolean
message : String
duration: Int
sessionId: String
sessionIds: String[]
data : String[]
timestamp: String
syncWindowLength: Long
syncTolerance: Long
- type
GameActionBuilder
actor:String
data:String
timestamp:String
sequenceNumber:Long
build()
1
- type
1
Type «enum»
ENTER
PING
PONG
USER_INPUT
HEARTBEAT
builds
1
- type
1
GameAction
actor:String
data:String
timestamp:String
sequenceNumber:Long
builder(Type):GameActionBuilder

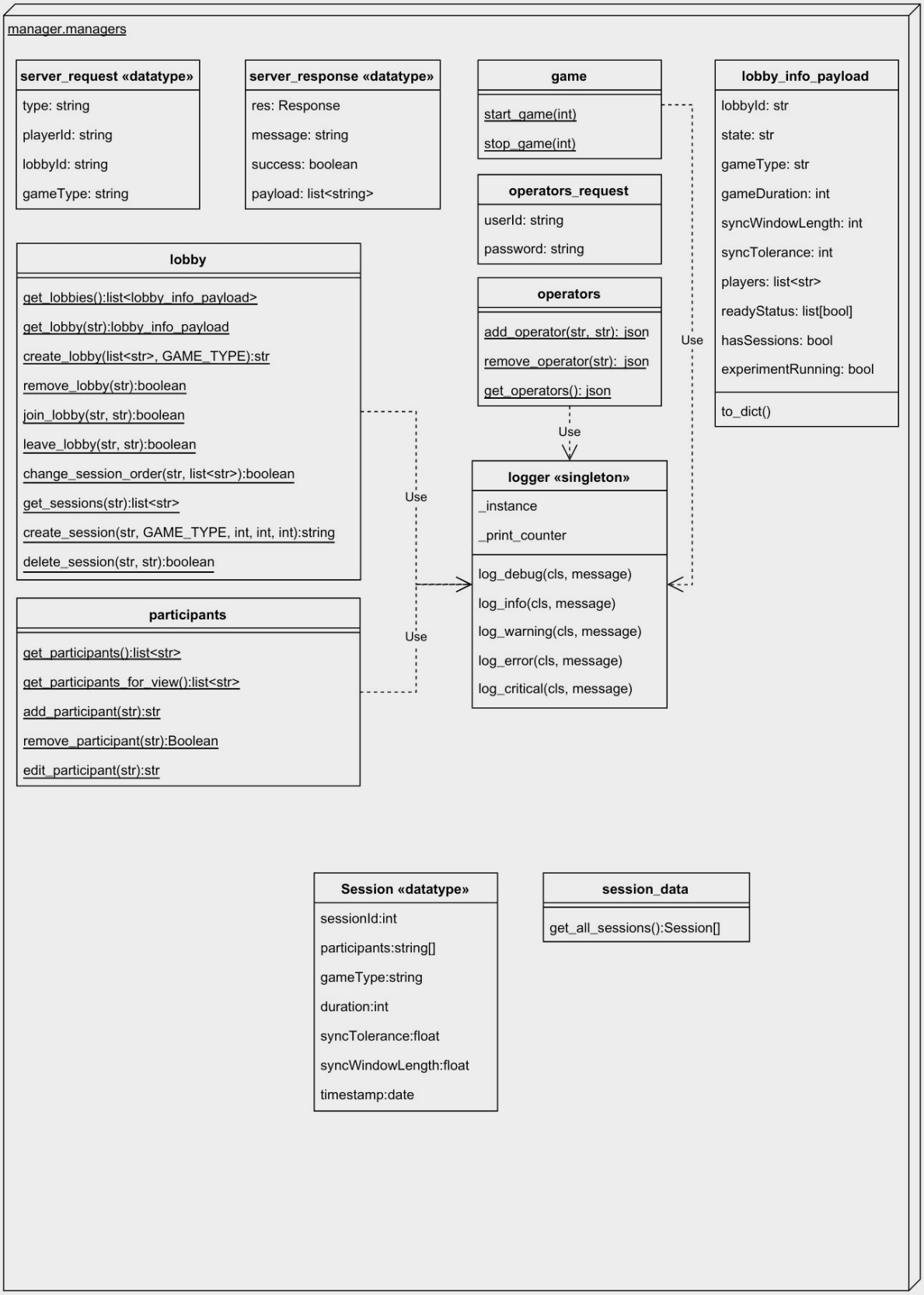

Type «enum»
USER_INPUT
SENSOR_DATA
NETWORK_DATA
SYNC_DATA
META_DATA
- type
1
SessionEvent
timestamp:Long
actor:String
data:String
toCompressedJson():String
fromCompressedJson(String):SessionEvent
1
- subType
SubType «enum»
CLICK
OPPONENT_CLICK
OPPONENT_ANGLE
ROTATION
OPPONENT_ROTATION
FREQUENCY
OPPONENT_FREQUENCY
HEART_RATE
HEART_RATE_VARIABILITY
BLOOD_OXYGEN
GYROSCOPE
ACCELEROMETER
LATENCY
TIMED_OUT
AVERAGE_LATENCY
MIN_LATENCY
MAX_LATENCY
JITTER
MEDIAN_LATENCY
MEASUREMENT_COUNT
TIMEOUT_THRESHOLD
TIMEOUTS_COUNT
PACKET_OUT_OF_ORDER
NETWORK_ERROR
RECONNECTED
SYNC_START_TIME
SYNC_END_TIME
SYNCED_AT_TIME
SERVER_START_TIME
CLIENT_START_TIME
TIME_SERVER_DELTA
SESSION_STARTED
SESSION_ENDED
SYNC_TOLERANCE
SYNC_WINDOW_LENGTH

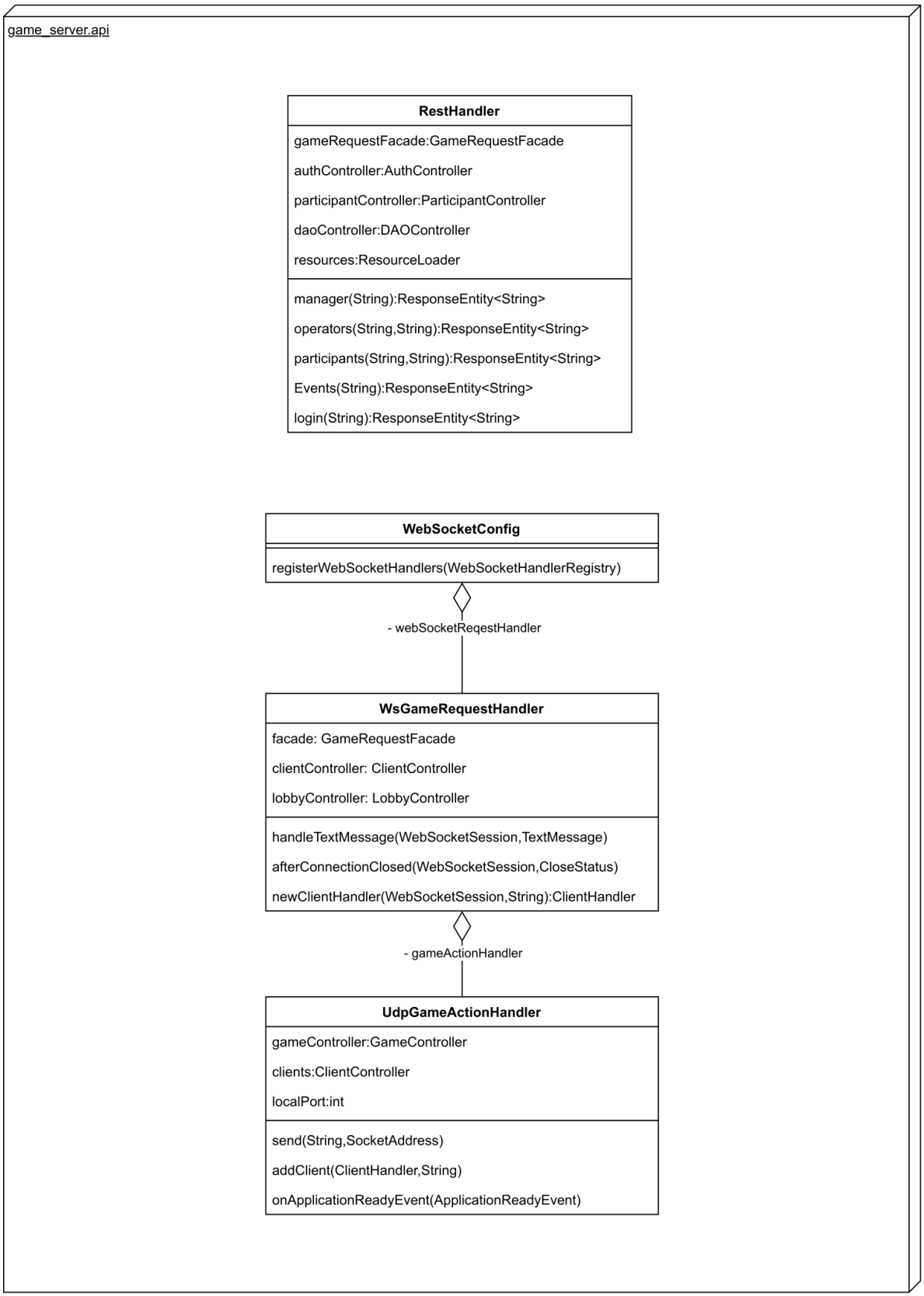

common.utils
JsonUtils
gson: Gson
serialize(Any):String
deserialize<T>(String,Type):T
PasswordGenerator
generatePassword(int,boolean):String
SimpleIdGenerator
counter:AtomicLong
length:int
generate(): String
Response
message:String
success:Boolean
payload:List<String>
constructor(String,boolean,List<Any>)
fun getOk(List<Any>):String
getError(String):String
common.etc
TimeRequest
time:Long
currentTimeMillis():TimeRequest
nanoTime():TimeRequest
1
- type
Type «enum»
CURRENT_TIME_MILLIS
NANO_TIME
common.networking
UdpClient
client:DatagramSocket
remoteAddress:String
remotePort:int
localPort:int
init()
send(String,String,int)
receive():String
receiveRaw():DatagramPacket
setTimeout(int)
interrupt()
clearPendingMessages():int
close()
RestApiClient
headers:Map<String,String>
params:Map<String,String>
send():String
withBody(String)
withPost()
withUri(String)
withHeader(String,String)
withParam(String,String)
withParams(Map<String,String>)
trustAllCertificates()
NonBlockingUdpClient
channel: DatagramChannel
remoteAddress: String
remotePort: Int
localPort: Int
init()
sendBusyWait(String,String,Int)
sendNonBlocking(String,String,Int):Boolean
sendNonBlocking(String):Boolean
receiveNonBlocking(ByteBuffer):SocketAddress
close()
assertChannelInitialized():DatagramChannel
WebSocketClient
messagesQueue:MutableList<String>
lock:Object
interrupted:boolean
executor:Executor
hasMessages():boolean
interrupt()
nextMessage(Long):String
nextMessageBlocking():String
clearPendingMessages():int
onMessage(String)
onOpen(ServerHandshake)
onClose(int,String,boolean)
onError(Exception)

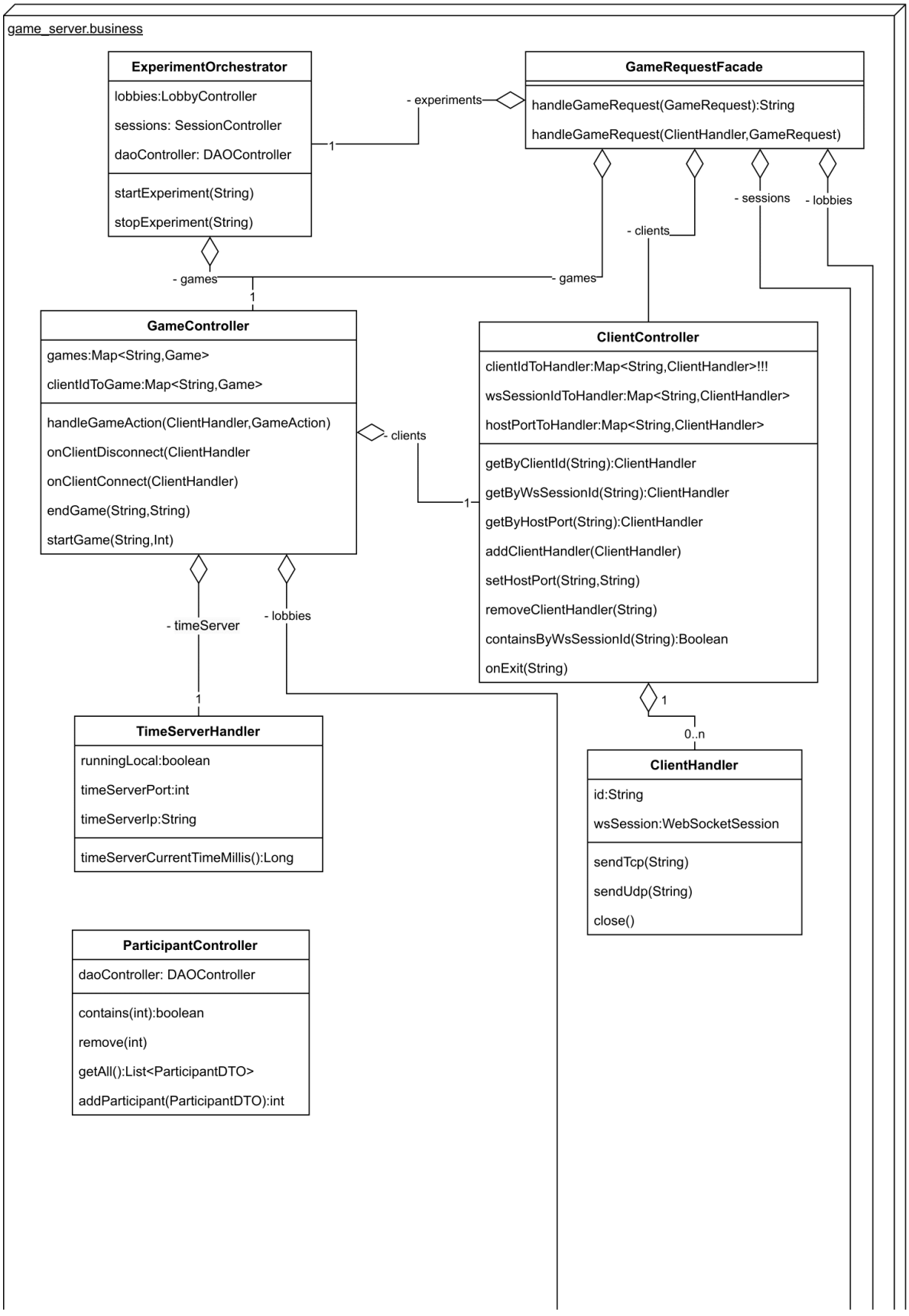

common.dataAccess
CreateTableQueryBuilder
tableColumns:ArrayList<Column>
primaryKeys:ArrayList<String>
checks:ArrayList<String>
query:StringBuilder
addColumn(String,ColumnType,String,ColumnModifiers)
addCompositeForeignKey(String[],String,String[],ON_UPDATE,ON_DELETE)
addForeignKey(String,String,String,ON_UPDATE,ON_DELETE)
addCheck(String)
build(): String
1
- foreignKeys
0..n
ForeignKey «datatype»
parentTableName: String
columns: String[]
references: String[]
1
- tableColumns
- onUpdate
- onDelete
ON_UPDATE «enum»
CASCADE
SET_NULL
SET_DEFAULT
RESTRICT
NO_ACTION
ON_DELETE «enum»
CASCADE
SET_NULL
SET_DEFAULT
RESTRICT
NO_ACTION
0..n
ColumnType «enum»
INTEGER
TEXT
REAL
BLOB
NUMERIC
VARCHAR_255(255)
1
- type
1
Column «datatype»
name: String,
defaultValue: String
1
- modifiers
ColumnModifiers
modifiers: Type[]
get()
Type «enum»
NOT_NULL
UNIQUE
PRIMARY_KEY
AUTO_INCREMENT
0..n
OfflineResultSet
rows:LinkedList<Map<String,Any>>
columnNames:String[]
columnCount:int
currentRow: int
isEmpty:boolean
getObject(String):Any
getTyped<T>(String):T
next():boolean
DaoException
constructor(String)

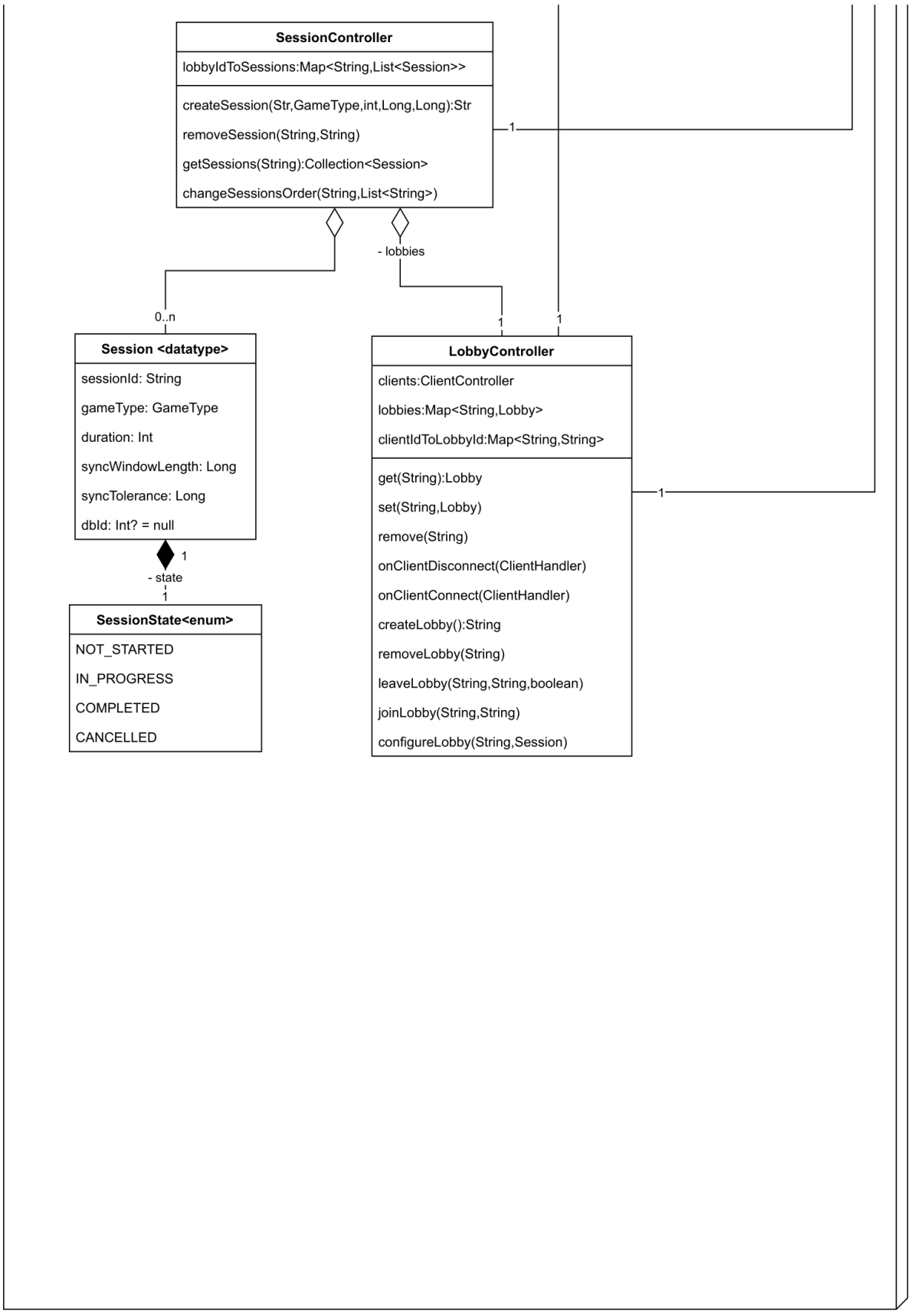

common.dataAccess.abstracts
AggregateDAOBase<T,Example>
cursor: SQLExecutor
tableName: String
buildObjectFromResultSet(OfflineResultSet):T
AggregateDAO<T,Example> «interface»
selectAggregate(EXAMPLE): List<T>
exists(EXAMPLE): Boolean


manager
GAME_TYPE «enum»
water_ripples
wine_glasses
flour_mill
GAME_REQUEST_TYPE «enum»:
get_online_player_ids
get_lobbies
get_lobby
create_lobby
remove_lobby
set_lobby_type
join_lobby
leave_lobby
start_experiment
end_experiment
change_sessions_order
get_sessions
create_session
remove_session
add_participant
remove_participant
edit_participant
add_operator
remove_operator
edit_operator
get_all_sessions
App
app:Flask
home()
login()
main_menu()
get_parts()
lobbies_menu()
create_lobby_action()
delete_lobby_action()
lobby()
start_game_route()
stop_game_route()
get_lobby_route()
update_session_order()
get_sessions_route()
add_session()
delete_session_route()
participants_menu()
add_part()
remove_part()
operators_menu()
add_oper()
get_opers()
remove_oper()
remove_oper()
get_all_sessions_route()
session_data_menu()
single_session_data()

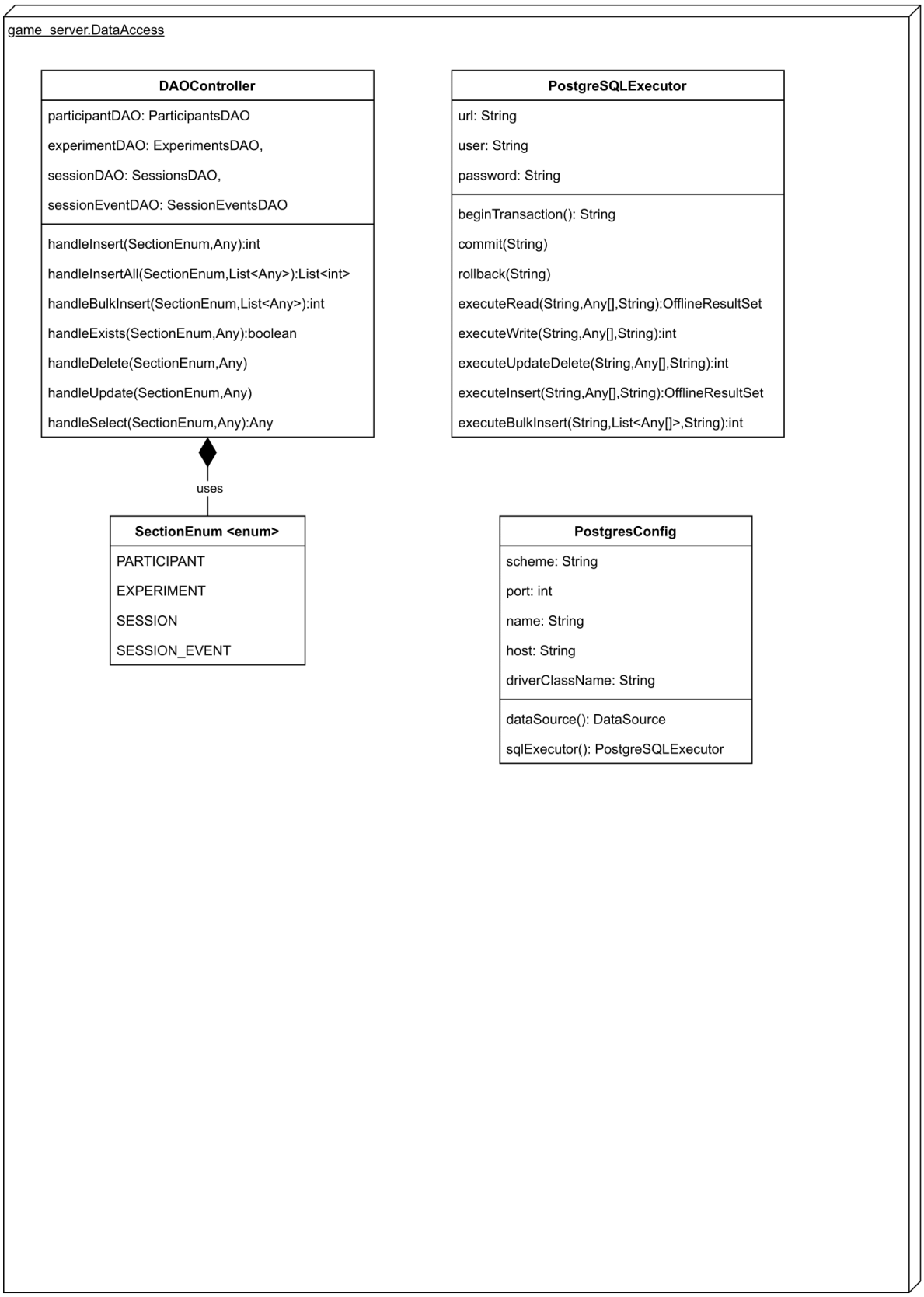

manager.managers
server_request «datatype»
type: string
playerId: string
lobbyId: string
gameType: string
server_response «datatype»
res: Response
message: string
success: boolean
payload: list<string>
game
start_game(int)
stop_game(int)
lobby_info_payload
lobbyId: str
state: str
gameType: str
gameDuration: int
syncWindowLength: int
syncTolerance: int
players: list<str>
readyStatus: list[bool]
hasSessions: bool
experimentRunning: bool
to_dict()
operators_request
userId: string
password: string
lobby
get_lobbies():list<lobby_info_payload>
get_lobby(str):lobby_info_payload
create_lobby(list<str>, GAME_TYPE):str
remove_lobby(str):boolean
join_lobby(str, str):boolean
leave_lobby(str, str):boolean
change_session_order(str, list<str>):boolean
get_sessions(str):list<str>
create_session(str, GAME_TYPE, int, int, int):string
delete_session(str, str):boolean
operators
add_operator(str, str): json
remove_operator(str): json
get_operators(): json
Use
Use
Use
Use
logger «singleton»
_instance
_print_counter
log_debug(cls, message)
log_info(cls, message)
log_warning(cls, message)
log_error(cls, message)
log_critical(cls, message)
participants
get_participants():list<str>
get_participants_for_view():list<str>
add_participant(str):str
remove_participant(str):Boolean
edit_participant(str):str
Session «datatype»
sessionId:int
participants:string[]
gameType:string
duration:int
syncTolerance:float
syncWindowLength:float
timestamp:date
session_data
get_all_sessions():Session[]

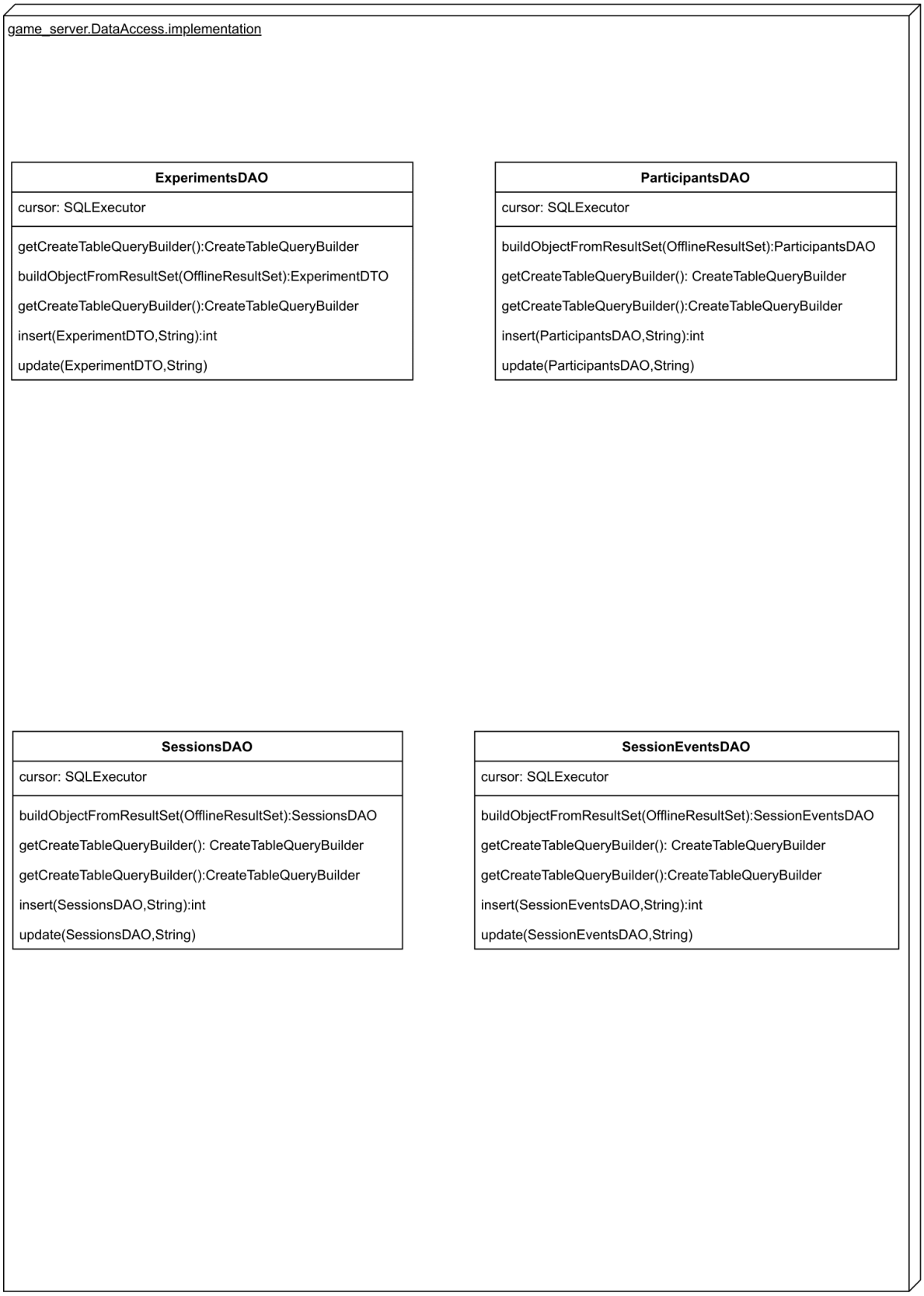

game_server.api
RestHandler
gameRequestFacade:GameRequestFacade
authController:AuthController
participantController:ParticipantController
daoController:DAOController
resources:ResourceLoader
manager(String):ResponseEntity<String>
operators(String,String):ResponseEntity<String>
participants(String,String):ResponseEntity<String>
Events(String):ResponseEntity<String>
login(String):ResponseEntity<String>
WebSocketConfig
registerWebSocketHandlers(WebSocketHandlerRegistry)
- webSocketReqestHandler
WsGameRequestHandler
facade: GameRequestFacade
clientController: ClientController
lobbyController: LobbyController
handleTextMessage(WebSocketSession,TextMessage)
afterConnectionClosed(WebSocketSession,CloseStatus)
newClientHandler(WebSocketSession,String):ClientHandler
- gameActionHandler
UdpGameActionHandler
gameController:GameController
clients:ClientController
localPort:int
send(String,SocketAddress)
addClient(ClientHandler,String)
onApplicationReadyEvent(ApplicationReadyEvent)


game_server.business
ExperimentOrchestrator
lobbies:LobbyController
sessions: SessionController
daoController: DAOController
startExperiment(String)
stopExperiment(String)
- experiments
1
GameRequestFacade
handleGameRequest(GameRequest):String
handleGameRequest(ClientHandler,GameRequest)
- sessions
- lobbies
- clients
- games
- games
1
GameController
games:Map<String,Game>
clientIdToGame:Map<String,Game>
handleGameAction(ClientHandler,GameAction)
onClientDisconnect(ClientHandler
onClientConnect(ClientHandler)
endGame(String,String)
startGame(String,Int)
- clients
1
ClientController
clientIdToHandler:Map<String,ClientHandler>!!!
wsSessionIdToHandler:Map<String,ClientHandler>
hostPortToHandler:Map<String,ClientHandler>
getByClientId(String):ClientHandler
getByWsSessionId(String):ClientHandler
getByHostPort(String):ClientHandler
addClientHandler(ClientHandler)
setHostPort(String,String)
removeClientHandler(String)
containsByWsSessionId(String):Boolean
onExit(String)
- timeServer
- lobbies
1
1
0..n
TimeServerHandler
runningLocal:boolean
timeServerPort:int
timeServerIp:String
timeServerCurrentTimeMillis():Long
ClientHandler
id:String
wsSession:WebSocketSession
sendTcp(String)
sendUdp(String)
close()
ParticipantController
daoController: DAOController
contains(int):boolean
remove(int)
getAll():List<ParticipantDTO>
addParticipant(ParticipantDTO):int


SessionController
lobbyIdToSessions:Map<String,List<Session>>
createSession(Str,GameType,int,Long,Long):Str
removeSession(String,String)
getSessions(String):Collection<Session>
changeSessionsOrder(String,List<String>)
1
- lobbies
0..n
1
1
Session <datatype>
sessionId: String
gameType: GameType
duration: Int
syncWindowLength: Long
syncTolerance: Long
dbId: Int? = null
1
- state
1
SessionState<enum>
NOT_STARTED
IN_PROGRESS
COMPLETED
CANCELLED
LobbyController
clients:ClientController
lobbies:Map<String,Lobby>
clientIdToLobbyId:Map<String,String>
get(String):Lobby
set(String,Lobby)
remove(String)
onClientDisconnect(ClientHandler)
onClientConnect(ClientHandler)
createLobby():String
removeLobby(String)
leaveLobby(String,String,boolean)
joinLobby(String,String)
configureLobby(String,Session)
1


game_server.business.auth
AuthController
userIdToUUID:Map<String,String>
encoder:PasswordEncoder
authenticateUser(String,String):String
revokeAuthentication(String):boolean
validateTokenOwnership(String,String):boolean
validateTokenAuthenticity(String):boolean
createUser(Credentials)
deleteUser(String)
CredentialsController
credentials:Map<String,Credentials>
get(String):Credentials
contains(String):boolean
set(String,Credentials)
getAll():List<String>
remove(String):boolean
Credentials <datatype>
userId:String
password:String
token:String
newPassword:String
0..n
SecurityConfig
authController:AuthController
securityFilterChain(HttpSecurity):SecurityFilterChain
game_server.business.games
WaterRipplesGame
lobbyId:String,
player1:ClientHandler
player2:ClientHandler
handleGameAction(ClientHandler,GameAction)
WineGlassesGame
lobbyId:String,
player1:ClientHandler
player2:ClientHandler
handleGameAction(ClientHandler,GameAction)
Game <abstract>
lobbyId:String
player1:ClientHandler
player2:ClientHandler
handleGameAction(ClientHandler,GameAction)
startGame(Long,int)
endGame(String)
sendGameAction(GameAction)
game_server.business.lobbies
LobbyState <enum>
WAITING
PLAYING
1
- state
1
LobbyInfo <datatype>
lobbyId: String,
gameType: GameType
gameDuration: Int
syncWindowLength: Long
syncTolerance: Long
players: List<String>
readyStatus: List<Boolean>
hasSessions: Boolean
experimentRunning: Boolean
Lobby
id: String
gameType: GameType
gameDuration: int
syncTolerance: long
syncWindowLength: long
player1Id : String
player2Id : String
add(player: String): boolean
remove(player: String): boolean
isReady(): boolean


game_server.DataAccess
DAOController
participantDAO: ParticipantsDAO
experimentDAO: ExperimentsDAO,
sessionDAO: SessionsDAO,
sessionEventDAO: SessionEventsDAO
handleInsert(SectionEnum,Any):int
handleInsertAll(SectionEnum,List<Any>):List<int>
handleBulkInsert(SectionEnum,List<Any>):int
handleExists(SectionEnum,Any):boolean
handleDelete(SectionEnum,Any)
handleUpdate(SectionEnum,Any)
handleSelect(SectionEnum,Any):Any
PostgreSQLExecutor
url: String
user: String
password: String
beginTransaction(): String
commit(String)
rollback(String)
executeRead(String,Any[],String):OfflineResultSet
executeWrite(String,Any[],String):int
executeUpdateDelete(String,Any[],String):int
executeInsert(String,Any[],String):OfflineResultSet
executeBulkInsert(String,List<Any[]>,String):int
uses
SectionEnum <enum>
PARTICIPANT
EXPERIMENT
SESSION
SESSION_EVENT
PostgresConfig
scheme: String
port: int
name: String
host: String
driverClassName: String
dataSource(): DataSource
sqlExecutor(): PostgreSQLExecutor


game_server.DataAccess.implementation
ExperimentsDAO
cursor: SQLExecutor
getCreateTableQueryBuilder():CreateTableQueryBuilder
buildObjectFromResultSet(OfflineResultSet):ExperimentDTO
getCreateTableQueryBuilder():CreateTableQueryBuilder
insert(ExperimentDTO,String):int
update(ExperimentDTO,String)
ParticipantsDAO
cursor: SQLExecutor
buildObjectFromResultSet(OfflineResultSet):ParticipantsDAO
getCreateTableQueryBuilder(): CreateTableQueryBuilder
getCreateTableQueryBuilder():CreateTableQueryBuilder
insert(ParticipantsDAO,String):int
update(ParticipantsDAO,String)
SessionsDAO
cursor: SQLExecutor
buildObjectFromResultSet(OfflineResultSet):SessionsDAO
getCreateTableQueryBuilder(): CreateTableQueryBuilder
getCreateTableQueryBuilder():CreateTableQueryBuilder
insert(SessionsDAO,String):int
update(SessionsDAO,String)
SessionEventsDAO
cursor: SQLExecutor
buildObjectFromResultSet(OfflineResultSet):SessionEventsDAO
getCreateTableQueryBuilder(): CreateTableQueryBuilder
getCreateTableQueryBuilder():CreateTableQueryBuilder
insert(SessionEventsDAO,String):int
update(SessionEventsDAO,String)

5.3 Class Description

1. Game Server

1.1. Api

RestHandler

Handles HTTP REST requests, routing them to the appropriate business logic components and managing responses.

UdpGameActionHandler

Processes incoming UDP game action packets for low-latency communication in real-time gameplay scenarios.

WsGameRequestHandler

Manages WebSocket connections, handling real-time game requests and ensuring efficient bidirectional communication.

1.2. dataAccess

DAOController

Coordinates data access operations across various DAOs, serving as a centralized interface for database interactions.

PostgreSQLExecutor

Executes SQL queries and transactions on a PostgreSQL database, handling connection management and error handling.

PostgresConfig

Manages configuration settings required to establish and maintain PostgreSQL database connections.

1.2.1. models

ParticipantDTO

Data Transfer Object representing participant information, facilitating data exchange between the server and database.

ExperimentDTO

Encapsulates experiment-related data for database operations and inter-module communication.

SessionDTO

Represents session-specific data, including participant details and session state for persistence and retrieval.

SessionEventDTO

Models session event data, capturing specific actions or occurrences during a game session.

ExpWithSessionsData

Aggregates experiment data with its related session information for comprehensive data handling.

SessionData

Holds detailed session-related information, including event history and participant states.

1.2.2. implementations

ExperimentsDAO

Handles CRUD operations for experiment-related data within the database.

ParticipantsDAO

Manages database transactions related to participant records.

SessionEventsDAO

Provides data access functionality for session events, including logging and retrieval.

SessionsDAO

Facilitates database operations for game sessions, managing session lifecycle data.

1.3. business

ClientController

Manages client connections, handling registration, disconnection, and client-related requests.

ClientHandler

Processes client-specific actions, managing data flow between clients and the server.

Session

Represents an active game session, maintaining its state, participants, and session-related events.

SessionState

Defines the current status of a session, tracking transitions like active, paused, or completed.

ExperimentOrchestrator

Coordinates the execution of experiments, managing the flow between sessions, participants, and data collection.

GameController

Central component for controlling game logic, handling game states, actions, and interactions.

GameRequestFacade

Provides a simplified interface for handling game requests, abstracting complex business logic from API handlers.

LobbyController

Manages game lobbies, including player matchmaking, lobby creation, and state transitions.

ParticipantController

Handles participant-related operations such as registration, authentication, and data management.

SessionController

Oversees the lifecycle of game sessions, from creation and management to termination.

TimeServerHandler

Synchronizes server time with clients to ensure consistent timing across distributed game sessions.

1.3.1. auth

AuthController

Manages authentication processes, including login, token validation, and access control.

Credentials

Represents user credentials, storing authentication data like usernames, passwords, or tokens securely.

CredentialsController

Handles credential-related operations, such as creation, validation, and updates.

SecurityConfig

Defines security settings and protocols, including encryption configurations and authentication policies.

1.3.2. games

Game

Abstract base class representing common properties and behaviors shared across all games.

WaterRipplesGame

Specific implementation of a game based on water ripple effects, managing its unique mechanics and logic.

WineGlassesGame

Game implementation focused on wine glass interactions, handling rules, states, and player actions.

1.3.3. lobbies

Lobby

Represents a game lobby where players gather before starting a session, managing participants and settings.

LobbyInfo

Contains metadata about a lobby, such as participant count, status, and configuration details.

LobbyState

Tracks the current state of a lobby, including whether it's open, full, or in-game.

2. Manager

GAME_TYPE

Enumeration or configuration defining different types of games supported by the system.

GAME_REQUEST_TYPE

Defines various request types that can be sent to games, such as start, pause, or player actions.

App

The main entry point for the Manager module, initializing core components and managing application lifecycle events.

2.1. Managers

server_request

Handles the creation and management of requests sent from the Manager to the game server.

Lobby

Manages lobby-related operations from the Manager's perspective, such as creation, updates, and participant tracking.

Participants

Oversees participant management, including registration, status monitoring, and data synchronization.

server_response

Processes responses received from the game server, interpreting and dispatching them to the appropriate handlers.

Game

Coordinates game-related activities within the Manager, such as tracking active games and managing game states.

operators_request

Handles requests initiated by operators (admins or moderators), facilitating control over sessions, games, and participants.

Operators

Manages operator accounts, permissions, and interactions with the game environment.

Logger

Provides logging functionality for tracking system events, errors, and operational metrics.

lobby_info_payload

Defines the structure of data packets containing lobby information, used for communication between server and Manager.

Session

A data object contains relevant information about a single session, for display in a graph on the manager's console.

Session_Data

A functional class for getting the sessions data from the DB, and transform to the relevant UI class.

3. Common

3.1. dataAccess

CreateTableQueryBuilder

Generates SQL queries for creating database tables dynamically based on specified schemas and constraints.

ForeignKey

Represents foreign key relationships between database tables, enforcing referential integrity.

ON_UPDATE

Defines the behavior of foreign keys when the referenced data is updated, such as cascading changes.

ON_DELETE

Specifies actions to be taken when referenced data is deleted, like setting null values or cascading deletions.

ColumnModifiers

Contains modifiers for database columns, such as NOT NULL, UNIQUE, or AUTO_INCREMENT.

ColumnType

Defines data types for database columns, like INTEGER, VARCHAR, or BOOLEAN.

Column

Represents a database table column, including its name, data type, and constraints.

Type

General representation of data types used within the database access layer.

OfflineResultSet

Stores query results that can be accessed offline, allowing data manipulation without a live database connection.

DaoException

Custom exception class for handling database access-related errors and failures.

3.1.1. abstracts

AggregateDAOBase

Abstract base class defining common operations for aggregate Data Access Objects (DAOs).

AggregateDAO

Extends AggregateDAOBase to implement data aggregation logic, handling complex queries and data processing.

3.2. networking

UdpClient

Manages UDP-based network communication, optimized for low-latency data transmission.

RestApiClient

Handles HTTP-based REST API requests, including sending, receiving, and processing responses.

WebSocketClient

Manages real-time, full-duplex communication over WebSockets for interactive client-server exchanges.

NonBlockingUdpClient

A UDP client designed for non-blocking operations, allowing asynchronous data handling without performance bottlenecks.

3.3. etc

TimeRequest

Represents a request to synchronize or retrieve time-related information from the server.

Type

Generic type identifier used for classifying data or actions within miscellaneous components.

3.4. utils

JsonUtils

Provides utility functions for serializing and deserializing JSON data.

PasswordGenerator

Generates secure, random passwords based on customizable rules and complexity requirements.

Response

Standardizes the structure of responses, including status codes, messages, and data payloads.

SimpleIdGenerator

Creates unique identifiers for sessions, participants, or other entities in the system.

3.5. gameserver

GameRequestBuilder

Facilitates the construction of game-related requests, ensuring correct formatting and data structure.

GameType

Defines various types of games available on the server, used for categorization and game logic routing.

GameRequest

Represents a request to perform an action in a game, such as moving a player or starting a session.

Type (GameRequest)

Specifies different request types within the GameRequest context, like join, leave, or action.

GameActionBuilder

Constructs game actions with the necessary parameters, making it easier to manage complex actions.

Type (GameActionBuilder)

Identifies action types for GameActionBuilder, such as player movement, item usage, or interactions.

GameAction

Encapsulates an action performed within a game, containing details like the actor, target, and action type.

SessionEvent

Represents events occurring during a session, such as player joins, disconnects, or in-game milestones.

Type (SessionEvent)

Classifies session events based on their nature, like connection events, gameplay milestones, or system notifications.

SubType

Provides further granularity to SessionEvent types, distinguishing between specific event categories.

4. Watch

4.1. Sensors

LocationSensorsHandler

Manages the collection and processing of spatial data from gyroscope and accelerometer sensors.

SensorsHandler

Acts as a central controller for handling the physical types of sensor data, such as heart rate, and IBI.

ConnectionManager

Oversees the connection lifecycle between sensors and the system, ensuring stable data transmission.

ConnectionObserver

Monitors the status of sensor connections, triggering alerts or actions when connection changes occur.

HeartRateListener

Captures heart rate data in real-time from the smartwatch and parse the data to HeartRateDate.

TrackerDataNotifier

Notifies registered components about new tracker data, ensuring timely updates across the system.

TrackerDataObserver

Observes tracker data changes and performs actions based on the received information.

HeartRateData

Datatype represents structured heart rate data, including all the measurement values.

BaseListener

Abstract class providing a base structure for implementing specific sensor data listeners.

4.2. Utils

FrequencyTracker

Tracks the frequency of sensor data updates, helping to monitor performance and detect anomalies.

LatencyTracker

Measures data transmission latency, assisting in network performance analysis.

PacketTracker

Monitors data packets for loss, delays, or errors during communication between devices.

LatencyStatistics

Provides statistical insights into latency trends, such as average delay, variance, and spikes.

WaitMonitor

Manages wait conditions, ensuring proper synchronization between asynchronous sensor events.

WavPlayer

Handles playback of .wav audio files, often used for alerts or feedback sounds.

WavFile

Represents a .wav audio file, including metadata like duration, sample rate, and file size.

4.3. Model

MainModel

Core data model managing the state and logic of the application, acting as the central data hub.

SessionEventCollector

Interface defining methods for collecting session-related events from various sources.

SessionEventCollectorImpl

Concrete implementation of SessionEventCollector, handling the actual collection and storage of session events.

4.4. ViewModel

MainViewModel

Connects the UI with the MainModel, managing application-wide UI states and data bindings.

State (MainViewModel)

Represents different UI states managed by the MainViewModel, such as loading, error, or active states.

GameViewModel

Manages the UI logic specific to game-related activities, including real-time updates and player interactions.

State (GameViewModel)

Defines various game states, such as in-progress, paused, or completed, for use in the GameViewModel.

WaterRipplesViewModel

Handles UI interactions and data processing for the Water Ripples game, including ripple animations and effects.

Ripple

Represents an individual ripple effect within the Water Ripples game, with properties like size, intensity, and lifespan.

WineGlassesViewModel

Manages game logic and UI state for the Wine Glasses game, including sound generation and visual feedback.

Arc

Represents an arc-shaped UI element used in the Wine Glasses game, likely to visualize sound waves or gestures.

FlourMillViewModel

Responsible for transmitting relevant information about the axle display and rotational feedback for the Flour Mill game.

Axle

Holds the state of the axle currently showing on the screen.

AxleSide

Represents a binary information of the current side of the axle (Left/Right).

4.5. View

MainActivity

The main entry point of the application's UI, hosting primary navigation and core screens.

GameActivity

Hosts the gameplay interface, managing real-time interactions, player input, and rendering.

WaterRipplesActivity

Dedicated activity for the Water Ripples game, handling its unique UI and sensor interactions.

WineGlassesActivity

Manages the UI and interactions specific to the Wine Glasses game, including sound-based gameplay mechanics.

WineGlassesActivity

Manages the UI and interactions specific to the Wine Glasses game, including sound-based gameplay mechanics.

FlourMillActivity

Manages the display of the flour mill, with all its intricacies, including external displays and haptics.

5.4 Unit Testing

Game server business logic – Unit test cases

ClientController

getByClientId()

GIVEN existing client WHEN getByClientId is called THEN correct client handler should be returned

GIVEN non-existing client WHEN getByClientId is called THEN null should be returned

getByWsSessionId()

GIVEN existing session ID WHEN getByWsSessionId is called THEN correct client handler should be returned

GIVEN non-existing session ID WHEN getByWsSessionId is called THEN null should be returned

getByHostPort()

GIVEN existing host port WHEN getByHostPort is called THEN correct client handler should be returned

GIVEN non-existing host port WHEN getByHostPort is called THEN null should be returned

addClientHandler()

GIVEN new client handler WHEN addClientHandler is called THEN client should be added successfully

GIVEN duplicate client handler WHEN addClientHandler is called THEN existing handler should be replaced

GIVEN existing client WHEN removeClientHandler is called THEN client should be removed successfully

GIVEN non-existing client WHEN removeClientHandler is called THEN no action should be taken

setHostPort()

GIVEN valid client and host port WHEN setHostPort is called THEN host port should be associated with client

GIVEN non-existing client WHEN setHostPort is called THEN exception should be thrown

GIVEN existing host port WHEN setHostPort is called THEN it should be updated successfully

containsByWsSessionId

GIVEN existing session ID WHEN containsByWsSessionId is called THEN it should return true

GIVEN non-existing session ID WHEN containsByWsSessionId is called THEN it should return false

getAllClientIds

GIVEN multiple clients WHEN getAllClientIds is called THEN list of all client IDs should be returned

GIVEN no clients WHEN getAllClientIds is called THEN empty list should be returned

GameController

handleGameAction()

GIVEN game running WHEN action received THEN handle ok

GIVEN game not running WHEN action received THEN throw exception

onClientDisconnect()

GIVEN client not in game WHEN client disconnect THEN do nothing

GIVEN client in game WHEN client disconnect THEN remove from lobby and end game

endGame()

GIVEN game running WHEN end game THEN success

GIVEN game not running WHEN end game THEN throw exception

GIVEN lobby does not exist WHEN end game THEN throw exception

startGame()

GIVEN lobby ready WHEN start game THEN success

GIVEN lobby not ready WHEN start game THEN throw exception

GIVEN lobby does not exist WHEN start game THEN throw exception

LobbyController

onClientDisconnect()

GIVEN client in lobby WHEN client disconnects THEN client should be removed from the lobby

GIVEN client not in any lobby WHEN client disconnects THEN do nothing

createLobby()

GIVEN nothing WHEN createLobby is called THEN a new lobby ID should be returned

GIVEN nothing WHEN called repeatedly THEN unique lobby IDs should be generated

removeLobby()

GIVEN empty lobby WHEN removeLobby is called THEN the lobby is removed successfully

GIVEN lobby with players WHEN removeLobby is called THEN lobby is removed and clients should be notified

GIVEN non-existing lobby WHEN removeLobby is called THEN an exception should be thrown

GIVEN existing lobby WHEN removeLobby is called twice THEN an exception should be thrown on second attempt

joinLobby()

GIVEN valid client and lobby WHEN joinLobby is called THEN client should be added successfully

GIVEN full lobby WHEN joinLobby is called THEN an exception should be thrown

GIVEN client already in another lobby WHEN joinLobby is called THEN an exception should be thrown

GIVEN non-existing lobby WHEN joinLobby is called THEN an exception should be thrown

leaveLobby()

GIVEN client in full lobby WHEN leaveLobby is called THEN client should be removed from lobby

GIVEN client doesn't exist WHEN leaveLobby is called THEN an exception should be thrown

GIVEN last client in lobby WHEN leaveLobby is called THEN client is removed form the lobby and lobby should be removed automatically

GIVEN client not in lobby WHEN leaveLobby is called THEN an exception should be thrown

toggleReady()

GIVEN client in lobby WHEN toggleReady is called THEN the ready status should be toggled

GIVEN client not in lobby WHEN toggleReady is called THEN an exception should be thrown

GIVEN client in lobby WHEN toggleReady is called multiple times THEN ready status should toggle each time

getLobby()

GIVEN valid lobby ID WHEN getLobby is called THEN correct lobby info should be returned

GIVEN invalid lobby ID WHEN getLobby is called THEN an exception should be thrown

GIVEN full lobby WHEN getLobby is called THEN correct number of players should be returned

getLobbiesInfo()

GIVEN existing lobbies WHEN getLobbiesInfo is called THEN all lobby info should be returned

GIVEN no lobbies WHEN getLobbiesInfo is called THEN empty list should be returned

configureLobby()

GIVEN valid lobby ID and game type WHEN configureLobby is called THEN lobby should be updated

GIVEN non-existing lobby WHEN configureLobby is called THEN an exception should be thrown

GIVEN lobby is playing WHEN configureLobby THEN throw exception

SessionController

createSession()

GIVEN valid lobby ID and parameters WHEN createSession is called THEN session should be created successfully

GIVEN non-existing lobby ID WHEN createSession is called THEN an exception should be thrown

GIVEN existing lobby with sessions WHEN createSession is called THEN new session should be added to lobby

GIVEN invalid duration WHEN createSession is called THEN an exception should be thrown

GIVEN invalid syncWindowLength WHEN createSession is called THEN an exception should be thrown

GIVEN invalid syncTolerance WHEN createSession is called THEN an exception should be thrown

removeSession()

GIVEN lobby with sessions WHEN removeSession is called THEN session should be removed successfully

GIVEN non-existing lobby ID WHEN removeSession is called THEN an exception should be thrown

GIVEN non-existing session ID WHEN removeSession is called THEN an exception should be thrown

getSessions()

GIVEN existing lobby with sessions WHEN getSessions is called THEN all sessions should be returned

GIVEN lobby with no sessions WHEN getSessions is called THEN empty list should be returned

GIVEN non-existing lobby ID WHEN getSessions is called THEN exception should be thrown

changeSessionsOrder()

GIVEN valid lobby and session order WHEN changeSessionsOrder is called THEN sessions should be reordered successfully

GIVEN non-existing lobby ID WHEN changeSessionsOrder is called THEN an exception should be thrown

GIVEN different number of session IDs WHEN changeSessionsOrder is called THEN an exception should be thrown

GIVEN session IDs not in lobby WHEN changeSessionsOrder is called THEN an exception should be thrown

GIVEN correct session order WHEN changeSessionsOrder is called THEN the order should be maintained

Watch.sensors.BaseListener

startTracker()

GIVEN a BaseListener with a handler, health tracker, tracker event listener set, and the handler is not running, WHEN startTracker() is called, THEN the health tracker should set the event listener.

GIVEN a BaseListener with a handler, health tracker, tracker event listener set, and the handler is already running, WHEN startTracker() is called, THEN the health tracker should not set the event listener.

GIVEN a BaseListener with a handler, tracker event listener set, the handler is not running, and the health tracker is set to null, WHEN startTracker() is called, THEN a NullPointerException should be thrown.

stopTracker()

GIVEN a BaseListener with a handler, health tracker, tracker event listener set, and the handler is running, WHEN stopTracker() is called, THEN the health tracker should unset the event listener, and the handler should remove callbacks and messages.

GIVEN a BaseListener with a handler, health tracker, tracker event listener set, and the handler is running, WHEN stopTracker() is called, THEN a NullPointerException should be thrown.

Watch.sensors.HeartRateListener

readValuesFromDataPoint()

GIVEN a valid and exist DataPoint WHEN readValuesFromDataPoint() is called with this data point, THEN the trackerDataObserver should receive heart rate data with matching status, heart rate, IBI, and IBI status values.

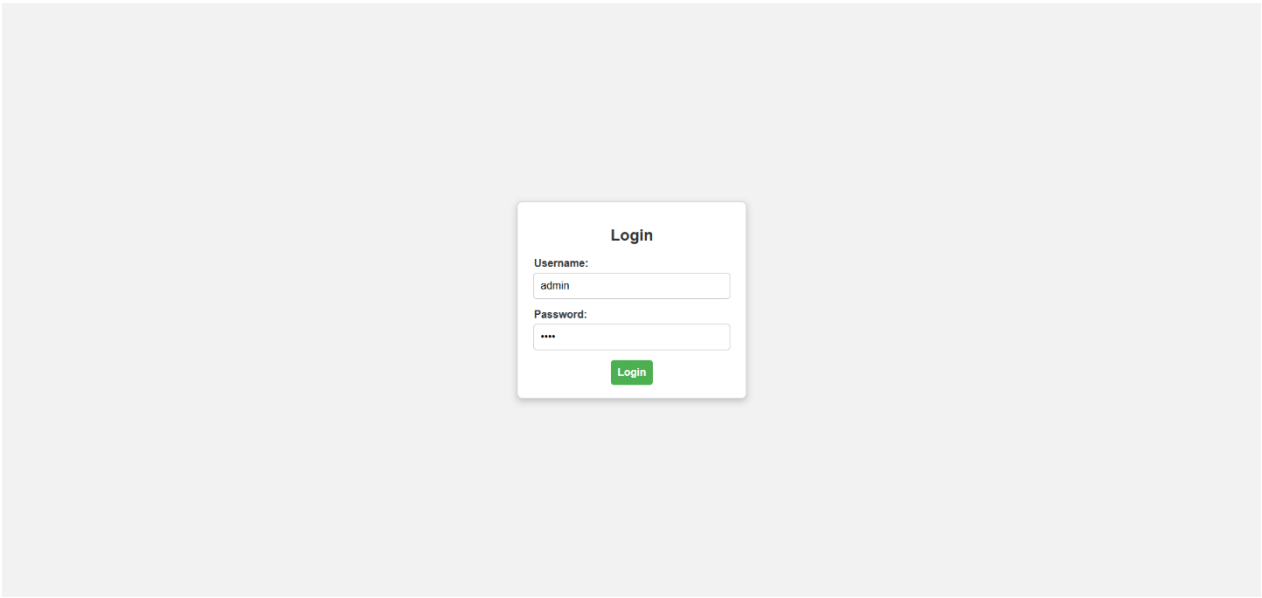
GIVEN a null DataPoint WHEN readValuesFromDataPoint() is called with this data point, THEN the trackerDataObserver should ignore the data recieved.

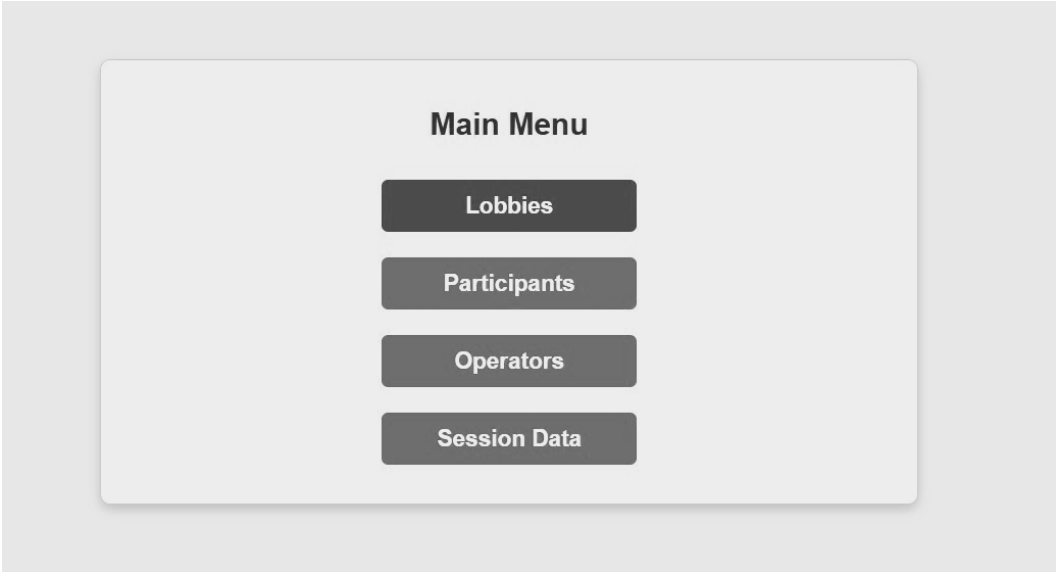
DataBase Unit Testing

The database will be tested with unit tests using an in-memory database. Every table will be tested for CRUD functionality and transactions.

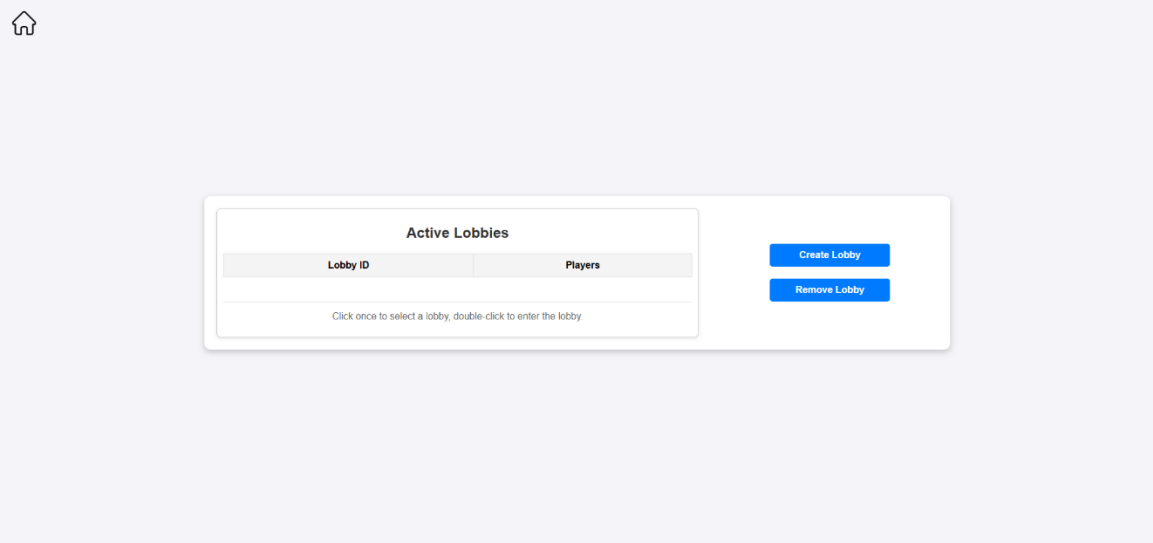
Chapter 6

User Interface Draft

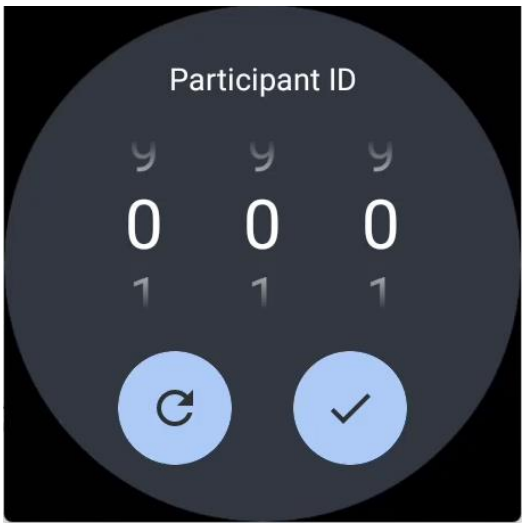


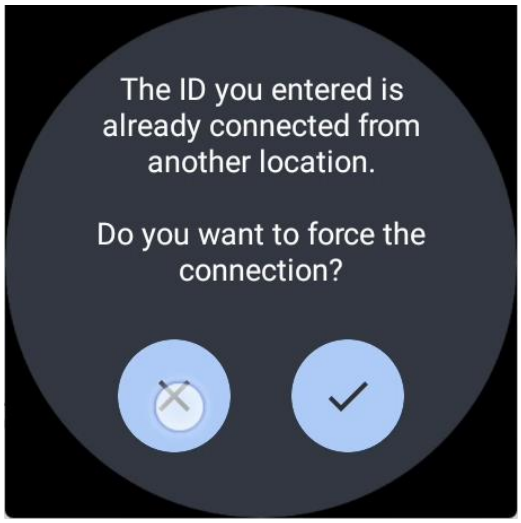


Lobbies main screen:

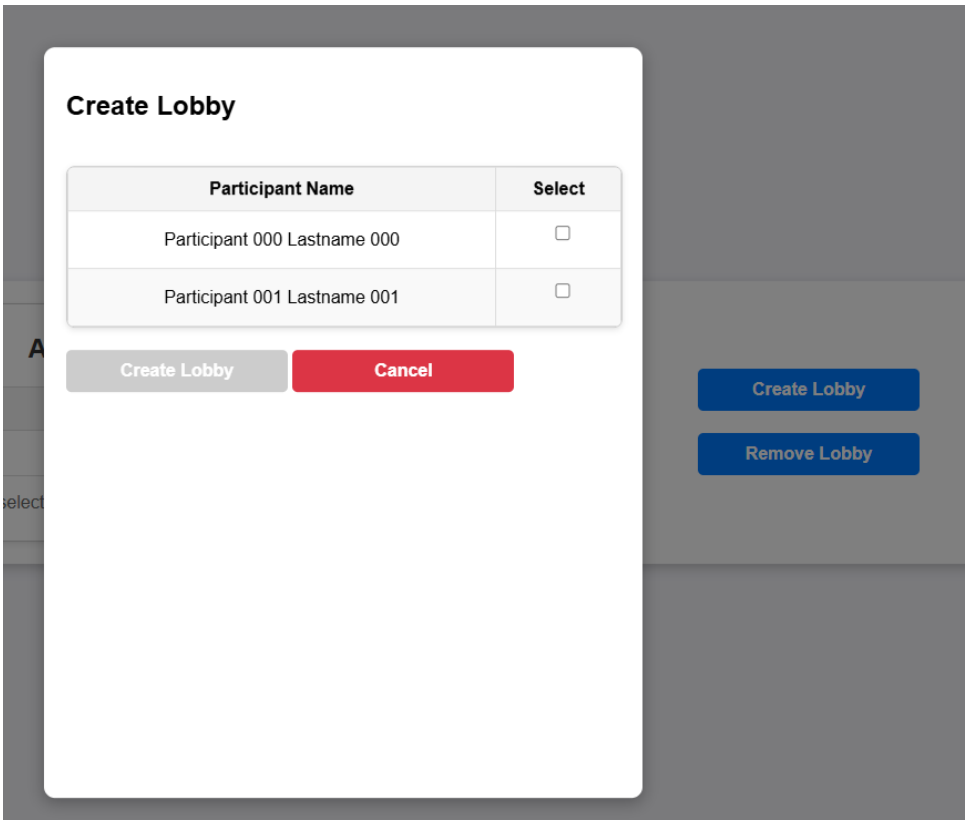


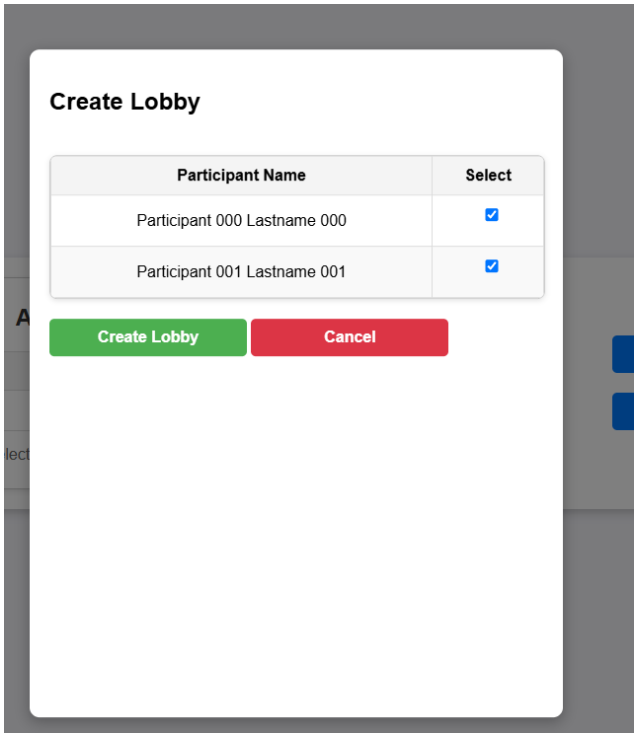
When open the app in the watch, first you need to pick your ID

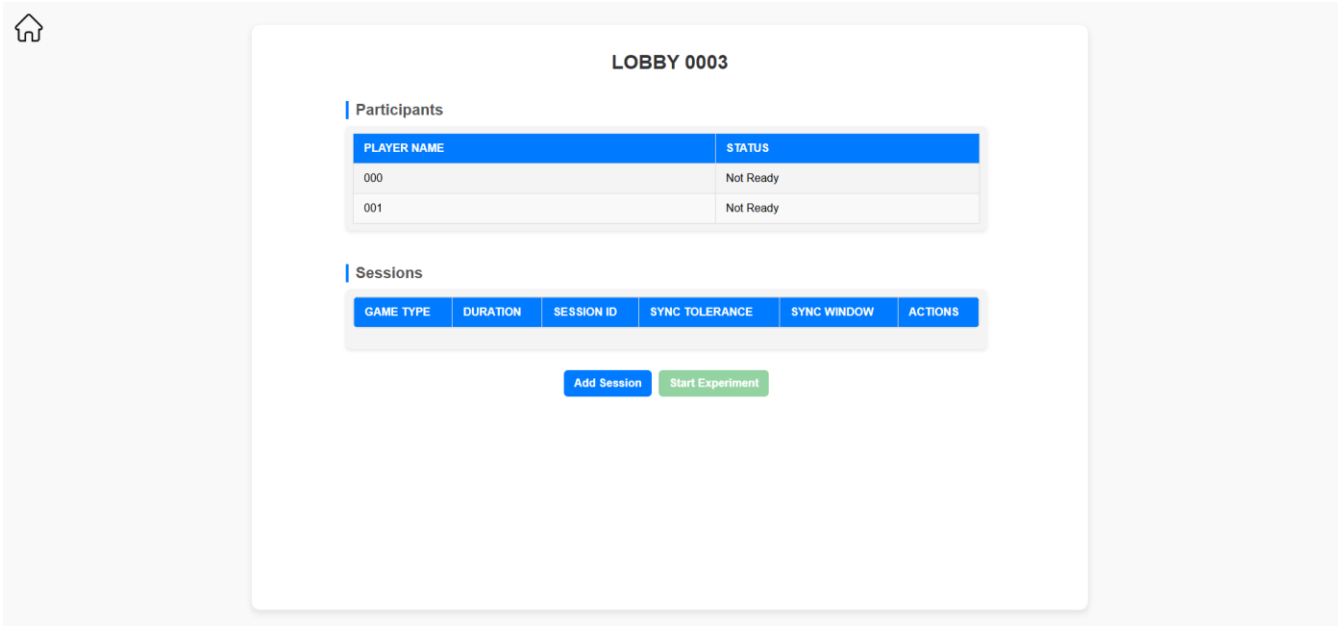




Creation of a lobby, You need to pick the participants ID's you want to join

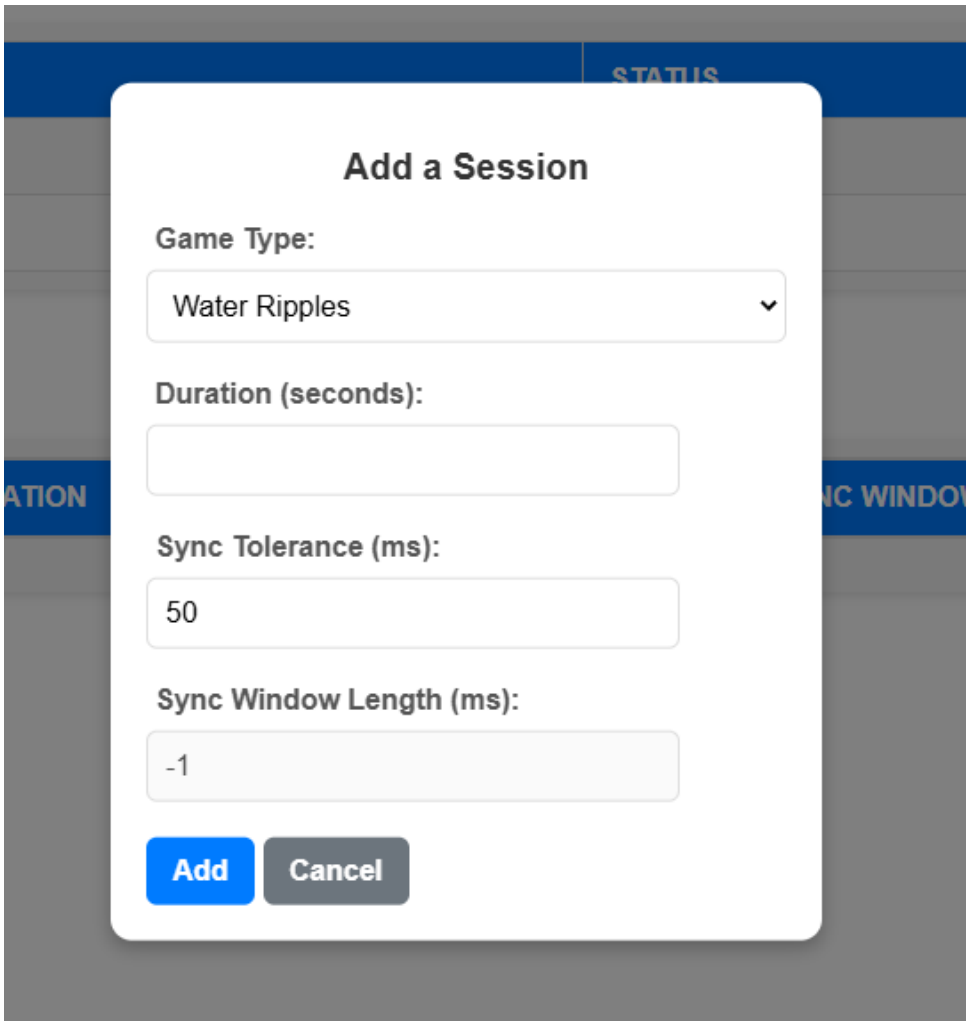




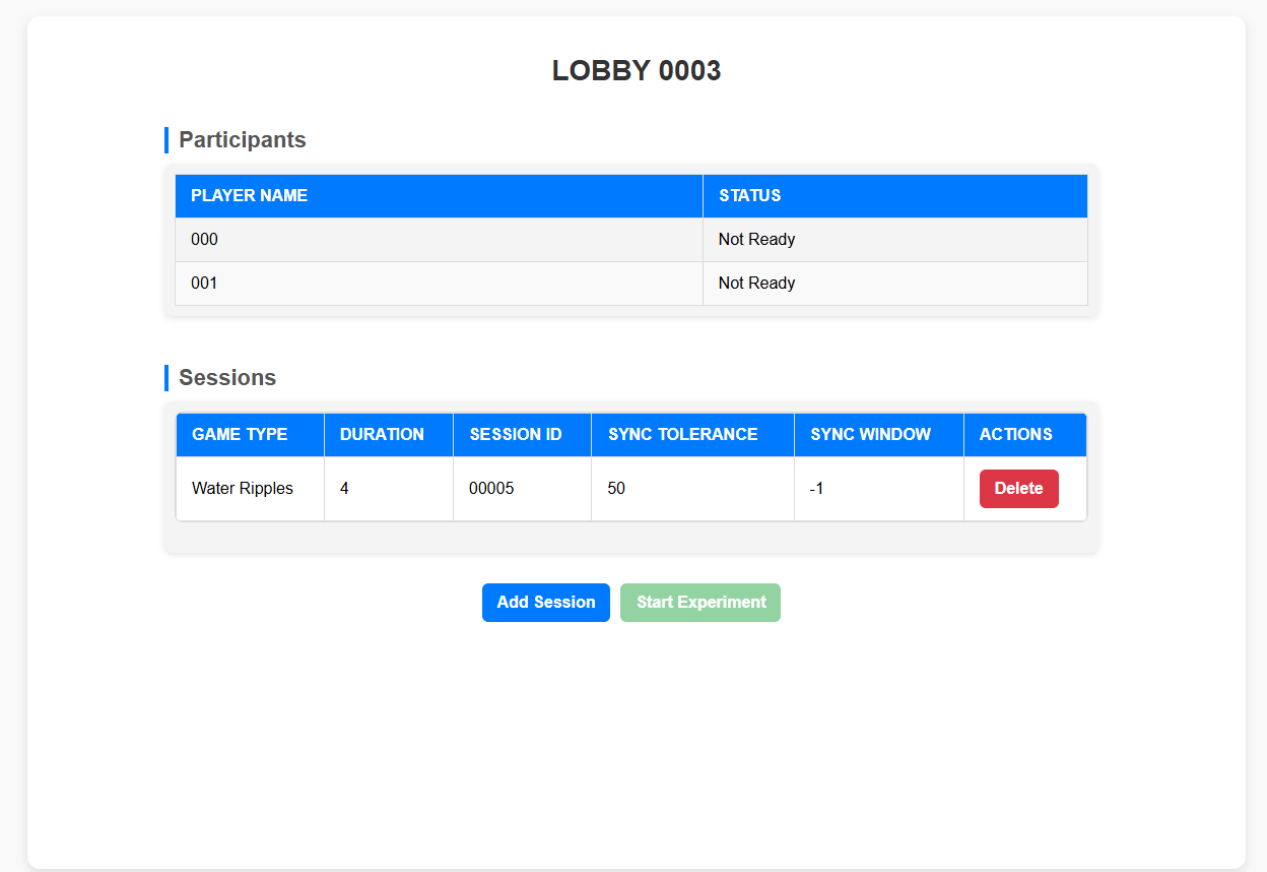


Lobby has been created

Adding a session :



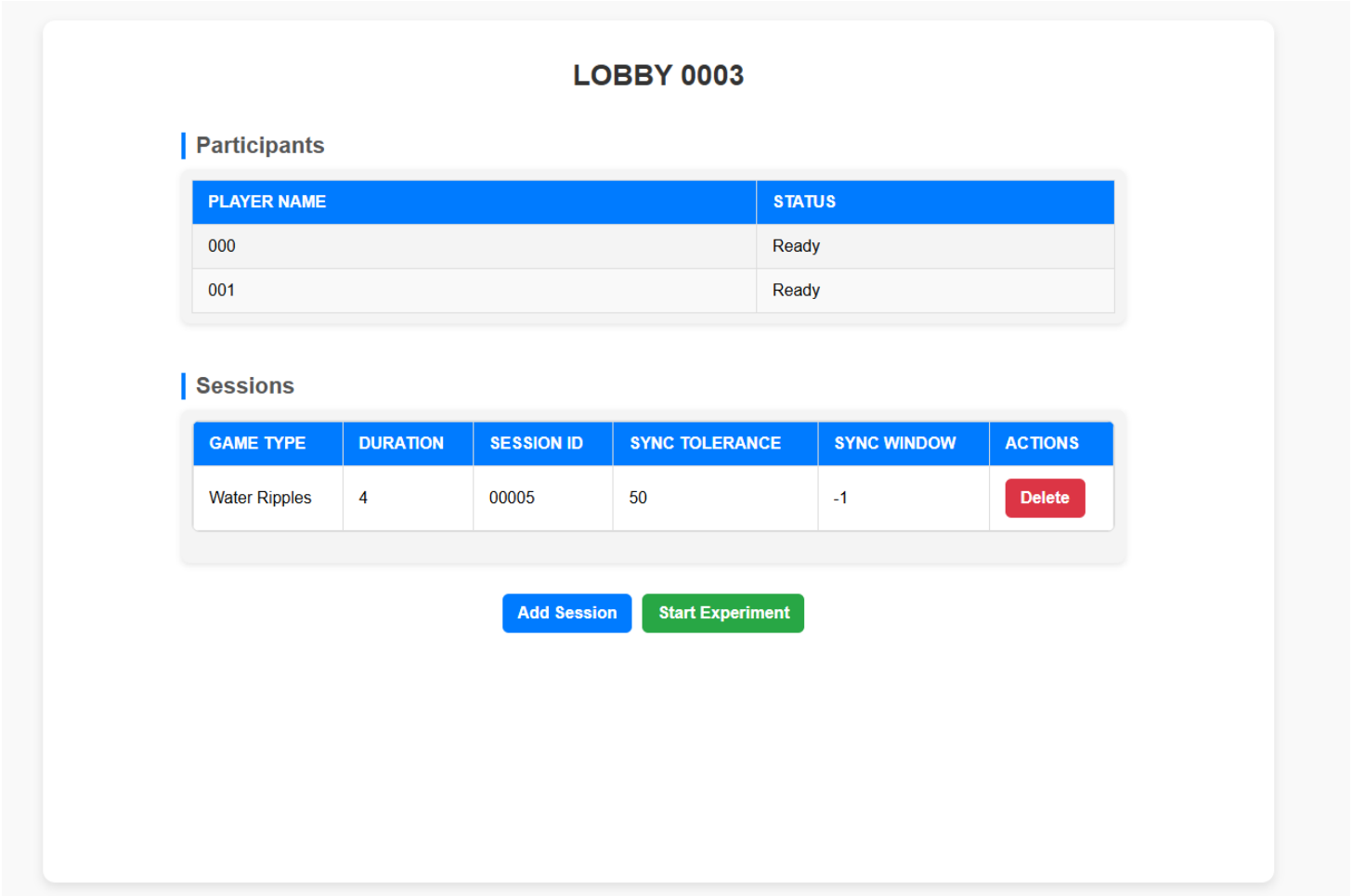
Session has been added to the list



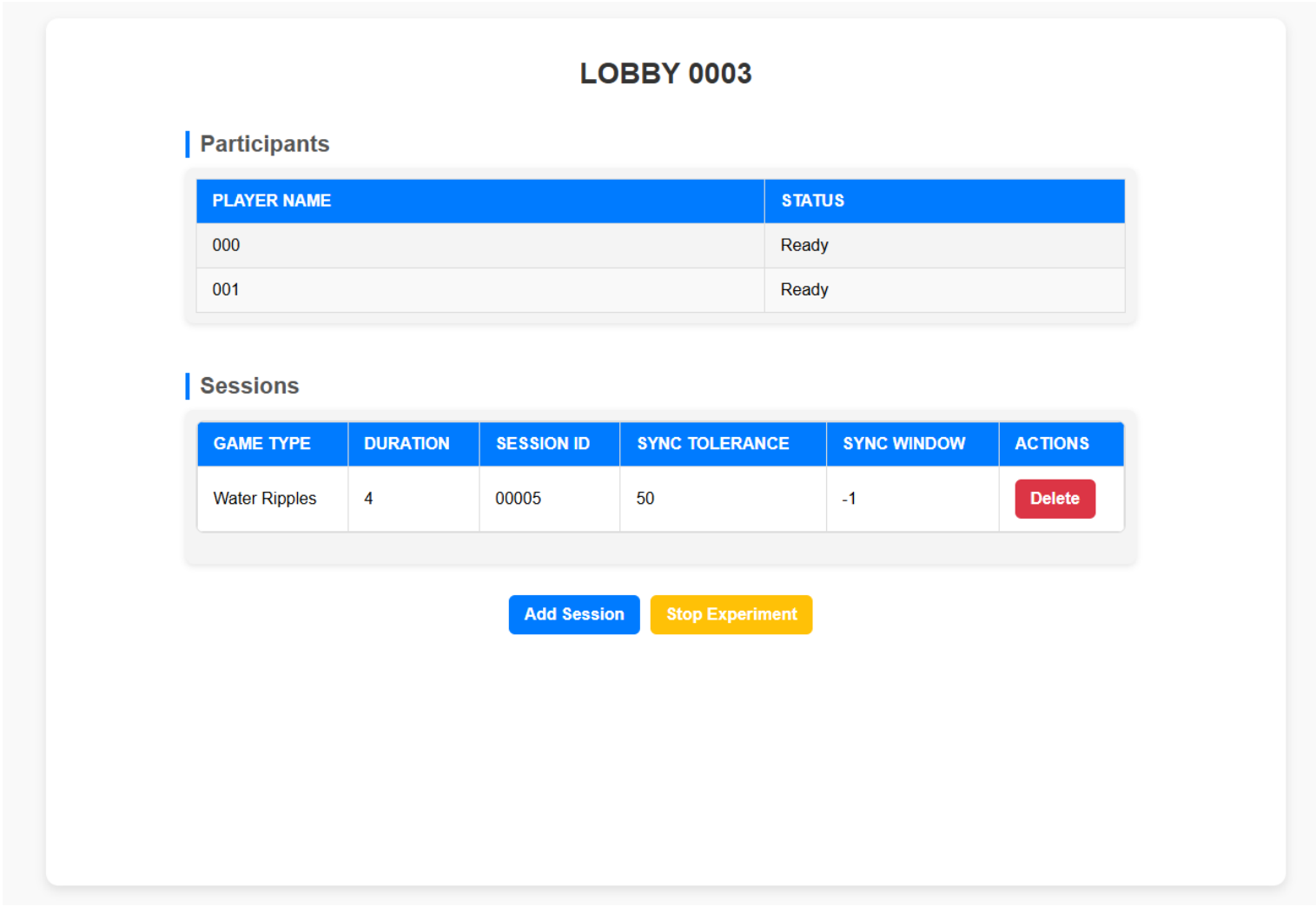
The watch has been joined to a lobby :

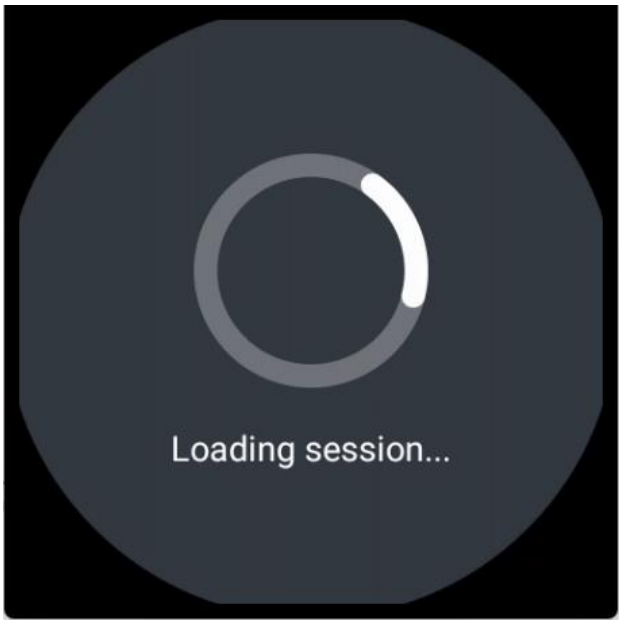
Pressed ready :

Ready statuses changed in the list

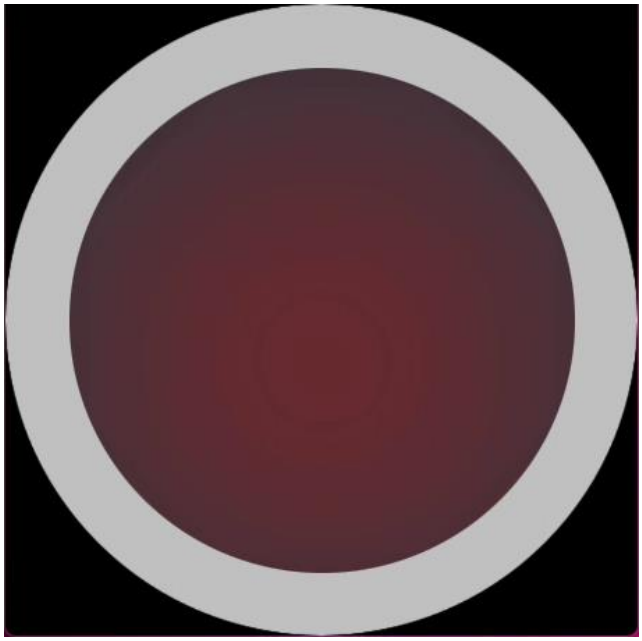


Start experiment:

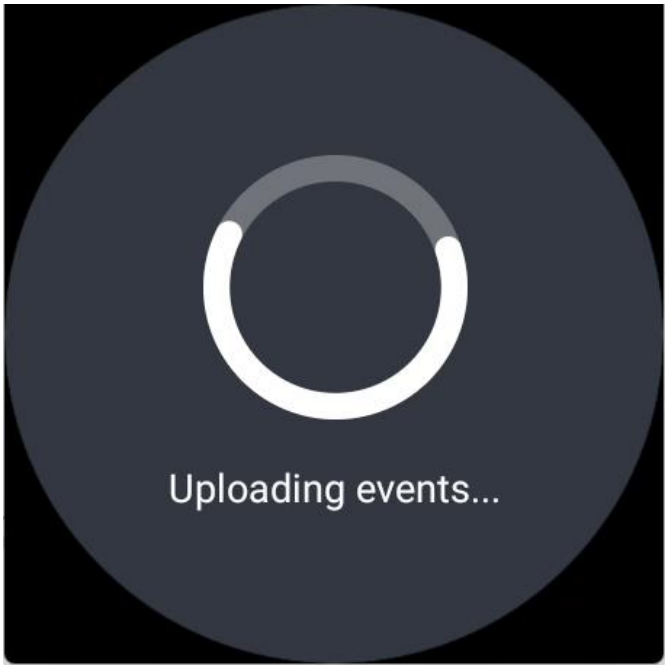




Based on the experiment type:

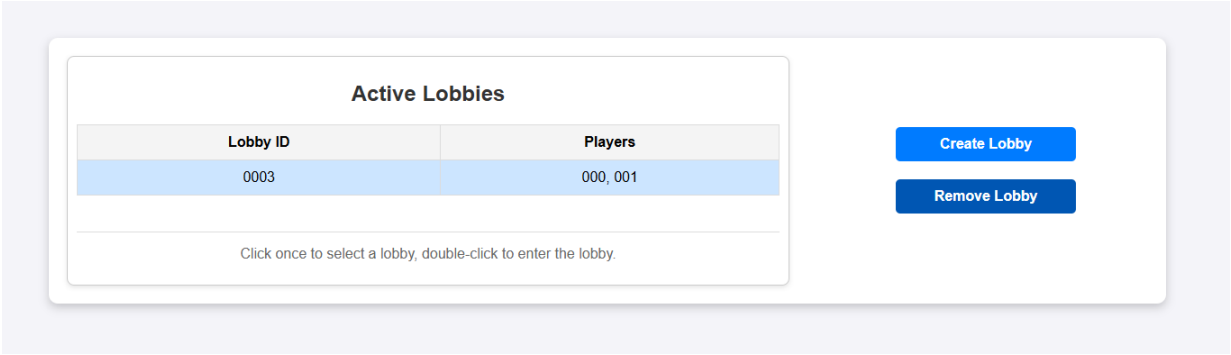


After a session end:

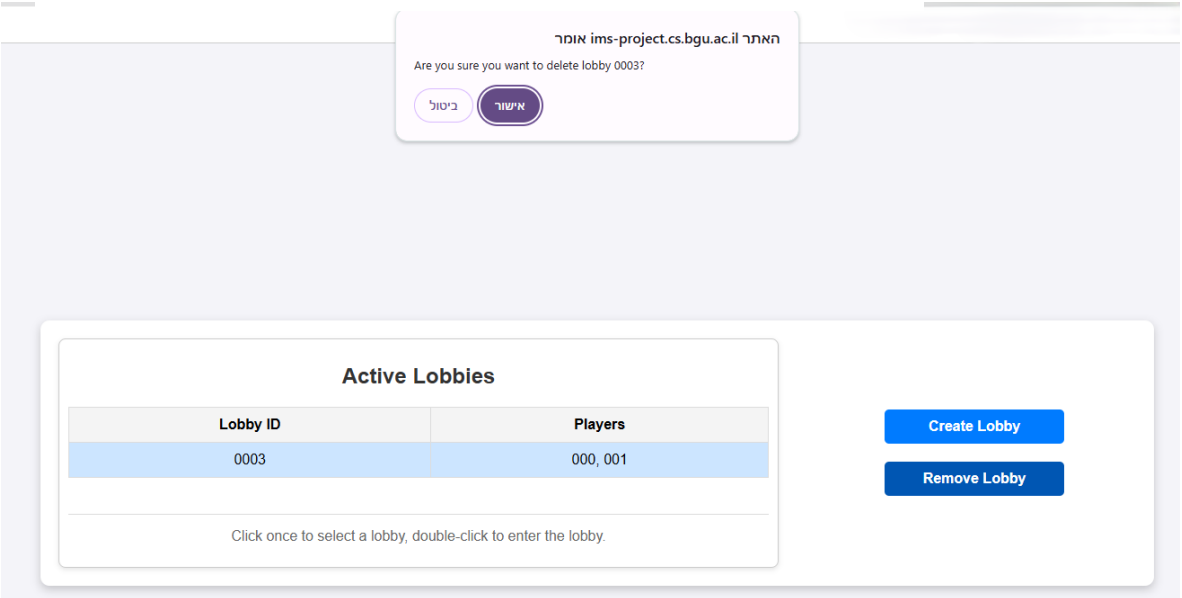


Lobbies main page :

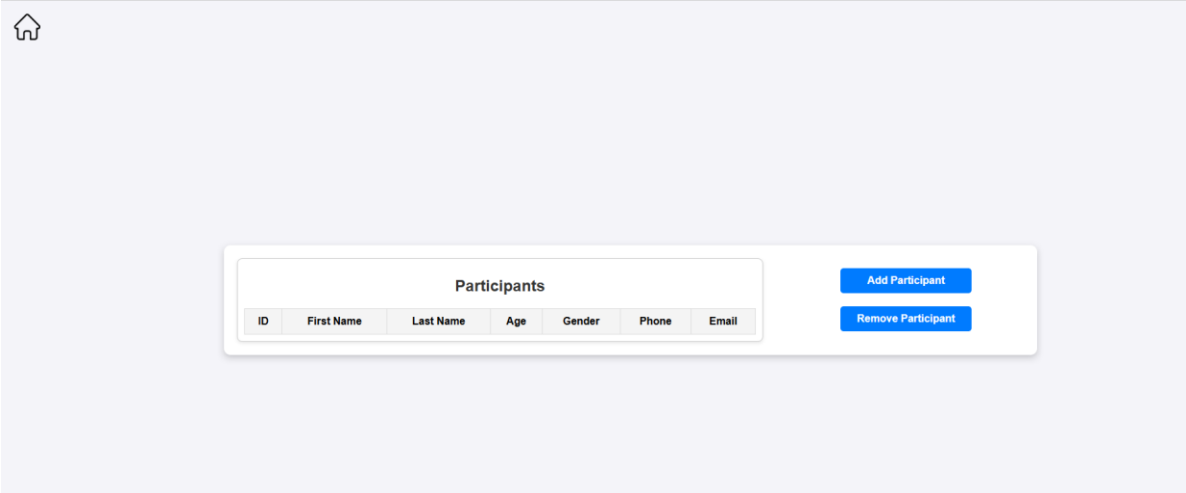
Choosing from the list



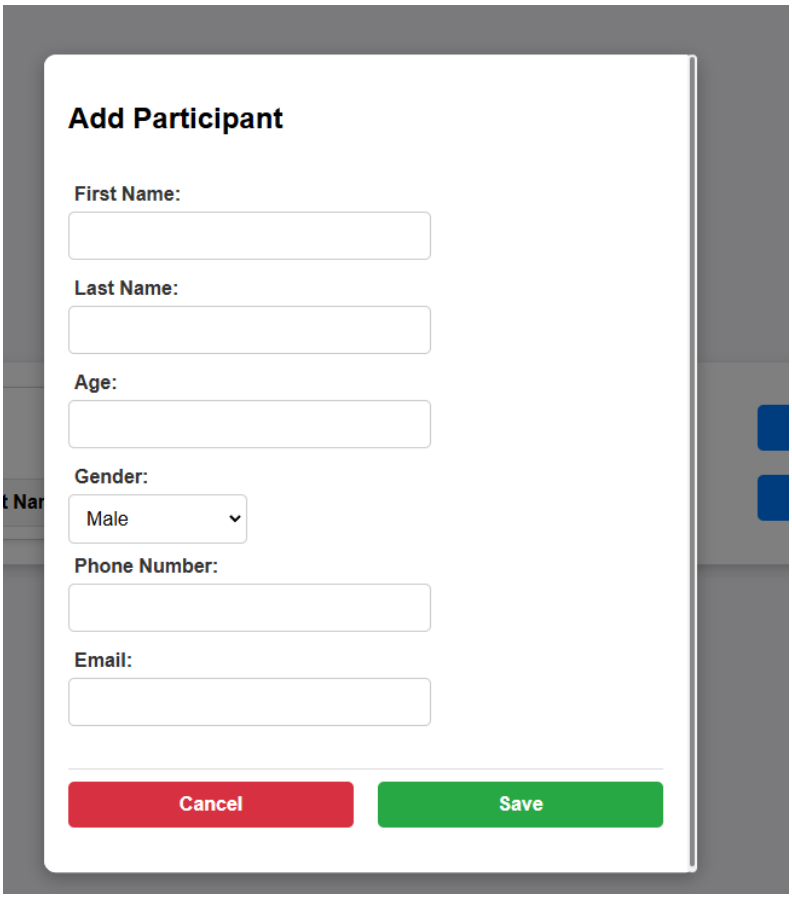
Removing a lobby :

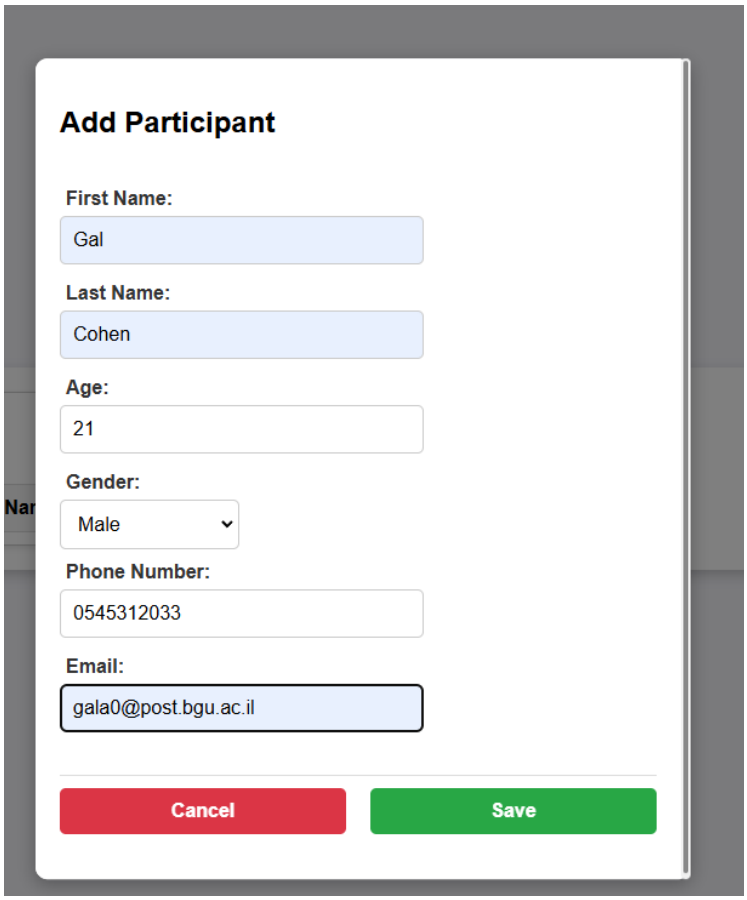


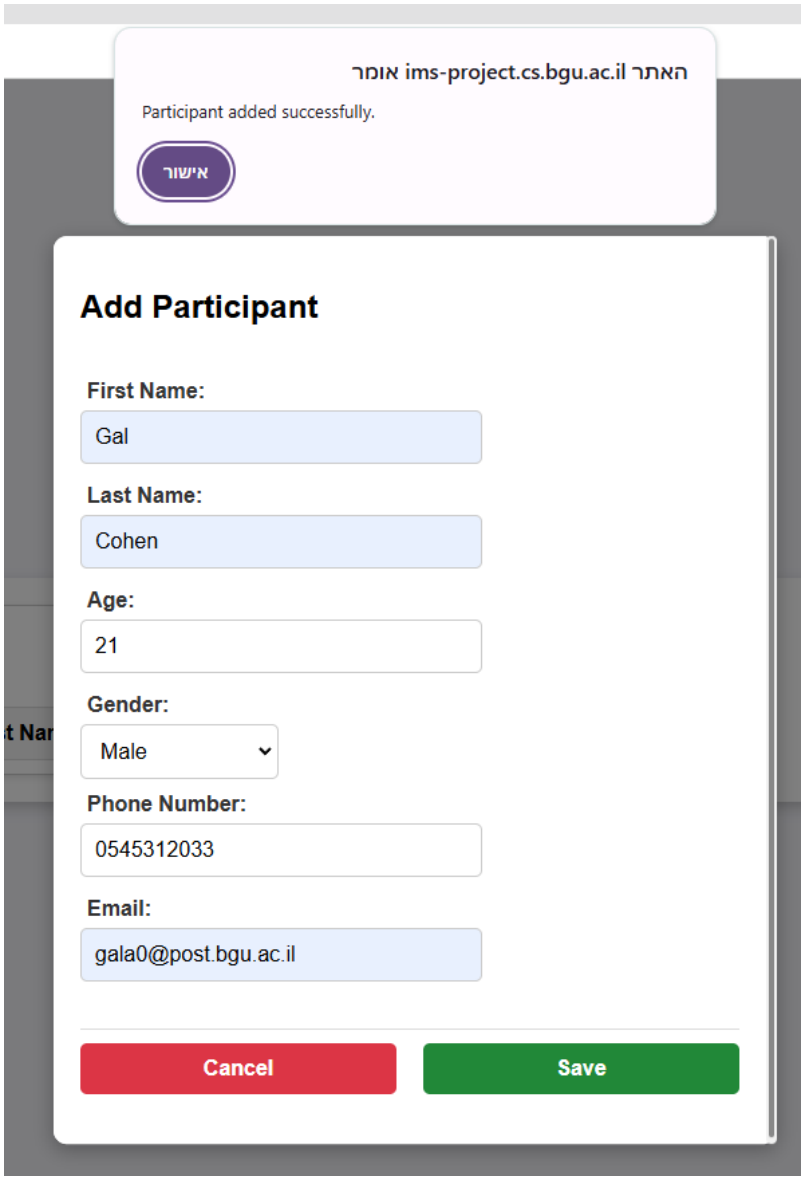
Participants main page :

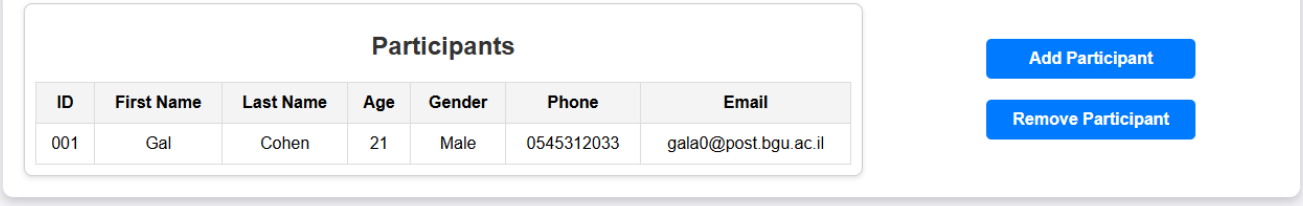


Adding a participant :

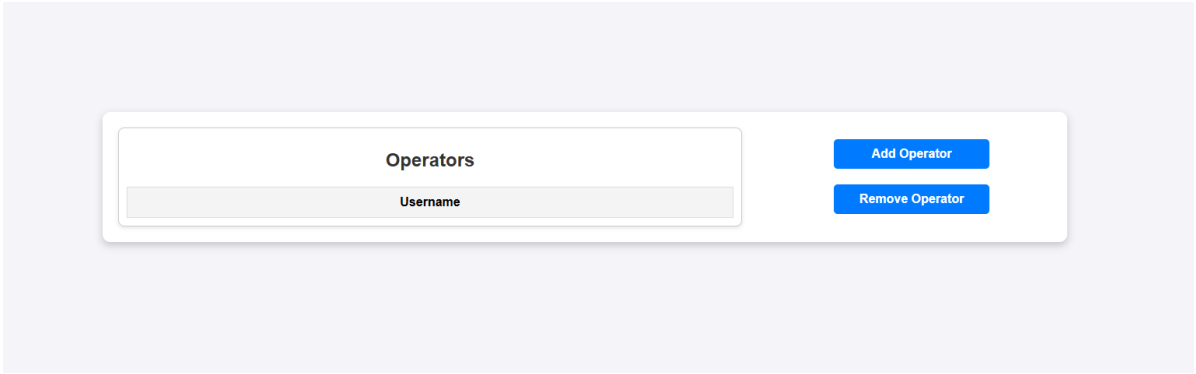


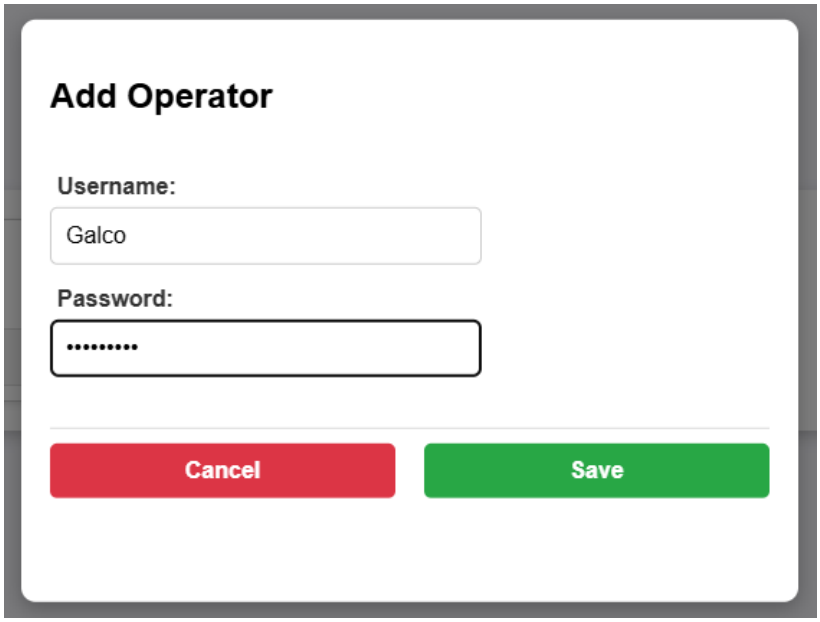


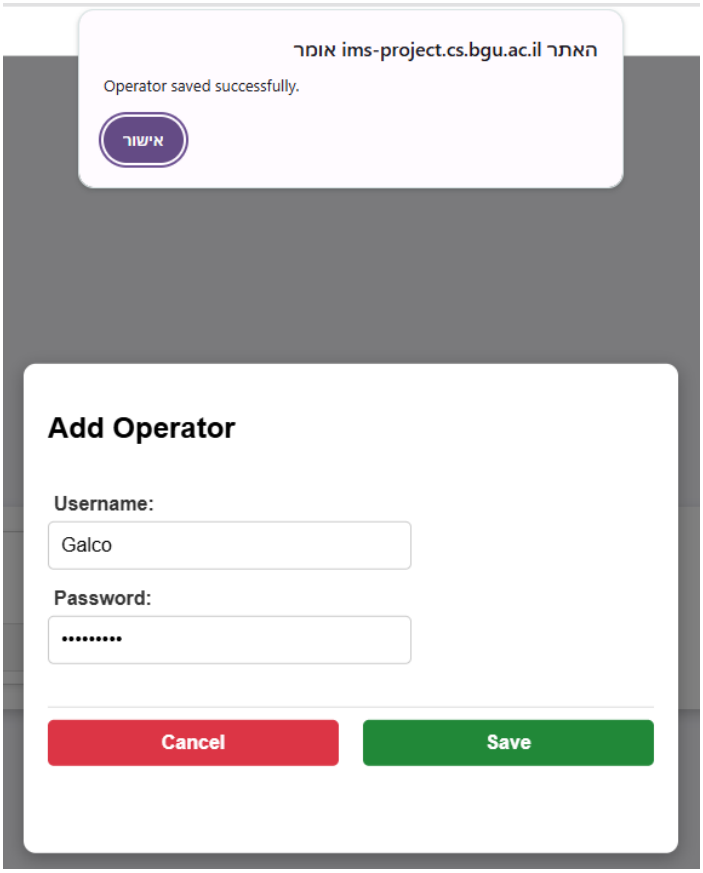

האתר ims-project.cs.bgu.ac.il אומר
Participant added successfully.
אישור
Add Participant
First Name:
Gal
Last Name:
Cohen
Age:
21
Gender:
Male
Phone Number:
0545312033
Email:
gala0@post.bgu.ac.il
Cancel
Save

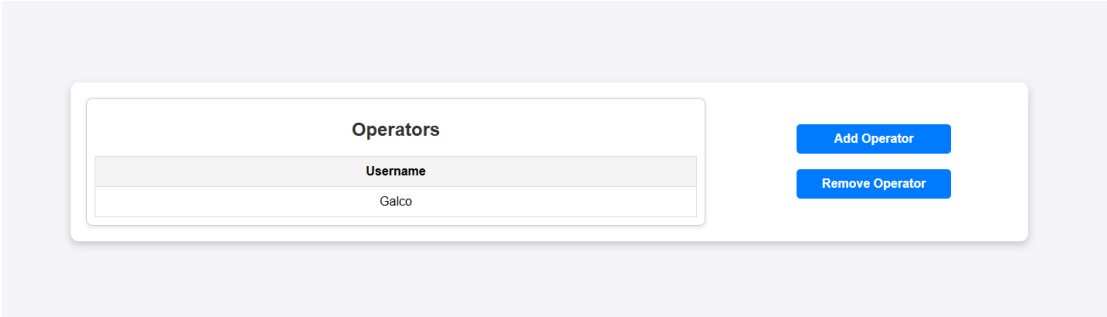

Participants
ID
First Name
Last Name
Age
Gender
Phone
Email
001
Gal
Cohen
21
Male
0545312033
gala0@post.bgu.ac.il
Add Participant
Remove Participant

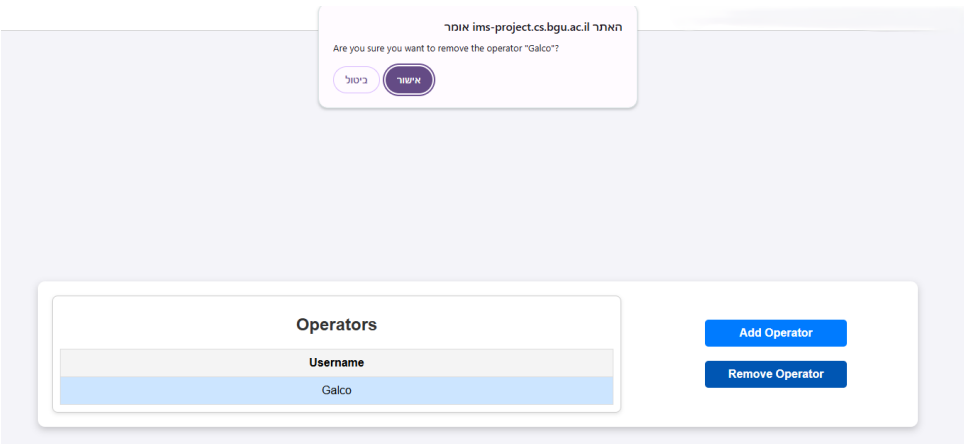
Operators main page :

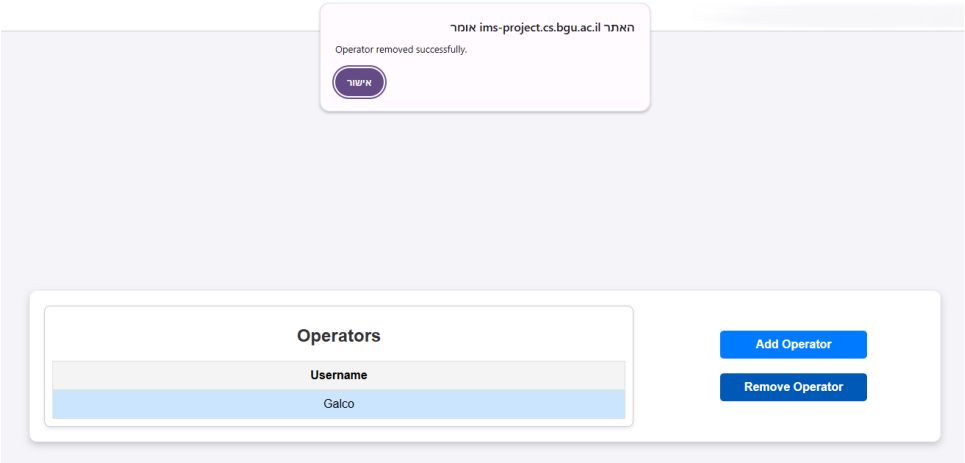




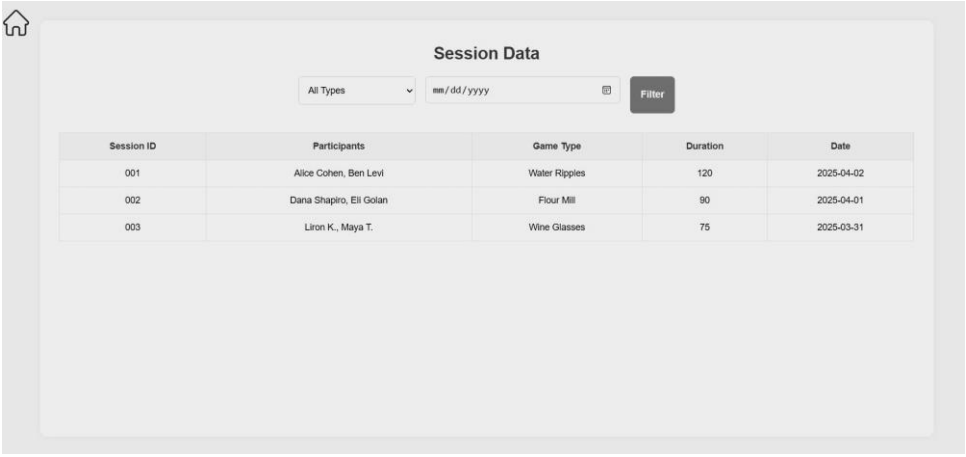




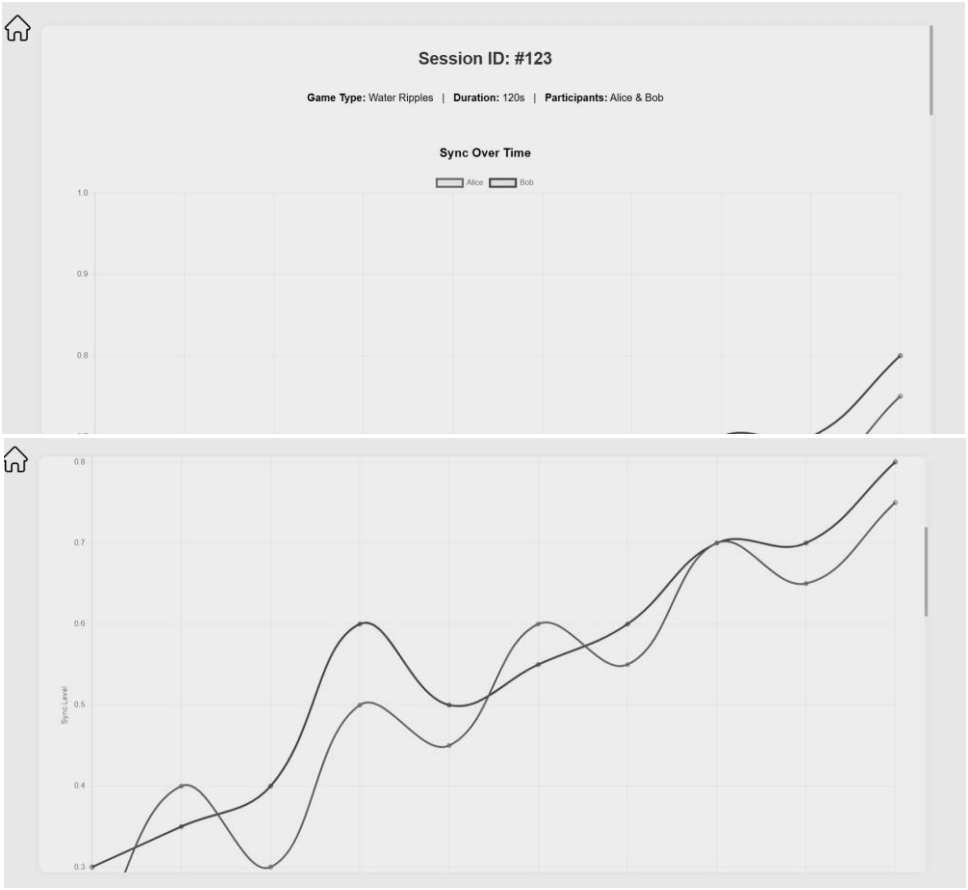




Session Data Page (initial) :



Session Data Graphs Page (initial) :



Chapter 7

Testing

Smartwatch App Test Cases

1. MainActivity UI & Flow

Test Case 1.1: Participant ID Entry Validation
Objective: Verify the ID picker UI and input logic.
Validation: Manually pick an ID through the 3-digit picker and confirm a valid ID. Ensure the UI updates correctly and transitions to the next screen.

Test Case 1.2: Loading Screen
Objective: Validate loading animation during network operations.
Validation: Manually trigger a network operation (e.g connecting with an id when launching the app) and observe the loading screen. Ensure the circular progress indicator spins smoothly and the "Connecting..." text is displayed.

Test Case 1.3: Lobby Status Screen
Objective: Ensure UI updates when assigned to a lobby.
Validation: Manually join a lobby and verify that the "Connected in Lobby" screen displays the correct user ID, lobby ID, game type, duration, and "Ready/Not Ready" button with a pulsing border.

---

2. WaterRipplesActivity UI & Interaction

Test Case 2.1: Tap Feedback
Objective: Validate ripple animation and sound on tap.
Validation: Manually tap the center button multiple times and observe the ripple animation and sound. Ensure each tap triggers a ripple and sound effect without delay.

Test Case 2.2: Performance Under High Load
Objective: Test UI responsiveness during rapid input.
Validation: Manually spam taps for 10 seconds and observe the app's performance. Ensure no crashes occur and the UI remains responsive (no visible lag).

---

3. WineGlassesActivity UI & Interaction

Test Case 3.1: Swipe Gesture Recognition
Objective: Validate circular swiping detection.

Validation: Manually swipe clockwise and counterclockwise on the screen. Verify that the yellow arc follows the finger position and the opponent's cyan arc updates in real-time.

Test Case 3.2: Haptic Feedback on Synchronization
Objective: Test haptic response during sync.
Validation: Manually swipe in sync with a paired user and observe the haptic feedback. Ensure strong vibration occurs when arcs overlap (sync) and mild vibration when out of sync.

---

4. Error Handling

Test Case 4.1: Network Failure Recovery
Objective: Test error UI and retry logic.
Validation: Manually disable Wi-Fi during a lobby connection and observe the error screen. Verify the error message is displayed and clicking "Dismiss" returns to the ID entry screen.

---

5. Post-Session UI

Test Case 5.1: Rating Prompt
Objective: Ensure post-session rating appears.
Validation: Manually end a session via the admin console and verify that the "Rate this session (1–7)" prompt appears immediately on the smartwatch.

Non-Functional Constraints Test Cases

1. Performance (Latency < 20ms)

Test Case 1.1: Latency Measurement

- **Objective:** Verify that data transfer latency between users is less than 20ms.
- **Validation:** Start a session between two users and perform a tap or swipe on one device. Measure the time until the action is reflected on the other device using a stopwatch or logging timestamps. Repeat the process multiple times to ensure consistency. The latency should consistently be less than 20ms.

---

2. Reliability & Stability (Session Continuity)

Test Case 2.1: Session Continuity When One User Disconnects

- **Objective:** Verify that the session stops if the other user is unavailable.
- **Validation:** Start a session between two devices and disconnect one device (turn off Wi-Fi). Observe the behavior of the app on the other device. The session should stop, and an error message should be displayed on the remaining device.

---

3. Usability (Screen Activity)

Test Case 3.1: Screen Activity During Session

- **Objective:** Verify that the screen remains active during the session.
- **Validation:** Start a session on a watch and wait for the screen timeout period. Observe if the screen remains active during the session. The screen should not turn off while the app is running.

---

5. User Polling (Post-Session Feedback)

Test Case 5.1: Post-Session Feedback Form

- **Objective:** Verify that users are prompted to fill out a feedback form after the session.
- **Validation:** Complete a session (e.g., Water Ripples or Wine Glasses) and observe if a feedback form appears after the session ends. Fill out the form and verify that the data is logged correctly. A feedback form should appear after the session, and the data should be logged correctly.

Database Testing

- The database will also be tested manually to ensure backup and restoration.
- Integration tests will be performed using a temporary PostgreSQL instance to verify real database interactions.

Manager (Web Interface) Testing

Manual Testing Approach: We will manually test all features, exploring every combination to ensure functionality, responsiveness, and stability.

Test Scenarios:

- Navigation & UI Flow – Verify all pages load correctly, buttons work, and navigation is smooth.
- Feature Testing – Test login, lobby management, game controls, and participant handling.
- Edge Cases & Stress Testing – Input invalid data, rapid interactions, and attempt restricted actions.
- Crash Testing – Try to break the system through excessive inputs and simultaneous actions.

Expected Outcome: The system should remain stable, all functionalities should work as intended, and errors should be properly handled without crashes.

Manager (Web Interface) Testing

- Testing the sensor readings after multiple watch exchanges between users.
- Testing the sensor readings in relation to a real measuring device, if we can get one.
- Testing the sensors during accelerated movement, or unusual movements.
- Using it during an experiment and testing the effect of actions on the watch on the sensor readings.