Learning Curve

Ash Duy, Alex Herman, Annikka Turmala

Implemented features

- Our project features a home page where you can either create or join a room! Once in a room you will be able to see a default card with a slider on it. The host of the room can change the question that relates to the slider. The host cannot move the slider themself. The guests in the room can move their slider, and the host will see the average of everyone's slider.

- Anyone can add their own card by clicking the button on the top right of the screen. You can give your card a title, and a body. The adding card feature is still in a beta version, because you will not see cards that were added to the room prior to you joining it.

- We also have a nice svg that is drawn when you go to our website! (It's super nice)

Features not implemented

- We were going to implement 3 types of addable cards; questions, polls, and comments. However, we only got around to adding the comment card.

- We did not implement the database mainly due to time constraints. Furthermore, the database does not add a huge amount of functionality, and there were more pertinent features.

- We did not add the ability to delete cards. The infrastructure for this feature would be cumbersome, and we wanted to focus our time on more pertinent features.

Our project did not change in any major way from project 5 and 6. The core functionality of our application was implemented well. This includes room creation and joining ability, the teacher's question scale feature, as well as the ability for each user to create their own question card. The additional features like polls and comments cards were bells and whistles that we were not able to implement at this time. Hopefully they will be seen in the future of this application!

3. Final Class Diagram and Comparison Statement

Not much changed in our diagrams between what we submitted for Project 6 while the changes from Project 5 may be more stark. Many of the key changes that occurred were due to time limiting our scope and our discovery of Angular's many features that made parts of our diagram redundant. For instance, a card factory wasn't necessarily needed due to Angular's structural directives which made rendering them automated.

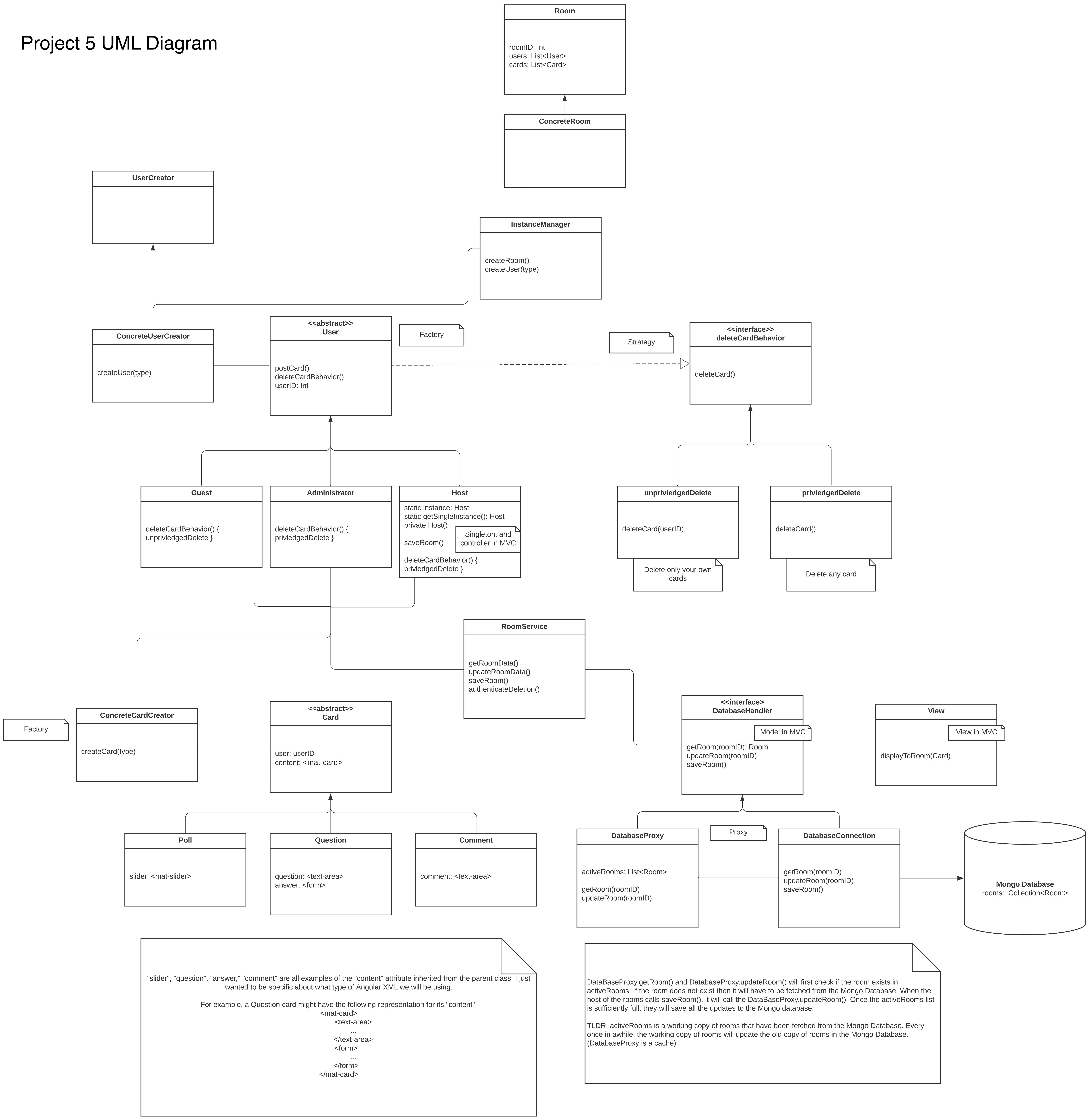
4. Third-Party Code Statement

To complete this project we relied heavily on the angular developer docs. We also used the rxjs docs, and the socket.io docs. For usage of StackOverflow or other similar websites, we commented links into the code.
https://angular.io/docs
https://rxjs.dev/api
https://socket.io/docs/v4/

5. Statement on the OOAD process for your overall Semester Project

The key OOAD design principles in our semester project were: diagraming before coding, building code modularly, and using design patterns. For diagramming, we created a UML diagram at each stage of the project, as well as an architecture diagram before starting on any actual code. For building code modularly, we focused on key takeaways from the class. For example, assuring that each of our objects and methods had a single purpose to the best of our ability.

Project 5 UML Diagram



"slider", "question", "answer," "comment" are all examples of the "content" attribute inherited from the parent class. I just wanted to be specific about what type of Angular XML we will be using.

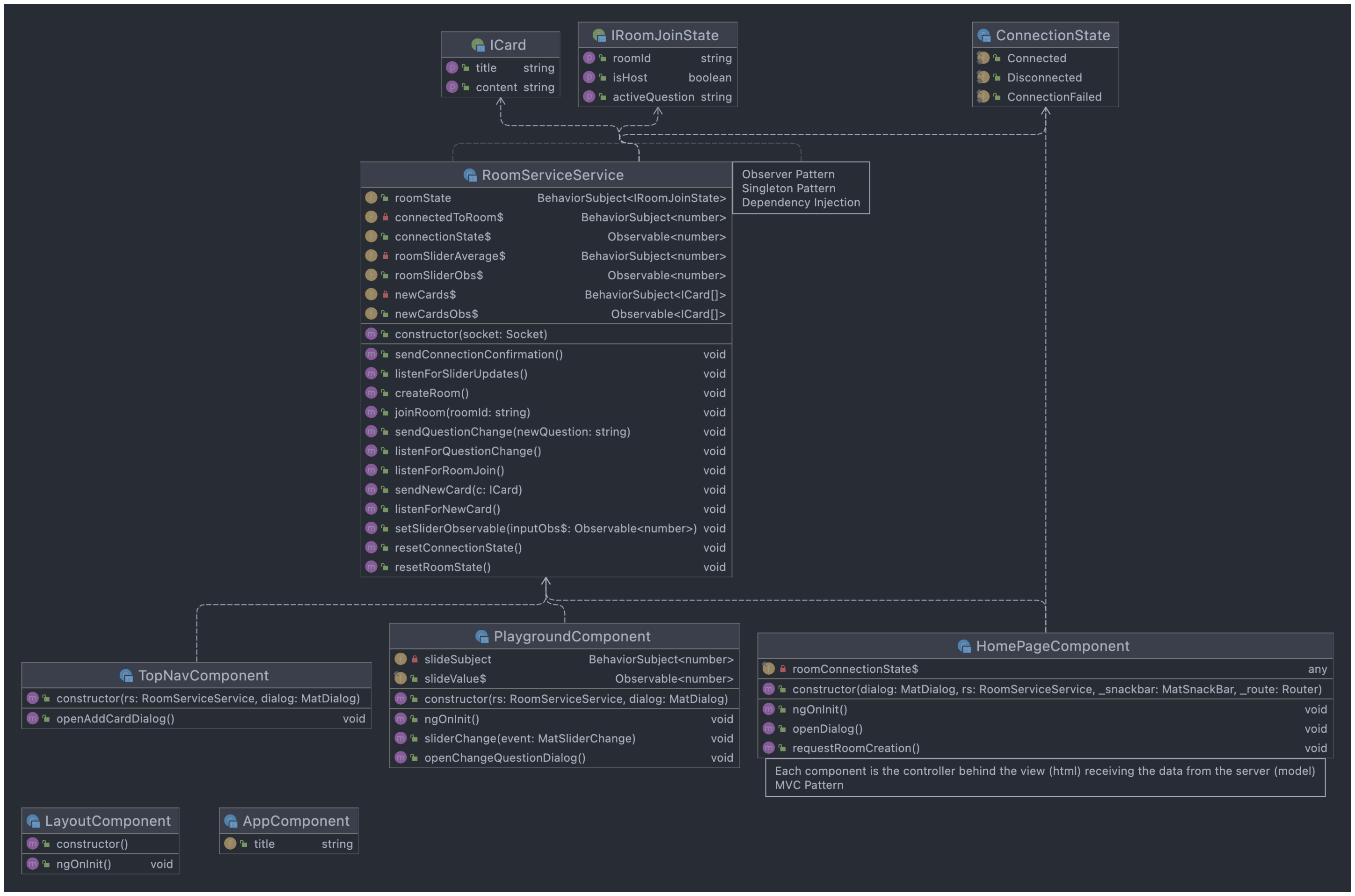
For example, a Question card might have the following representation for its "content":

```
<mat-card>
    <text-area>
        ...
    </text-area>
    <form>
        ...
    </form>
</mat-card>
```

DataBaseProxy.getRoom() and DatabaseProxy.updateRoom() will first check if the room exists in activeRooms. If the room does not exist then it will have to be fetched from the Mongo Database. When the host of the rooms calls saveRoom(), it will call the DataBaseProxy.updateRoom(). Once the activeRooms list is sufficiently full, they will save all the updates to the Mongo database.

TLDR: activeRooms is a working copy of rooms that have been fetched from the Mongo Database. Every once in awhile, the working copy of rooms will update the old copy of rooms in the Mongo Database. (DatabaseProxy is a cache)

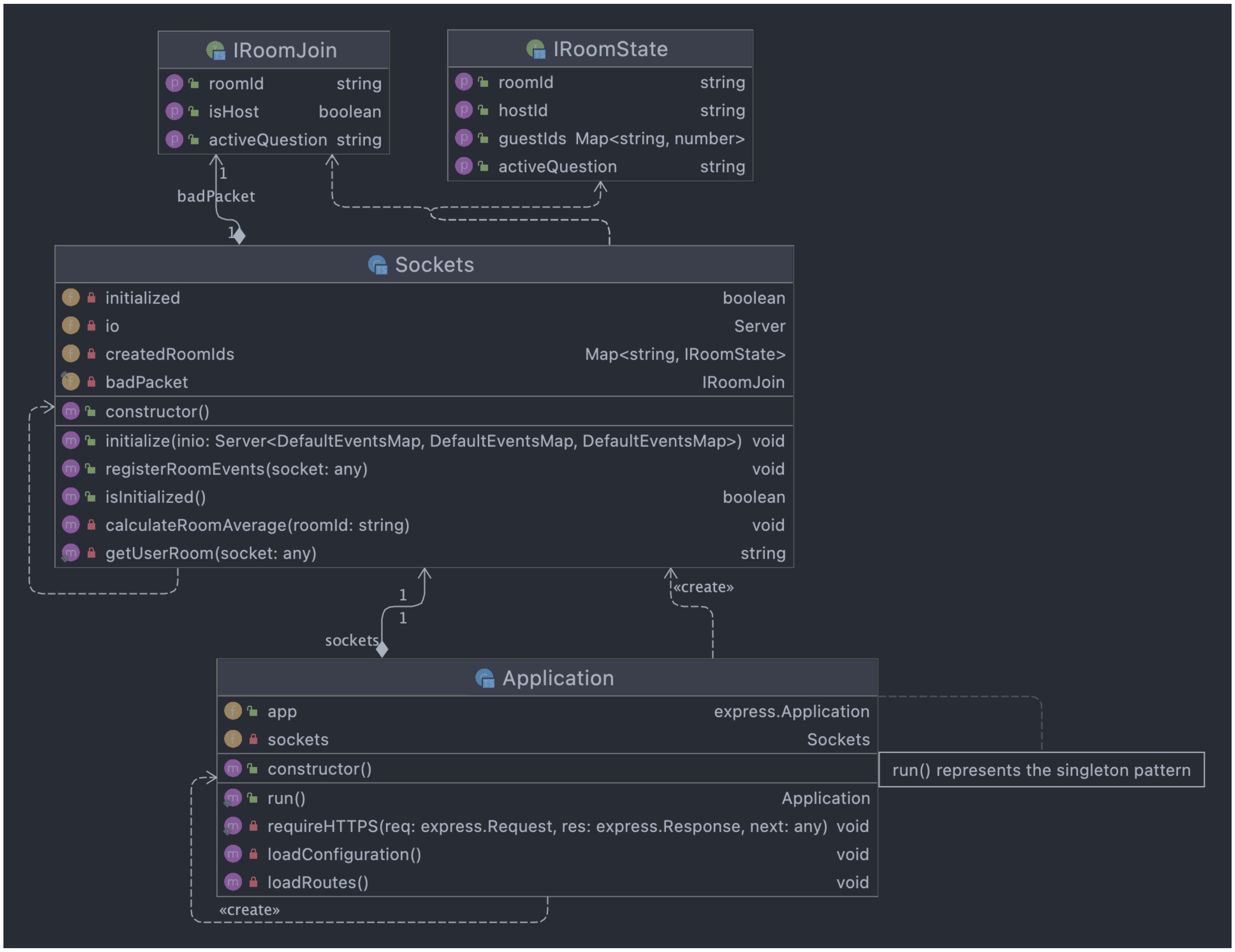
Final Client UML Diagram


ICard
title string
content string
IRoomJoinState
roomId string
isHost boolean
activeQuestion string
ConnectionState
Connected
Disconnected
ConnectionFailed
RoomServiceService
roomState BehaviorSubject<IRoomJoinState>
connectedToRoom$ BehaviorSubject<number>
connectionState$ Observable<number>
roomSliderAverage$ BehaviorSubject<number>
roomSliderObs$ Observable<number>
newCards$ BehaviorSubject<ICard[]>
newCardsObs$ Observable<ICard[]>
constructor(socket: Socket)
sendConnectionConfirmation() void
listenForSliderUpdates() void
createRoom() void
joinRoom(roomId: string) void
sendQuestionChange(newQuestion: string) void
listenForQuestionChange() void
listenForRoomJoin() void
sendNewCard(c: ICard) void
listenForNewCard() void
setSliderObservable(inputObs$: Observable<number>) void
resetConnectionState() void
resetRoomState() void
TopNavComponent
constructor(rs: RoomServiceService, dialog: MatDialog)
openAddCardDialog() void
PlaygroundComponent
slideSubject BehaviorSubject<number>
slideValue$ Observable<number>
constructor(rs: RoomServiceService, dialog: MatDialog)
ngOnInit() void
sliderChange(event: MatSliderChange) void
openChangeQuestionDialog() void
HomePageComponent
roomConnectionState$ any
constructor(dialog: MatDialog, rs: RoomServiceService, _snackbar: MatSnackBar, _route: Router)
ngOnInit() void
openDialog() void
requestRoomCreation() void
LayoutComponent
constructor()
ngOnInit() void
AppComponent
title string

Observer Pattern
Singleton Pattern
Dependency Injection

Each component is the controller behind the view (html) receiving the data from the server (model)
MVC Pattern

Final Server UML Diagram


IRoomJoin
roomId string
isHost boolean
activeQuestion string
IRoomState
roomId string
hostId string
guestIds Map<string, number>
activeQuestion string
1
badPacket
1
Sockets
initialized boolean
io Server
createdRoomIds Map<string, IRoomState>
badPacket IRoomJoin
constructor()
initialize(inio: Server<DefaultEventsMap, DefaultEventsMap, DefaultEventsMap>) void
registerRoomEvents(socket: any) void
isInitialized() boolean
calculateRoomAverage(roomId: string) void
getUserRoom(socket: any) string
1
1
sockets
«create»
Application
app express.Application
sockets Sockets
constructor()
run() Application
requireHTTPS(req: express.Request, res: express.Response, next: any) void
loadConfiguration() void
loadRoutes() void
«create»
run() represents the singleton pattern