

Module Interface Specification for The Crazy Tens

Team #25, The Crazy Four

Ruida Chen

Ammar Sharbat

Alvin Qian

Jiaming Li

January 17, 2026

1 Revision History

Date	Version	Notes
Nov 12th	Rev-1	Module M1-M11
Nov 12th	Rev-1	Module M12-M22
Nov 13th	Rev-1	Fix consistency
Nov 13th	Rev-1	Fix correlation
Nov 13th	Rev-1	Fix consistency
Jan 14th	Rev0	Remove module 20,21,22

2 Symbols, Abbreviations and Acronyms

See MG Documentation at [MG](#)

Contents

1 Revision History	i
2 Symbols, Abbreviations and Acronyms	ii
3 Introduction	1
4 Notation	1
5 Module Decomposition	1
6 MIS of API Module (M1)	3
6.1 Module	3
6.2 Uses	3
6.3 Syntax	3
6.3.1 Exported Constants	3
6.3.2 Exported Access Programs	3
6.4 Semantics	4
6.4.1 State Variables	4
6.4.2 Environment Variables	4
6.4.3 Assumptions	4
6.4.4 Access Routine Semantics	4
6.4.5 Local Functions	6
6.4.6 Considerations	6
7 MIS of Real-time Gateway Module (M2)	7
7.1 Module	7
7.2 Uses	7
7.3 Syntax	7
7.3.1 Exported Constants	7
7.3.2 Exported Access Programs	7
7.4 Semantics	7
7.4.1 State Variables	7
7.4.2 Environment Variables	8
7.4.3 Assumptions	8
7.4.4 Access Routine Semantics	8
7.4.5 Local Functions	9
7.4.6 Considerations	9
8 MIS of Matchmaking Module (M3)	10
8.1 Module	10
8.2 Uses	10
8.3 Syntax	10

8.3.1	Exported Constants	10
8.3.2	Exported Access Programs	10
8.4	Semantics	10
8.4.1	State Variables	10
8.4.2	Environment Variables	10
8.4.3	Assumptions	11
8.4.4	Access Routine Semantics	11
8.4.5	Local Functions	11
8.4.6	Considerations	12
9	MIS of Authentication Module (M4)	13
9.1	Module	13
9.2	Uses	13
9.3	Syntax	13
9.3.1	Exported Constants	13
9.3.2	Exported Access Programs	13
9.4	Semantics	14
9.4.1	State Variables	14
9.4.2	Environment Variables	14
9.4.3	Assumptions	14
9.4.4	Access Routine Semantics	14
9.4.5	Local Functions	16
9.4.6	Considerations	16
10	MIS of Repository Module (M5)	17
10.1	Module	17
10.2	Uses	17
10.3	Syntax	17
10.3.1	Exported Constants	17
10.3.2	Exported Access Programs	18
10.4	Semantics	18
10.4.1	State Variables	18
10.4.2	Environment Variables	19
10.4.3	Assumptions	19
10.4.4	Access Routine Semantics	19
10.4.5	Local Functions	21
10.4.6	Considerations	21
11	MIS of Audit Module (M6)	22
11.1	Module	22
11.2	Uses	22
11.3	Syntax	22
11.3.1	Exported Constants	22

11.3.2 Exported Access Programs	22
11.4 Semantics	22
11.4.1 State Variables	22
11.4.2 Environment Variables	22
11.4.3 Assumptions	23
11.4.4 Access Routine Semantics	23
11.4.5 Local Functions	24
11.4.6 Considerations	24
12 MIS of Real-time Client Module (M7)	25
12.1 Module	25
12.2 Uses	25
12.3 Syntax	25
12.3.1 Exported Constants	25
12.3.2 Exported Access Programs	25
12.4 Semantics	25
12.4.1 State Variables	25
12.4.2 Environment Variables	25
12.4.3 Assumptions	25
12.4.4 Access Routine Semantics	26
12.4.5 Local Functions	26
12.4.6 Considerations	26
13 MIS of Application Shell Module (M8)	27
13.1 Module	27
13.2 Uses	27
13.3 Syntax	27
13.3.1 Exported Constants	27
13.3.2 Exported Access Programs	27
13.4 Semantics	27
13.4.1 State Variables	27
13.4.2 Environment Variables	27
13.4.3 Assumptions	28
13.4.4 Access Routine Semantics	28
13.4.5 Local Functions	28
13.4.6 Considerations	28
14 MIS of Authentication Client Module (M9)	29
14.1 Module	29
14.2 Uses	29
14.3 Syntax	29
14.3.1 Exported Constants	29
14.3.2 Exported Access Programs	29

14.4 Semantics	29
14.4.1 State Variables	29
14.4.2 Environment Variables	29
14.4.3 Assumptions	30
14.4.4 Access Routine Semantics	30
14.4.5 Local Functions	31
14.4.6 Considerations	32
15 MIS of Lobby View Module (M10)	33
15.1 Module	33
15.2 Uses	33
15.3 Syntax	33
15.3.1 Exported Constants	33
15.3.2 Exported Access Programs	33
15.4 Semantics	33
15.4.1 State Variables	33
15.4.2 Environment Variables	33
15.4.3 Assumptions	33
15.4.4 Access Routine Semantics	34
15.4.5 Local Functions	34
15.4.6 Considerations	34
16 MIS of Game Board View Module (M11)	35
16.1 Module	35
16.2 Uses	35
16.3 Syntax	35
16.3.1 Exported Constants	35
16.3.2 Exported Access Programs	35
16.4 Semantics	35
16.4.1 State Variables	35
16.4.2 Environment Variables	35
16.4.3 Assumptions	35
16.4.4 Access Routine Semantics	36
16.4.5 Local Functions	36
16.4.6 Considerations	36
17 MIS of Move Controller Module (M12)	37
17.1 Module	37
17.2 Uses	37
17.3 Syntax	37
17.3.1 Exported Constants	37
17.3.2 Exported Access Programs	37
17.4 Semantics	37

17.4.1 State Variables	37
17.4.2 Environment Variables	37
17.4.3 Assumptions	37
17.4.4 Access Routine Semantics	38
17.4.5 Local Functions	38
17.4.6 Considerations	38
18 MIS of Scoreboard View Module (M13)	39
18.1 Module	39
18.2 Uses	39
18.3 Syntax	39
18.3.1 Exported Constants	39
18.3.2 Exported Access Programs	39
18.4 Semantics	39
18.4.1 State Variables	39
18.4.2 Environment Variables	39
18.4.3 Assumptions	40
18.4.4 Access Routine Semantics	40
18.4.5 Local Functions	41
18.4.6 Considerations	41
19 MIS of Profile View Module (M14)	42
19.1 Module	42
19.2 Uses	42
19.3 Syntax	42
19.3.1 Exported Constants	42
19.3.2 Exported Access Programs	42
19.4 Semantics	42
19.4.1 State Variables	42
19.4.2 Environment Variables	42
19.4.3 Assumptions	42
19.4.4 Access Routine Semantics	43
19.4.5 Local Functions	43
19.4.6 Considerations	43
20 MIS of Game Engine Module (M15)	44
20.1 Module	44
20.2 Uses	44
20.3 Syntax	44
20.3.1 Exported Constants	44
20.3.2 Exported Access Programs	44
20.4 Semantics	44
20.4.1 State Variables	44

20.4.2 Environment Variables	44
20.4.3 Assumptions	44
20.4.4 Access Routine Semantics	45
20.4.5 Local Functions	45
20.4.6 Considerations	45
21 MIS of Rules Module (M16)	46
21.1 Module	46
21.2 Uses	46
21.3 Syntax	46
21.3.1 Exported Constants	46
21.3.2 Exported Access Programs	46
21.4 Semantics	47
21.4.1 State Variables	47
21.4.2 Environment Variables	47
21.4.3 Assumptions	47
21.4.4 Access Routine Semantics	47
21.4.5 Local Functions	49
21.4.6 Considerations	49
22 MIS of Scoring Module (M17)	50
22.1 Module	50
22.2 Uses	50
22.3 Syntax	50
22.3.1 Exported Constants	50
22.3.2 Exported Access Programs	50
22.4 Semantics	50
22.4.1 State Variables	50
22.4.2 Environment Variables	50
22.4.3 Assumptions	50
22.4.4 Access Routine Semantics	51
22.4.5 Local Functions	51
22.4.6 Considerations	51
23 MIS of Base Conversion Module (M18)	52
23.1 Module	52
23.2 Uses	52
23.3 Syntax	52
23.3.1 Exported Constants	52
23.3.2 Exported Access Programs	52
23.4 Semantics	52
23.4.1 State Variables	52
23.4.2 Environment Variables	52

23.4.3 Assumptions	52
23.4.4 Access Routine Semantics	53
23.4.5 Local Functions	53
23.4.6 Considerations	54
24 MIS of Game Actions Module (M19)	55
24.1 Module	55
24.2 Uses	55
24.3 Syntax	55
24.3.1 Exported Constants	55
24.3.2 Exported Access Programs	55
24.4 Semantics	55
24.4.1 State Variables	55
24.4.2 Environment Variables	55
24.4.3 Assumptions	55
24.4.4 Access Routine Semantics	56
24.4.5 Local Functions	56
24.4.6 Considerations	56
25 MIS of Operating System Module (M20)	57
25.1 Module	57
25.2 Uses	57
25.3 Syntax	57
25.3.1 Exported Constants	57
25.3.2 Exported Access Programs	57
25.4 Semantics	57
25.4.1 State Variables	57
25.4.2 Environment Variables	57
25.4.3 Assumptions	57
25.4.4 Access Routine Semantics	57
25.4.5 Local Functions	58
25.4.6 Considerations	58
26 MIS of Browser Runtime Module (M21)	59
26.1 Module	59
26.2 Uses	59
26.3 Syntax	59
26.3.1 Exported Constants	59
26.3.2 Exported Access Programs	59
26.4 Semantics	59
26.4.1 State Variables	59
26.4.2 Environment Variables	59
26.4.3 Assumptions	59

26.4.4	Access Routine Semantics	59
26.4.5	Local Functions	60
26.4.6	Considerations	60
27	MIS of Database Module (M22)	61
27.1	Module	61
27.2	Uses	61
27.3	Syntax	61
27.3.1	Exported Constants	61
27.3.2	Exported Access Programs	61
27.4	Semantics	61
27.4.1	State Variables	61
27.4.2	Environment Variables	61
27.4.3	Assumptions	61
27.4.4	Access Routine Semantics	62
27.4.5	Local Functions	62
27.4.6	Considerations	62
28	Appendix	63

3 Introduction

The following document details the Module Interface Specifications for The Crazy Tens

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at <https://github.com/The-Crazy-Four-Games/Crazy-Eights-Game>.

4 Notation

The structure of the MIS for modules comes from ?, with the addition that template modules have been adapted from ?. The mathematical notation comes from Chapter 3 of ?. For instance, the symbol $::=$ is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | \dots | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by The Crazy Tens.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	\mathbb{Z}	a number without a fractional component in $(-\infty, \infty)$
natural number	\mathbb{N}	a number without a fractional component in $[1, \infty)$
real	\mathbb{R}	any number in $(-\infty, \infty)$

The specification of The Crazy Tens uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, The Crazy Tens uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	Level 2	Level 3 (Leaf Modules)
		(None)
Hardware-Hiding Module	(Core Domain Logic)	M15 M16 M17 M18 M19
Behaviour-Hiding Module		M1 M2 M3 M4 M5 M6
Software Decision Module		M7 M8 M9
	Backend (Server)	M10 M11 M12 M13 M14
	Frontend (Client)	

Table 1: Module Hierarchy

6 MIS of API Module (M1)

6.1 Module

API

6.2 Uses

- M4 Authentication Module
- M5 Repository Module
- M6 Audit Module

6.3 Syntax

6.3.1 Exported Constants

- **APIVersion:** string := “v1”

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
postRegister	req: RegisterRequest	HttpResponse	BadRequest, UsernameTaken, WeakPassword
postLogin	req: LoginRequest	HttpResponse	BadRequest, InvalidCredentials, AccountNotFound
postGuestSession	req: GuestSessionRequest	HttpResponse	BadRequest
postLogout	req: LogoutRequest	HttpResponse	BadRequest, InvalidToken
getProfile	req: AuthenticateRequest	HttpResponse	Unauthorized, RecordNotFound
putProfile	req: UpdateProfileRequest	HttpResponse	BadRequest, Unauthorized, RecordNotFound
postNewGame	req: NewGameRequest	HttpResponse	BadRequest, Unauthorized
getHealth	req: HttpRequest	HttpResponse	None

6.4 Semantics

6.4.1 State Variables

None. (Stateless HTTP routes; no persistent state is maintained by this module.)

6.4.2 Environment Variables

- **HTTPServer**: The web server/runtime that receives HTTP requests and dispatches routes.
- **Auth**: Authentication service provided by M4.
- **Repository**: Persistence interface provided by M5.
- **Audit**: Audit logging service provided by M6.

6.4.3 Assumptions

- Requests and responses follow the REST endpoint structure and payload schemas defined by the system.
- Authentication-protected endpoints include a session token (e.g., `Authorization` header).
- M4, M5, and M6 are correctly configured and available in the server runtime.

6.4.4 Access Routine Semantics

`postRegister(req)`

- transition: Validate *req* payload (username/password). Call `Auth.createAccount`. Log an authentication audit event via `Audit.logAuthEvent`.
- output: Return `HttpResponse` containing success status and auth token (on success).
- exception: `BadRequest` if payload invalid. Propagate `UsernameTaken`/`WeakPassword` from authentication logic.

`postLogin(req)`

- transition: Validate *req*. Call `Auth.login`. Log auth event via `Audit.logAuthEvent`.
- output: Return `HttpResponse` containing auth token (on success).
- exception: `BadRequest` if payload invalid. `InvalidCredentials`/`AccountNotFound` if login fails.

`postGuestSession(req)`

- transition: Validate *req*. Call **Auth.createGuestSession**. Log auth event via **Audit.logAuthEvent**.
- output: Return **HttpResponse** containing guest token.
- exception: **BadRequest** if payload invalid.

postLogout(*req*)

- transition: Validate *req*. Call **Auth.logout**. Log auth event via **Audit.logAuthEvent**.
- output: Return **HttpResponse** confirming logout.
- exception: **BadRequest** if payload invalid. **InvalidToken** if token invalid.

getProfile(*req*)

- transition: Verify token from *req* by calling **Auth.verifyToken**. On success, read user profile from **Repository** (e.g., `getPlayerProfile`/equivalent). Log a profile access event via **Audit.logSystemEvent**.
- output: Return **HttpResponse** containing the user profile data.
- exception: **Unauthorized** if token invalid/expired. **RecordNotFound** if profile/user does not exist.

putProfile(*req*)

- transition: Verify token via **Auth.verifyToken**. Validate profile update payload. Update profile via **Repository** (e.g., `updatePlayerProfile`/equivalent). Log update event via **Audit.logSystemEvent**.
- output: Return **HttpResponse** containing updated profile.
- exception: **BadRequest** if payload invalid. **Unauthorized** if token invalid/expired. **RecordNotFound** if user/profile not found.

postNewGame(*req*)

- transition: Verify token via **Auth.verifyToken**. Validate request body (game setup parameters). Persist/initialize game bootstrap data via **Repository** as required by the SRS. Log gameplay/system event via **Audit.logGameplayEvent** or **Audit.logSystemEvent**. (Real-time gameplay proceeds through M2.)
- output: Return **HttpResponse** confirming game creation/bootstrapping and any returned identifiers.
- exception: **BadRequest** if payload invalid. **Unauthorized** if token invalid/expired.

`getHealth(req)`

- transition: Return basic service status (optionally include shallow checks of dependencies).
- output: Return `HttpResponse` with status 200 `OK` when the API server is up.
- exception: None.

6.4.5 Local Functions

None.

6.4.6 Considerations

- **Secret:** REST endpoint structure, request/response payload schemas, and HTTP conventions for stateless backend capabilities.
- **Service:** exposes stateless HTTP routes for authentication, profile management, and bootstrapping new games as defined in the SRS; real-time gameplay is handled separately by M2.
- This module should remain stateless; persistence and security concerns are delegated to M5 and M4, while operational logging is delegated to M6.

7 MIS of Real-time Gateway Module (M2)

7.1 Module

Real-time Gateway

7.2 Uses

- M4 Authentication Module
- M5 Repository Module
- M15 Game Action Module

7.3 Syntax

7.3.1 Exported Constants

None.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
handleConnection	socket: ClientSocket	void	SessionError
handleJoinGame	socket: ClientSocket, data: JoinGameData	void	SessionError, NotFound
handleSubmitMove	socket: ClientSocket, data: MoveData	void	InvalidMove, Not Your Turn, SessionError
registerGameSession	gameID: GameID, ses- sion: GameSession	void	LobbyError
emitGameState	socket: ClientSocket, state: GameState	void	
broadcastGameState	roomID: GameID, state: GameState	void	

7.4 Semantics

7.4.1 State Variables

- **activeGames:** Map<*GameID*, *GameSession*> — A map holding the live *GameSession* objects for all currently active games.

7.4.2 Environment Variables

- **RealtimeAdapter**: The environment (e.g., a WebSocket server) that manages client connections.
- **ClientSocket**: An opaque handle representing a single client connection.

7.4.3 Assumptions

- The **RealtimeAdapter** is running and forwards events to these handlers.
- **AuthToken** (inside **JoinGameData**) is verifiable by M4.
- M15 provides the legality check for the "move" operation and the state evolution.

7.4.4 Access Routine Semantics

handleConnection(*socket*)

- transition: Registers the new *socket* with the **RealtimeAdapter**. Attaches handlers for **handleJoinGame**, **handleSubmitMove**, etc.
- output: **void**.
- exception: **SessionError** if the connection handshake fails.

handleJoinGame(*socket*, *data*)

- transition: 1. Calls **M4.verifyToken(*data.authToken*)** to get a **UserID**. 2. Retrieves the **GameSession** from **activeGames** using ***data.gameID***. 3. Associates the *socket* with the **GameSession** and **UserID**.
- output: Calls **emitGameState(*socket*, *currentState*)** to send the current state to the joining player.
- exception: **SessionError** if **authToken** is invalid. **NotFound** if the **GameSession** does not exist in **activeGames**.

handleSubmitMove(*socket*, *data*)

- transition: Identifies **UserID** and **GameSession** from the *socket*. Calls **M19.validateAction(*data.gameState*)** to check legality. If valid, calls **M19.executeAction(*data.gameState*)** to get the **newGameState**. Updates the **GameSession** in **activeGames** with **newGameState**.
- output: Calls **broadcastGameState(*gameID*, *newGameState*)** to send the new state to all players in that session.
- exception: **InvalidMove**. **NotYourTurn**. **SessionError**.

registerGameSession(*gameID*, *session*)

- transition: Called by M3. Adds the new *session* to the **activeGames** map.
- output: **void**.
- exception: **LobbyError** if the *gameID* is already active.

7.4.5 Local Functions

None.

7.4.6 Considerations

- The secret of this module is the management of stateful connections and the mapping of **ClientSocket** handles to active **GameSessions**.
- This module acts as the server-authoritative state synchronizer, delegating all game logic to M19.

8 MIS of Matchmaking Module (M3)

8.1 Module

Matchmaking

8.2 Uses

- M4 Authentication Module
- M15 Game Engine Module
- M2 Real-time Gateway Module

8.3 Syntax

8.3.1 Exported Constants

None.

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
createLobby	token: AuthToken, options: GameOptions	lobby: Lobby	LobbyError
joinLobby	token: AuthToken, lobbyID: LobbyID	void	LobbyFull, LobbyNotFound, AuthError
startMatch	token: AuthToken, lobbyID: LobbyID	gameID: GameID	LobbyNotFound, NotLobbyHost, AuthError, GameCreationError

8.4 Semantics

8.4.1 State Variables

- **lobbies:** Map[LobbyID, Lobby] — Holds all active, waiting-for-players **Lobby** objects [cite: 639-640].

8.4.2 Environment Variables

None.

8.4.3 Assumptions

- Any `AuthToken` is verifiable by M4.
- Modules M15 and M2 are available when `startMatch` is invoked.

8.4.4 Access Routine Semantics

`createLobby(token, options)`

- transition: Calls M4.`verifyToken(token)` to get `UserID`. Generates a unique `LobbyID`. Creates a new `Lobby` object (setting `hostID = UserID`). Adds this new object to the `lobbies` map[cite: 636].
- output: Returns the newly created `Lobby`.
- exception: `AuthError`. `LobbyError`.

`joinLobby(token, lobbyID)`

- transition: Calls M4.`verifyToken(token)` to get `UserID`. Looks up the `Lobby` in `lobbies`. Verifies `Lobby.status == 'waiting'` and `Lobby.players.length < MAX_PLAYERS`. If valid, appends `UserID` to the `Lobby.players` array[cite: 636].
- output: `void`.
- exception: `LobbyFull`. `LobbyNotFound`. `AuthError`.

`startMatch(token, lobbyID)`

- transition: 1. Calls M4.`verifyToken(token)` to get `UserID`. 2. Looks up the `Lobby` in `lobbies` and verifies `UserID` is the host. 3. Calls M15.`createGame(lobby.players, options)` to get a new `GameState`[cite: 669]. 4. Creates a new `GameSession` object. 5. Calls M2.`registerGameSession(lobbyID, newGameSession)` to hand off the session to the real-time server [cite: 630-631]. 6. Removes the `Lobby` from the `lobbies` map[cite: 636].
- output: Returns the `GameID` (which may be the `LobbyID`).
- exception: `LobbyNotFound`. `NotLobbyHost`. `AuthError`. `GameCreationError`.

8.4.5 Local Functions

None.

8.4.6 Considerations

- The secret of this module is the `Lobby` data structure and the management of the `lobbies` map.
- This module bridges the stateless API (M1) and the stateful game session (M2) by handling the pre-game lobby state [cite: 636-637].

9 MIS of Authentication Module (M4)

9.1 Module

Authentication

9.2 Uses

None.

9.3 Syntax

9.3.1 Exported Constants

None.

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
createAccount	username: string, password: string	AuthResult	UsernameTaken, WeakPassword, CredentialStoreError
login	username: string, password: string	AuthResult	InvalidCredentials, AccountNotFound, CredentialStoreError
createGuestSession	deviceId: string	AuthResult	CredentialStoreError
logout	token: SessionToken	void	InvalidToken
issueToken	userId: UserID, role: UserRole	SessionToken	TokenSigningError
verifyToken	token: SessionToken	TokenClaims	InvalidToken, ExpiredToken, TokenSigningError
refreshToken	token: SessionToken	SessionToken	InvalidToken, ExpiredToken, TokenSigningError

9.4 Semantics

9.4.1 State Variables

- **hashConfig**: Configuration for password hashing (e.g., algorithm selection, cost parameters).
- **tokenSigner**: A component holding token-signing capability (e.g., HMAC secret or private key handle).
- **credentialStore**: Internal credential storage mechanism used by this module (implementation-defined).
- **activeSessions**: A mapping from `SessionToken` to session metadata (e.g., `userId`, `expiry`, `role`). (If sessions are stateless JWTs, this may be minimal or omitted.)

9.4.2 Environment Variables

- **TokenSigningKey**: Secret key material used to sign/verify tokens (stored securely outside the codebase).
- **SystemClock**: Source of current time used for expiry validation.

9.4.3 Assumptions

- Password hashing configuration is correctly set and kept consistent across deployments.
- **TokenSigningKey** is available and protected (not exposed to clients).
- The internal **credentialStore** is available and correctly initialized.

9.4.4 Access Routine Semantics

createAccount(*username*, *password*)

- transition: Validate *username* format and *password* strength. Hash *password* using **hashConfig**. Store account and credential info in **credentialStore**. Issue a session token for the new account via **issueToken**.
- output: Return `AuthResult` containing (at minimum) a `SessionToken` and user identity information.
- exception: `UsernameTaken` if *username* already exists in **credentialStore**. `WeakPassword` if *password* fails policy. `CredentialStoreError` if storage fails.

login(*username*, *password*)

- transition: Retrieve credential record for *username* from **credentialStore**. Verify *password* against stored hash. On success, issue a new token via **issueToken**.

- output: Return `AuthResult` containing a valid `SessionToken`.
- exception: `AccountNotFound` if `username` does not exist. `InvalidCredentials` if verification fails. `CredentialStoreError` if retrieval fails.

`createGuestSession(deviceId)`

- transition: Create or reuse a guest identity bound to `deviceId` (policy-defined). Store minimal guest metadata in `credentialStore` if required. Issue a guest token via `issueToken`.
- output: Return `AuthResult` containing a guest `SessionToken`.
- exception: `CredentialStoreError` if storage fails.

`logout(token)`

- transition: Invalidate `token`. If tokens are stateless, record the token (or its identifier) in a denylist until expiry; otherwise remove from `activeSessions`.
- output: `void`.
- exception: `InvalidToken` if `token` is malformed or cannot be invalidated under the configured policy.

`issueToken(userId, role)`

- transition: Construct token claims (e.g., `userId`, `role`, issued-at, expiry). Sign token using `tokenSigner` backed by `TokenSigningKey`.
- output: Return a signed `SessionToken`.
- exception: `TokenSigningError` if signing fails or key material is unavailable.

`verifyToken(token)`

- transition: Verify signature using `TokenSigningKey`. Validate expiry against `SystemClock`. Optionally check denylist / `activeSessions`.
- output: Return `TokenClaims` extracted from the token.
- exception: `InvalidToken` if signature/format validation fails. `ExpiredToken` if expiry has passed. `TokenSigningError` if verification fails due to key issues.

`refreshToken(token)`

- transition: Verify `token` via `verifyToken`. If eligible for refresh, issue a new token with extended expiry via `issueToken`. Optionally revoke the old token (denylist) depending on policy.
- output: Return a new `SessionToken`.
- exception: `InvalidToken` if verification fails. `ExpiredToken` if the token is no longer refreshable. `TokenSigningError` if issuing the new token fails.

9.4.5 Local Functions

None.

9.4.6 Considerations

- **Secret:** password hashing configuration, credential storage details, and token-signing keys.
- **Service:** creates accounts, validates logins, manages guest sessions, and issues/verifies tokens used by the rest of the backend.
- Other backend modules should not handle raw passwords or token signing directly; they should rely on this module's access routines.
- If hashing algorithms or token mechanisms change, modifications are localized to this module; callers remain unchanged.

10 MIS of Repository Module (M5)

10.1 Module

Repository

10.2 Uses

None.

10.3 Syntax

10.3.1 Exported Constants

None.

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
findPlayerByUsername	username: string	Player	RecordNotFound, DatabaseConnectionError
findPlayerById	playerId: PlayerID	Player	RecordNotFound, DatabaseConnectionError
createPlayer	data: PlayerData	Player	UniqueConstraintViolation, DatabaseConnectionError
updatePlayerProfile	playerId: PlayerID, data: PlayerProfileData	Player	RecordNotFound, DatabaseConnectionError
storeCredential	playerId: PlayerID, cred: CredentialData	void	RecordNotFound, UniqueConstraintViolation, DatabaseConnectionError
getCredentialByUsername	username: string	CredentialRecord	RecordNotFound, DatabaseConnectionError
saveMatchResult	result: MatchResult	void	DatabaseConnectionError
getMatchHistory	playerId: PlayerID, limit: nat, offset: nat	seq of MatchResult	DatabaseConnectionError
getPlayerStats	playerId: PlayerID	PlayerStats	RecordNotFound, DatabaseConnectionError
deletePlayer	playerId: PlayerID	void	RecordNotFound, DatabaseConnectionError

10.4 Semantics

10.4.1 State Variables

- **dbConnectionPool:** A connection pool managing active connections to the PostgreSQL database.

10.4.2 Environment Variables

- **PostgreSQLServer**: The external PostgreSQL database instance that executes all SQL queries.
- **ConnectionString**: A secure configuration value used to initialize database connectivity.

10.4.3 Assumptions

- **PostgreSQLServer** is running and reachable from the server-side runtime.
- **ConnectionString** is provided securely and grants appropriate permissions.
- The database schema for players, credentials, match history, and statistics is initialized and consistent with this module's queries.

10.4.4 Access Routine Semantics

findPlayerByUsername(*username*)

- transition: Acquire a connection from **dbConnectionPool**. Execute a SQL SELECT over the player table filtered by *username*.
- output: Return the **Player** record.
- exception: **RecordNotFound** if no player matches *username*. **DatabaseConnectionError** if query/connection fails.

findPlayerById(*playerId*)

- transition: Acquire a connection. Execute a SQL SELECT over the player table filtered by *playerId*.
- output: Return the **Player** record.
- exception: **RecordNotFound** if *playerId* does not exist. **DatabaseConnectionError** if query/connection fails.

createPlayer(*data*)

- transition: Acquire a connection. Execute a SQL INSERT to create a new player using *data*.
- output: Return the created **Player** (including generated **PlayerID**).
- exception: **UniqueConstraintViolation** if a unique field (e.g., *username*) already exists. **DatabaseConnectionError** if query/connection fails.

updatePlayerProfile(*playerId*, *data*)

- transition: Acquire a connection. Execute a SQL UPDATE to modify the profile fields for *playerId* using *data*.
- output: Return the updated Player.
- exception: RecordNotFound if *playerId* does not exist. DatabaseConnectionError if query/connection fails.

storeCredential(*playerId*, *cred*)

- transition: Acquire a connection. Execute a SQL INSERT (or UPSERT, if supported by the schema) to store credential material for *playerId*.
- output: void.
- exception: RecordNotFound if *playerId* does not exist. UniqueConstraintViolation if the credential record violates a uniqueness rule. DatabaseConnectionError if query/connection fails.

getCredentialByUsername(*username*)

- transition: Acquire a connection. Execute a SQL SELECT joining username → player → credential record (schema-dependent).
- output: Return the CredentialRecord.
- exception: RecordNotFound if no credential is found for *username*. DatabaseConnectionError if query/connection fails.

saveMatchResult(*result*)

- transition: Acquire a connection. Execute a SQL INSERT into match history / results tables using *result*.
- output: void.
- exception: DatabaseConnectionError if query/connection fails.

getMatchHistory(*playerId*, *limit*, *offset*)

- transition: Acquire a connection. Execute a SQL SELECT over match history filtered by *playerId*, ordered by time, returning a window defined by *limit* and *offset*.
- output: Return a sequence of MatchResult.
- exception: DatabaseConnectionError if query/connection fails.

getPlayerStats(*playerId*)

- transition: Acquire a connection. Execute a SQL **SELECT** (and/or aggregate queries) to retrieve computed statistics for *playerId*.
- output: Return **PlayerStats**.
- exception: **RecordNotFound** if *playerId* does not exist (or has no stats record per schema). **DatabaseConnectionError** if query/connection fails.

deletePlayer(*playerId*)

- transition: Acquire a connection. Execute SQL **DELETE** operations (possibly cascading per schema) to remove the player and related records.
- output: **void**.
- exception: **RecordNotFound** if *playerId* does not exist. **DatabaseConnectionError** if query/connection fails.

10.4.5 Local Functions

None.

10.4.6 Considerations

- **Secret:** the database implementation (PostgreSQL), schema design, optimized SQL queries, and database access strategies (e.g., pooling, transactions).
- **Service:** provides a clean persistence interface for storing and retrieving players, credentials, match history, and statistics, shielding callers from database specifics.
- This module abstracts all CRUD operations and schema details; callers never issue SQL directly.
- If the persistence technology changes (e.g., PostgreSQL to another DB), only this module needs to be rewritten; callers remain unchanged.

11 MIS of Audit Module (M6)

11.1 Module

Audit

11.2 Uses

None.

11.3 Syntax

11.3.1 Exported Constants

- **DefaultRetentionDays**: nat := 30
- **MaxEventPayloadSize**: nat := 8192 (bytes)

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
logAuthEvent	event: AuthAuditEvent	void	LogStoreError
logGameplayEvent	event: GameplayAuditEvent	void	LogStoreError
logSystemEvent	event: SystemAuditEvent	void	LogStoreError
queryAuditEvents	filter: AuditQueryFilter	seq of AuditEvent	LogStoreError
purgeExpiredEvents	retentionDays: nat	nat	LogStoreError
redactEventPayload	eventId: AuditEventID, fields: seq of string	void	RecordNotFound, LogStoreError

11.4 Semantics

11.4.1 State Variables

- **retentionPolicyDays**: nat := **DefaultRetentionDays**
- **logStore**: Internal storage target/driver for audit events (implementation-defined; may be DB table, file sink, external log service).

11.4.2 Environment Variables

- **SystemClock**: Source of current time for timestamps and retention enforcement.

- **AuditStorageConfig**: Configuration describing storage target(s), retention, and access credentials for the logging backend.

11.4.3 Assumptions

- The event schema (fields for authentication/gameplay/system events) matches the storage format used by **logStore**.
- The storage target(s) configured by **AuditStorageConfig** are reachable during normal operation.
- Retention policies are chosen to satisfy operational debugging, security, and compliance requirements.

11.4.4 Access Routine Semantics

logAuthEvent(*event*)

- transition: Validate *event* payload size $\leq \text{MaxEventPayloadSize}$. Attach timestamp from **SystemClock**. Serialize and append/store in **logStore** using the authentication-event schema.
- output: **void**.
- exception: **LogStoreError** if storage/append fails.

logGameplayEvent(*event*)

- transition: Validate payload size and required fields. Attach timestamp. Serialize and append/store in **logStore** using the gameplay-event schema.
- output: **void**.
- exception: **LogStoreError** if storage/append fails.

logSystemEvent(*event*)

- transition: Validate payload size and required fields. Attach timestamp. Serialize and append/store in **logStore** using the system-event schema.
- output: **void**.
- exception: **LogStoreError** if storage/append fails.

queryAuditEvents(*filter*)

- transition: Translate *filter* constraints (time window, event type, user/session identifiers, severity, correlation id, etc.) into **logStore** query operations. Execute query against **logStore**. Apply any post-filters (e.g., pagination) as required.

- output: Return a sequence of `AuditEvent` matching *filter*.
- exception: `LogStoreError` if querying fails.

`purgeExpiredEvents(retentionDays)`

- transition: Compute cutoff time = `SystemClock` – *retentionDays*. Delete, archive, or compact entries older than cutoff according to `AuditStorageConfig` policy.
- output: Return the number of events purged (nat).
- exception: `LogStoreError` if purge/archive fails.

`redactEventPayload(eventId, fields)`

- transition: Locate *eventId* in `logStore`. Replace, mask, or remove specified *fields* in the persisted payload (schema-dependent). Persist the redacted version and mark the record as redacted.
- output: `void`.
- exception: `RecordNotFound` if *eventId* does not exist. `LogStoreError` if update/writeback fails.

11.4.5 Local Functions

None.

11.4.6 Considerations

- **Secret:** the exact event schema, retention policy, and storage targets for operational logs.
- **Service:** captures authentication, gameplay, and system events to support debugging, compliance, security, and user inquiries.
- This module is a library utility; it should not impose dependencies on other modules.
- If storage targets or retention strategies change, updates are localized to this module; event producers remain unchanged as long as the API is stable.

12 MIS of Real-time Client Module (M7)

12.1 Module

Real-time Client

12.2 Uses

- M2 Real-time Gateway Module

12.3 Syntax

12.3.1 Exported Constants

None.

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
connect	-	void	ConnectionFailed
disconnect	-	void	
on	eventName: 'gameStateUpdate', callback: (state) => void	void	
emit	eventName: 'submitMove', move: Move	void	ConnectionFailed

12.4 Semantics

12.4.1 State Variables

- **socket:** Socket — The `Socket.io-client` instance.
- **isConnected:** bool — Flag indicating the connection status.

12.4.2 Environment Variables

- **BrowserRuntime (M21):** The client's web browser environment providing Web-
Socket APIs.

12.4.3 Assumptions

- The M2 server is running and its URL is accessible to the client.
- The browser environment (M21) supports WebSockets.

12.4.4 Access Routine Semantics

connect()

- transition: Initializes and establishes the WebSocket connection to M2. Sets **socket** to the new instance and **isConnected** to **true** on success.
- output: **void**.
- exception: **ConnectionFailed** if the connection times out or is rejected.

disconnect()

- transition: Closes the active WebSocket connection. Sets **isConnected** to **false** and **socket** to **null**.
- output: **void**.

on(*eventName*, *callback*)

- transition: Registers an event listener on the **socket** instance. When M2 emits an event matching *eventName* (e.g., 'gameStateUpdate'), the *callback* is invoked with the data payload.
- output: **void**.

emit(*eventName*, *move*)

- transition: Serializes and sends the *move* data to the M2 server over the **socket** connection, under the *eventName* (e.g., 'submitMove').
- output: **void**.
- exception: **ConnectionFailed** if **isConnected** is **false**.

12.4.5 Local Functions

None.

12.4.6 Considerations

- The secret of this module is the WebSocket connection state and reconnection logic.
- It is the client-side counterpart to M2.
- UI modules (e.g., M11, M12) use this module to receive state updates and send user actions.

13 MIS of Application Shell Module (M8)

13.1 Module

Application Shell

13.2 Uses

- M21 Browser Runtime Module
- M9 Authentication Client Module
- M10 Lobby View Module
- M11 Game Board View Module
- M14 Profile View Module

13.3 Syntax

13.3.1 Exported Constants

None.

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ReactProps	JSX.Element	RouteNotFound

13.4 Semantics

13.4.1 State Variables

- **currentUser**: User — null — Stores the state of the currently logged-in user.
- **currentRoute**: string — The active route from the browser's URL.

13.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing the DOM for rendering and the URL History API for routing.

13.4.3 Assumptions

- The React library is loaded in the M21 environment.
- The browser supports the History API.
- Modules M9, M10, M11, and M14 are available to be rendered as children.

13.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads the URL path from the **BrowserRuntime (M20)** to update **currentRoute**. Reads the authentication status to update **currentUser**. Renders the global layout (header, footer). Selectively renders a child module (M9, M10, M11, or M14) based on **currentRoute** and **currentUser**.
- output: Returns a React Element (`JSX.Element`) for the **BrowserRuntime (M20)** to render to the DOM.
- exception: `RouteNotFound` if **currentRoute** does not match any entry in the application's routing table.

13.4.5 Local Functions

None.

13.4.6 Considerations

- The secret of this module is the application routing table and the global layout structure.
- This module acts as a controller view, deciding which page (M10, M11, M14) to display based on URL and authentication state.

14 MIS of Authentication Client Module (M9)

14.1 Module

Authentication Client

14.2 Uses

- M1 API Module

14.3 Syntax

14.3.1 Exported Constants

- **TokenStorageKey**: string := “authToken”

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
renderLoginForm	None	ViewModel	None
renderSignupForm	None	ViewModel	None
submitLogin	username: string, password: string	AuthViewState	InvalidInput, AuthFailed, NetworkError
submitSignup	username: string, password: string	AuthViewState	InvalidInput, AuthFailed, NetworkError
submitLogout	None	AuthViewState	NetworkError
storeToken	token: string	void	TokenStorageError
loadToken	None	OptionalString	TokenStorageError
clearToken	None	void	TokenStorageError

14.4 Semantics

14.4.1 State Variables

- **currentToken**: OptionalString (cached token currently held by the client)
- **authStatus**: AuthStatus (e.g., LoggedOut, LoggingIn, LoggedIn, Error)

14.4.2 Environment Variables

- **ClientRuntime**: Browser/client framework runtime that renders UI and handles events.

- **SecureStorage:** Client-side storage mechanism (e.g., cookie/local storage/session storage), implementation-defined.

14.4.3 Assumptions

- The client calls M1 for authentication-related HTTP requests (login/signup/logout).
- Token storage is best-effort secure according to the chosen client storage mechanism.
- Network calls may fail; the UI reports errors to the user.

14.4.4 Access Routine Semantics

renderLoginForm()

- transition: Construct a login form view model (username/password inputs, submit action, error placeholder).
- output: Return the `ViewModel` for the login form.
- exception: None.

renderSignupForm()

- transition: Construct a signup form view model (username/password inputs, submit action, error placeholder).
- output: Return the `ViewModel` for the signup form.
- exception: None.

submitLogin(*username, password*)

- transition: Validate inputs (non-empty, basic format checks). Call M1 authentication route (e.g., `postLogin`). On success, extract token from the response and call **storeToken**. Update **currentToken** and **authStatus**.
- output: Return `AuthViewState` indicating success/failure and any user-facing message.
- exception: `InvalidInput` if validation fails. `AuthFailed` if server rejects credentials. `NetworkError` if request fails.

submitSignup(*username, password*)

- transition: Validate inputs. Call M1 authentication route (e.g., `postRegister`). On success, store returned token via **storeToken**. Update **currentToken** and **authStatus**.
- output: Return `AuthViewState` indicating success/failure and any user-facing message.

- exception: `InvalidInput` if validation fails. `AuthFailed` if server rejects signup. `NetworkError` if request fails.

`submitLogout()`

- transition: If `currentToken` exists, call M1 logout route (e.g., `postLogout`) best-effort. Call `clearToken`. Set `currentToken := None` and `authStatus := LoggedOut`.
- output: Return `AuthViewState` indicating the user is logged out.
- exception: `NetworkError` if the logout request fails (token clearing still proceeds locally).

`storeToken(token)`

- transition: Persist *token* into `SecureStorage` under `TokenStorageKey`. Set `currentToken := token`.
- output: `void`.
- exception: `TokenStorageError` if the storage operation fails.

`loadToken()`

- transition: Read token from `SecureStorage` under `TokenStorageKey`. If present, set `currentToken := token`; otherwise set `currentToken := None`.
- output: Return `currentToken`.
- exception: `TokenStorageError` if the storage read fails.

`clearToken()`

- transition: Remove `TokenStorageKey` from `SecureStorage`. Set `currentToken := None`.
- output: `void`.
- exception: `TokenStorageError` if removal fails.

14.4.5 Local Functions

None.

14.4.6 Considerations

- **Secret:** decisions about secure token storage and the flows for refreshing or clearing credentials in the browser.
- **Service:** presents login, signup, and logout experiences while coordinating with **M1** for authentication calls.
- **Type Note:** `OptionalString = string ∪ {None}`.

15 MIS of Lobby View Module (M10)

15.1 Module

Lobby View

15.2 Uses

- M1 API Module
- M7 Real-time Client Module
- M21 Browser Runtime Module

15.3 Syntax

15.3.1 Exported Constants

None.

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ReactProps	JSX.Element	
handleCreateGame	-	void	CreateGameError
handleJoinGame	lobbyID: LobbyID	void	JoinGameError

15.4 Semantics

15.4.1 State Variables

- **lobbiesList**: Lobby[] — An array of available game lobbies.
- **selectedLobby**: LobbyID — null — The ID of the lobby currently selected in the UI.
- **isLoading**: bool — True if a create or join operation is in progress.

15.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing DOM rendering.

15.4.3 Assumptions

- Modules M1 and M7 are available and configured.
- This module is rendered by M8 (Application Shell).

15.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads **lobbiesList**, **selectedLobby**, and **isLoading** from state.
- output: Returns a **JSX.Element** that renders the UI for listing, creating, and joining game lobbies. Renders a loading indicator if **isLoading** is true.

handleCreateGame()

- transition: Sets **isLoading** to true. Calls **M1.createGame** to create a new lobby. On success, receives a **newLobbyID** and calls **handleJoinGame(newLobbyID)**.
- output: **void**.
- exception: **CreateGameError** (displayed in UI) if the **M1** call fails.

handleJoinGame(*lobbyID*)

- transition: Sets **isLoading** to true. Calls **M7.emit('joinGame', { lobbyID: lobbyID, ... })**. On success, the **M7/M2** connection will trigger a state change that **M8** will use to render **M11**.
- output: **void**.
- exception: **JoinGameError** (displayed in UI) if **M7** fails to join.

15.4.5 Local Functions

None.

15.4.6 Considerations

- The secret of this module is the UI layout for displaying, creating, and joining games.
- It coordinates user actions, calling **M1** for lobby creation and **M7** for joining a real-time session.

16 MIS of Game Board View Module (M11)

16.1 Module

Game Board View

16.2 Uses

- M21 Browser Runtime Module
- M12 Move Controller Module

16.3 Syntax

16.3.1 Exported Constants

None.

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ReactProps	JSX.Element	None

16.4 Semantics

16.4.1 State Variables

- **clientGameState**: GameState — The current game state object (hands, deck, discard pile).
- **validMoves**: Card[] — An array of cards in the player's hand that are legal to play.

16.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing DOM rendering and CSS.

16.4.3 Assumptions

- This module is rendered by M8 when a game is active.
- The **clientGameState** and **validMoves** are provided (likely as props).
- Event handlers from M12 are attached to the rendered elements.

16.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads **clientGameState** and **validMoves** from state/props.
- output: Returns a **JSX.Element** that renders the main game interface, including the player's hand, the discard pile, and the deck. It visually highlights any cards in the hand that are also present in the **validMoves** list.

16.4.5 Local Functions

None.

16.4.6 Considerations

- The secret of this module is the DOM/CSS structure and animation logic used to render the game board.
- This is primarily a "dumb" rendering component; it displays state and delegates user input handling to M12.

17 MIS of Move Controller Module (M12)

17.1 Module

Move Controller

17.2 Uses

- Real-time Client Module (M7)
- Browser Runtime Module (M21)

17.3 Syntax

17.3.1 Exported Constants

None.

17.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
handlePlayCard	card: Card	void	InvalidMoveUI
handleDrawCard	-	void	InvalidMoveUI

17.4 Semantics

17.4.1 State Variables

- selectedCard: Card
- uiFeedback: string

17.4.2 Environment Variables

- **BrowserRuntime (M21):** Provides user input events from the UI.

17.4.3 Assumptions

- Module M7 is connected.
- These access programs are bound to UI elements rendered by M11.

17.4.4 Access Routine Semantics

handlePlayCard(*card*)

- transition: Performs client-side pre-validation. If invalid, sets **uiFeedback**. If valid, calls M7.emit('submitMove', { action: 'play', payload: card }).
- output: **void**.
- exception: InvalidMoveUI (captured in **uiFeedback** state).

handleDrawCard()

- transition: Performs client-side pre-validation. If invalid, sets **uiFeedback**. If valid, calls M7.emit('submitMove', { action: 'draw' }).
- output: **void**.
- exception: InvalidMoveUI (captured in **uiFeedback** state).

17.4.5 Local Functions

- **clientSidePreValidation(...)**: Local logic to pre-check moves.

17.4.6 Considerations

- The secret of this module is the client-side input handling logic and pre-validation rules.
- It decouples the M11 view from the M7 client service.

18 MIS of Scoreboard View Module (M13)

18.1 Module

Scoreboard View

18.2 Uses

- M18 Base Conversion Module

18.3 Syntax

18.3.1 Exported Constants

- DefaultScoreBase: nat := 10
- AlternateScoreBase: nat := 12

18.3.2 Exported Access Programs

Name	In	Out	Exceptions
renderScoreboard	data: ScoreboardData	ViewModel	InvalidScoreData
toggleScoreBase	base: nat	void	InvalidBase
formatScore	score: int, base: nat	string	InvalidBase, InvalidScoreData
renderRoundSummary	summary: RoundSummaryData	ViewModel	InvalidScoreData
animateRoundTransition	from: ViewModel, to: ViewModel	void	None

18.4 Semantics

18.4.1 State Variables

- currentBase: nat := DefaultScoreBase
- lastRendered: ViewModel (cached view state for transitions/animations)

18.4.2 Environment Variables

- ClientRuntime: The client-side framework/runtime responsible for rendering UI components and handling user interactions.
- DisplayPreferences: User/UI preferences (e.g., preferred score base, accessibility settings).

18.4.3 Assumptions

- Input scores in **ScoreboardData** are valid integers produced by the game logic.
- Dozenal formatting uses the digit-symbol mapping provided by **M18**.
- The client runtime supports re-rendering and simple transitions/animations.

18.4.4 Access Routine Semantics

renderScoreboard(*data*)

- transition: Validate *data* (players, ordering, score values). For each displayed score, call **formatScore** to produce both decimal and dozenal strings (or the currently selected base). Produce a **ViewModel** suitable for rendering, and update **lastRendered**.
- output: Return the constructed **ViewModel**.
- exception: **InvalidScoreData** if *data* is missing required fields or contains invalid score values.

toggleScoreBase(*base*)

- transition: If *base* is **DefaultScoreBase** or **AlternateScoreBase**, set **currentBase** := *base* and update **DisplayPreferences** if applicable; otherwise reject.
- output: **void**.
- exception: **InvalidBase** if *base* is not supported.

formatScore(*score*, *base*)

- transition: Validate *score*. If *base* = 10, convert *score* to decimal string using standard formatting. If *base* = 12, call **M18.decimalToDozenal(score)** to obtain the dozenal string.
- output: Return the formatted score string.
- exception: **InvalidBase** if *base* is unsupported. **InvalidScoreData** if *score* is not a valid integer (implementation-defined).

renderRoundSummary(*summary*)

- transition: Validate *summary* fields (round index, deltas, totals). Format all displayed score values using **formatScore**. Produce a summary **ViewModel** for end-of-round display.
- output: Return the summary **ViewModel**.

- exception: `InvalidScoreData` if *summary* is malformed or contains invalid scores.

`animateRoundTransition(from, to)`

- transition: Run client-side UI animation/transition from *from* to *to* (implementation-defined). Update `lastRendered` := *to*.
- output: `void`.
- exception: None.

18.4.5 Local Functions

None.

18.4.6 Considerations

- **Secret:** presentation choices for multi-base score displays and animations for round summaries.
- **Service:** shows standings after each round, presenting both decimal and dozenal scores in a clear, accessible format.
- This is a client-side UI module; it should not contain core scoring logic, only formatting and presentation. Numeric base conversion details are delegated to **M18**.

19 MIS of Profile View Module (M14)

19.1 Module

Profile View Module

19.2 Uses

- M9 Authentication Client Module
- [M17 Scoring Module](#)

19.3 Syntax

19.3.1 Exported Constants

None.

19.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
loadProfile	PlayerID	ProfileData	DatabaseReadError
updateProfile	PlayerID, Profile-Data	None	DatabaseWriteError
renderProfile	ProfileData	None	None

19.4 Semantics

19.4.1 State Variables

- **currentProfile**: cached profile data of the active player.
- **sessionStats**: recent gameplay summary for quick access.

19.4.2 Environment Variables

- Backend database connection or local storage; user interface display frame.

19.4.3 Assumptions

- Each player has a unique identifier.
- Profile data is fetched before rendering.

19.4.4 Access Routine Semantics

loadProfile(*PlayerID*)

- transition: retrieves stored profile data from database or cache.
- output: **ProfileData**.

updateProfile(*PlayerID*, *ProfileData*)

- transition: commits new statistics or preferences to persistent storage.
- output: **None**.

renderProfile(*ProfileData*)

- transition: displays the user's avatar, username, and score summary.
- output: **None**.

19.4.5 Local Functions

- **formatStats()**: formats match statistics for display.

19.4.6 Considerations

- This module must protect user data integrity and minimize latency when loading or updating profile information.

20 MIS of Game Engine Module (M15)

20.1 Module

Game Engine

20.2 Uses

- M16 Rules Module
- M17 Scoring Module
- M19 Game Actions Module

20.3 Syntax

20.3.1 Exported Constants

None.

20.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
initializeGame	PlayerList	GameState	InvalidSetupException
processTurn	Action	GameState	InvalidActionException
checkWinCondition	GameState	Boolean	None
reshuffleDeck	None	None	EmptyDeckException

20.4 Semantics

20.4.1 State Variables

- **currentState**: current configuration of the game.
- **activePlayer**: player ID whose turn is in progress.
- **drawPile, discardPile**: sets of remaining and played cards.

20.4.2 Environment Variables

- Game state repository, player actions, and random seed generator.

20.4.3 Assumptions

- Each player performs one valid action per turn.
- Randomness is seeded for reproducibility.

20.4.4 Access Routine Semantics

initializeGame(*PlayerList*)

- transition: distributes cards, sets starting player, and creates the discard pile.
- output: **GameState**.

processTurn(*Action*)

- transition: validates and executes one action, then triggers scoring update.
- output: **GameState**.

checkWinCondition(*GameState*)

- output: evaluates if any player has no cards remaining.

reshuffleDeck()

- transition: moves discard cards back into draw pile and randomizes order.
- output: **None**.

20.4.5 Local Functions

- **advanceTurn()**: calculates next player index.

20.4.6 Considerations

- The module must maintain consistency across all players and prevent race conditions during state transitions.

21 MIS of Rules Module (M16)

21.1 Module

Rules

21.2 Uses

- M18 Base Conversion Module

21.3 Syntax

21.3.1 Exported Constants

- **DozenalBase**: nat := 12
- **DozenalTarget**: string := “10” (base-12)
- **DozenalTargetDecimal**: nat := 12 (= M18.dozenalToDecimal(**DozenalTarget**))
- **InitialHandSize**: nat := 5

21.3.2 Exported Access Programs

Name	In	Out	Exceptions
initRound	deck: Deck, players: seq of PlayerID	RoundState	InvalidDeck, InvalidPlayers
isPlayable	state: RoundState, playerID: PlayerID, card: Card	boolean	InvalidRoundState, InvalidPlayer, InvalidCard
getPlayableCards	state: RoundState, playerID: PlayerID	seq of Card	InvalidRoundState, InvalidPlayer
applyPlay	state: RoundState, playerID: PlayerID, card: Card, chosenSuit: Suit?	RoundState	IllegalMove, InvalidRoundState, InvalidPlayer, InvalidCard, InvalidSuitChoice
mustDraw	state: RoundState, playerID: PlayerID	boolean	InvalidRoundState, InvalidPlayer
applyDraw	state: RoundState, playerID: PlayerID	RoundState	EmptyDeck, InvalidRoundState, InvalidPlayer
isRoundOver	state: RoundState	boolean	InvalidRoundState

21.4 Semantics

21.4.1 State Variables

None. (This module is stateless; all game situation is carried in `RoundState`.)

21.4.2 Environment Variables

None.

21.4.3 Assumptions

- This module currently supports 1v1 gameplay (two players) for round initialization and legality checks.
- `RoundState` contains: `hands` (mapping from `PlayerID` to a sequence of `Card`), `drawPile`, `discardPile`, `topCard`, `currentTurn`, and an optional `forcedSuit`.
- A **wildcard** is the rank “10” and is always playable regardless of suit/rank/sum constraints.
- A **face card** does not participate in the dozenal-sum rule; it can only be played via match-suit or match-rank (unless it is a wildcard).
- The dozenal-sum rule target is base-12 “10” (i.e., decimal 12).

21.4.4 Access Routine Semantics

`initRound(deck, players)`

- transition: Validate that `players` contains exactly two distinct players and `deck` has enough cards. Deal **InitialHandSize** cards from `deck` to each player’s hand. Draw one card from the deck to initialize `discardPile` and set `topCard`. Set `forcedSuit` := None. Set `currentTurn` to the starting player (policy-defined).
- output: Return the initialized `RoundState`.
- exception: `InvalidPlayers` if `players` is not a valid 1v1 set. `InvalidDeck` if the deck lacks sufficient cards or is malformed.

`isPlayable(state, playerId, card)`

- transition: Validate `state`, `playerId`, and that `card` is in `state.hands[playerId]`. Let `t` := `state.topCard`.
- output: Return `true` iff the following rule holds:
 - If `card` is a wildcard (rank 10), then `true`.

- Else if $state.forcedSuit \neq \text{None}$, then **true** iff $card.suit$ equals $state.forcedSuit$.
- Else (no forced suit), **true** iff at least one is satisfied:
 - * match suit: $card.suit = t.suit$
 - * match rank: $card.rank = t.rank$
 - * dozenal-sum: $card$ and t are not face cards and $card.rank + t.rank = \mathbf{Dozenal-TargetDecimal}$
- exception: **InvalidRoundState** if $state$ is malformed. **InvalidPlayer** if $playerId$ is not in the round. **InvalidCard** if $card$ is malformed or not in the player's hand.

getPlayableCards($state, playerId$)

- transition: Validate $state$ and $playerId$. For each $card$ in $state.hands[playerId]$, include it in the result iff **isPlayable($state, playerId, card$)** is **true**.
- output: Return the sequence of playable cards (possibly empty).
- exception: **InvalidRoundState** if $state$ is malformed. **InvalidPlayer** if $playerId$ is not in the round.

applyPlay($state, playerId, card, chosenSuit$)

- transition: Validate $state$ and turn order (must be $playerId$'s turn). Require **is-Playable($state, playerId, card$) = true**. Remove $card$ from $state.hands[playerId]$ and push it onto $state.discardPile$; set $state.topCard := card$.
 - If $card$ is a wildcard (rank 10), then $chosenSuit$ must be provided; set $state.forcedSuit := chosenSuit$.
 - Else set $state.forcedSuit := \text{None}$.

Advance $state.currentTurn$ to the other player.

- output: Return the updated **RoundState**.
- exception: **IllegalMove** if the play is not legal under the rules (including playing out of turn). **InvalidSuitChoice** if $card$ is wildcard and $chosenSuit$ is missing/invalid. **InvalidRoundState/InvalidPlayer/InvalidCard** as applicable.

mustDraw($state, playerId$)

- transition: Validate $state$ and $playerId$. Compute $playables := \text{getPlayableCards}(\mathit{state}, \mathit{playerId})$.
- output: Return **true** iff $playables$ is empty.
- exception: **InvalidRoundState** if $state$ is malformed. **InvalidPlayer** if $playerId$ is not in the round.

applyDraw(*state*, *playerId*)

- transition: Validate *state* and turn order. Require **mustDraw(*state*, *playerId*) = true**. Pop the top card from *state.drawPile* and add it to *state.hands[playerId]*. (Whether the drawn card may be played immediately is handled by the caller/game flow policy; this module only applies the draw.)
- output: Return the updated RoundState.
- exception: **EmptyDeck** if *state.drawPile* is empty. **InvalidRoundState/InvalidPlayer** as applicable.

isRoundOver(*state*)

- transition: Validate *state*. Check whether any player's hand is empty.
- output: Return **true** iff $\exists p$ such that *state.hands[p]* is empty; otherwise **false**.
- exception: **InvalidRoundState** if *state* is malformed.

21.4.5 Local Functions

None.

21.4.6 Considerations

- **Secret:** exact move-validation criteria, including how matching ranks, suits, dozenal sums, and special cards (wildcard) are handled.
- **Service:** confirms whether a proposed play is legal and enumerates valid plays for a player based on the current round situation; also provides helpers to initialize a round and apply state transitions for play/draw.
- This module does **not** compute scoring; scoring is handled by a separate module.
- Dozenal interpretation is delegated to **M18**; this module uses the decimal target (**DozenalTargetDecimal**) corresponding to base-12 “10”.

22 MIS of Scoring Module (M17)

22.1 Module

Scoring

22.2 Uses

- [M18 Base Conversion Module](#)

22.3 Syntax

22.3.1 Exported Constants

- **FaceCardScore**: nat := 10

22.3.2 Exported Access Programs

Name	In	Out	Exceptions
computeRoundScore	winner: PlayerID, loser- Hand: seq of Card	string	InvalidHand

22.4 Semantics

22.4.1 State Variables

None.

22.4.2 Environment Variables

None.

22.4.3 Assumptions

- This module is called only when a round ends (i.e., the winner has emptied their hand).
- The round score equals the sum of the remaining cards in the loser's hand.
- Face cards (J/Q/K) are worth **FaceCardScore**.
- Rank 10 (including wildcard 10) is worth 10.
- Numeric cards are worth their rank value.
- The returned score is represented in dozenal (base-12) using **M18.decimalToDozenal**.

22.4.4 Access Routine Semantics

computeRoundScore(*winner*, *loserHand*)

- transition: Validate *loserHand*. Compute a decimal total:

$$total = \sum_{c \in loserHand} scoreValue(c)$$

where:

$$scoreValue(c) = \begin{cases} \text{FaceCardScore} & \text{if } c \text{ is a face card} \\ 10 & \text{if } rank(c) = 10 \\ rank(c) & \text{otherwise} \end{cases}$$

Convert the decimal *total* to dozenal by calling **M18.decimalToDozenal**(*total*).

- output: Return the dozenal score string for the round winner.
- exception: **InvalidHand** if any card in *loserHand* is malformed or has an unsupported rank.

22.4.5 Local Functions

None.

22.4.6 Considerations

- **Secret:** the scoring equation that converts remaining cards into round points.
- **Service:** calculates the round score at the end of a round and returns it in dozenal form for display.

23 MIS of Base Conversion Module (M18)

23.1 Module

Base Conversion

23.2 Uses

None.

23.3 Syntax

23.3.1 Exported Constants

- **DozenalBase**: nat := 12
- **DecimalBase**: nat := 10

23.3.2 Exported Access Programs

Name	In	Out	Exceptions
decimalToDozenal	n: int	string	InvalidNumberFormatException
dozenalToDecimal	s: string	int	InvalidNumberFormatException
normalizeDozenal	s: string	string	InvalidNumberFormatException
isValidDozenal	s: string	boolean	None

23.4 Semantics

23.4.1 State Variables

None.

23.4.2 Environment Variables

None.

23.4.3 Assumptions

- The dozenal representation uses a fixed, consistent mapping between values and symbols for digits (including the two extra digits beyond 0–9).
- Input strings for conversion do not contain whitespace unless explicitly handled by `normalizeDozenal`.

23.4.4 Access Routine Semantics

decimalToDozenal(*n*)

- transition: Convert integer *n* from base **DecimalBase** to base **DozenalBase** using repeated division and remainder; map each remainder to the corresponding dozenal digit symbol; produce the resulting string (including a sign if *n* < 0).
- output: Return a dozenal **string** representing *n*.
- exception: **InvalidNumberFormat** if *n* is not a valid finite integer (implementation-defined; e.g., NaN or non-integer input in a loosely-typed context).

dozenalToDecimal(*s*)

- transition: Normalize *s* (optional). Validate that every character is a recognized dozenal digit symbol (and optional leading sign). Compute the decimal value by positional evaluation in base **DozenalBase**.
- output: Return the decimal **int** value represented by *s*.
- exception: **InvalidNumberFormat** if *s* contains invalid digits/symbols or is otherwise malformed.

normalizeDozenal(*s*)

- transition: Standardize *s* into canonical dozenal format (e.g., trim whitespace, normalize casing, map alternate symbols to the project-standard digit symbols).
- output: Return the normalized dozenal **string**.
- exception: **InvalidNumberFormat** if *s* cannot be normalized into a valid dozenal representation.

isValidDozenal(*s*)

- transition: Check whether *s* is a syntactically valid dozenal string under the module's digit-symbol mapping.
- output: Return **true** iff *s* is valid; otherwise return **false**.
- exception: None.

23.4.5 Local Functions

None.

23.4.6 Considerations

- **Secret:** the mapping of digits and symbols used to move between decimal and dozenal numbers.
- **Service:** translates numeric values to and from dozenal form for scoring logic and UI presentation.
- This is a behaviour-hiding utility module: callers rely on the conversion API without knowing the digit-symbol mapping details.

24 MIS of Game Actions Module (M19)

24.1 Module

Game Actions Module

24.2 Uses

- M15 Game Engine Module
- M16 Rules Module

24.3 Syntax

24.3.1 Exported Constants

None.

24.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
createAction	ActionType, Parameters	Action	InvalidActionType
validateAction	Action, GameState	Boolean	InvalidMoveException
executeAction	Action, GameState	GameState	ActionExecutionError
undoAction	Action, GameState	GameState	None

24.4 Semantics

24.4.1 State Variables

- **pendingActions**: a queue of unexecuted player actions.
- **lastAction**: most recent action for rollback or replay.

24.4.2 Environment Variables

- Backend execution environment.

24.4.3 Assumptions

- Each action follows the command pattern and can be validated independently.
- The game engine (M15) ensures single-threaded execution for action safety.

24.4.4 Access Routine Semantics

createAction(*ActionType, Parameters*)

- output: constructs an action object from parameters (e.g., “play card 8 spade”).

validateAction(*Action, GameState*)

- output: checks if the action is allowed under current rules (by calling M16).

executeAction(*Action, GameState*)

- transition: applies changes to game state (by calling M15) and notifies observers.
- output: **GameState**.

undoAction(*Action, GameState*)

- transition: reverses the last applied change for testing or debugging.
- output: **GameState**.

24.4.5 Local Functions

- **serializeAction()**: converts an action into a string or JSON for replay logging.

24.4.6 Considerations

- This module improves maintainability by isolating gameplay logic into self-contained actions, enabling undo/redo and deterministic testing.

25 MIS of Operating System Module (M20)

25.1 Module

Operating System Module

25.2 Uses

None.

25.3 Syntax

25.3.1 Exported Constants

None.

25.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
scheduleTask	Function, Delay	None	None
readFile	Path	String	IOError
writeFile	Path, String	None	IOError

25.4 Semantics

25.4.1 State Variables

- System scheduler queue.

25.4.2 Environment Variables

- Host OS file system and process manager.

25.4.3 Assumptions

- OS provides basic thread safety and asynchronous task execution.

25.4.4 Access Routine Semantics

scheduleTask(*Function, Delay*)

- transition: executes a callback after a specified delay.
- output: None.

readFile(*Path*)

- output: retrieves content from a local file path.

writeFile(*Path*, *String*)

- transition: writes content to a file.
- output: **None**.

25.4.5 Local Functions

- **validatePath(*path*):** checks file system accessibility.

25.4.6 Considerations

- This module ensures portability across Windows, macOS, and Linux.

26 MIS of Browser Runtime Module (M21)

26.1 Module

Browser Runtime Module

26.2 Uses

None.

26.3 Syntax

26.3.1 Exported Constants

None.

26.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
registerEvent	String, Callback	None	None
renderElement	HTMLElement	None	None
storeLocalData	Key, Value	None	StorageException

26.4 Semantics

26.4.1 State Variables

- Local storage cache, active event listeners.

26.4.2 Environment Variables

- Browser environment (HTML5, Web APIs).

26.4.3 Assumptions

- All browser APIs are available in the execution environment.

26.4.4 Access Routine Semantics

registerEvent(*String, Callback*)

- transition: binds a function to a specified DOM event.
- output: None.

renderElement(*HTMLElement*)

- transition: draws a UI element on the screen.
- output: None.

storeLocalData(*Key*, *Value*)

- transition: writes data into browser storage for persistence.
- output: None.

26.4.5 Local Functions

- **serialize(*obj*)**: converts objects into storable string format.

26.4.6 Considerations

- Must be compatible with modern browsers and responsive frameworks.

27 MIS of Database Module (M22)

27.1 Module

Database Module

27.2 Uses

None.

27.3 Syntax

27.3.1 Exported Constants

None.

27.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
connect	ConnectionString	Boolean	DatabaseConnectionError
query	SQLStatement	ResultSet	QueryError
insertRecord	Table, Data	Boolean	InsertError
updateRecord	Table, Data	Boolean	UpdateError

27.4 Semantics

27.4.1 State Variables

- **dbConnection:** current active database session.
- **cache:** optional in-memory data cache.

27.4.2 Environment Variables

- Database server or local SQLite environment.

27.4.3 Assumptions

- Database connection string is valid and accessible.

27.4.4 Access Routine Semantics

connect(*ConnectionString*)

- transition: establishes a session with the database server.
- output: Boolean.

query(*SQLStatement*)

- output: executes read operations and returns result sets.

insertRecord(*Table, Data*)

- transition: inserts a new entry into the specified table.
- output: Boolean.

updateRecord(*Table, Data*)

- transition: modifies existing records based on key values.
- output: Boolean.

27.4.5 Local Functions

- **sanitizeInput()**: prevents SQL injection attacks.

27.4.6 Considerations

- Database operations must remain atomic and logged to ensure integrity and traceability.

28 Appendix

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), if any, needed to be changed, and why?
5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)
6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)