Module Interface Specification for The Crazy Tens

Team #25, The Crazy Four
Ruida Chen
Ammar Sharbat
Alvin Qian
Jiaming Li

January 19, 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
Jan 17th	Rev0	update m1,4,5,6,9,13,16,17,18
Jan 18th	Rev0	update the remaining part

2 Symbols, Abbreviations and Acronyms

See MG Documentation at MG

Contents

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 $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

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 (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)

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

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: AuthenticatedRequest	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

- M5 Repository Module
- M6 Audit Module
- M15 Game Engine 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, gameID: GameID	void	SessionError, NotFound
handleSubmitAction	socket: ClientSocket, action: Action	void	InvalidAction, SessionError
registerGameSession	gameID: GameID, state: GameState	void	SessionError
emitGameState	socket: ClientSocket, state: GameState	void	
broadcastGameState	gameID: GameID, state: GameState	void	

7.4 Semantics

7.4.1 State Variables

- **activeSessions**: Map¡GameID, GameState¿ — Stores the current authoritative game state for each active game session.

7.4.2 Environment Variables

- **RealtimeAdapter**: The WebSocket or socket-based runtime environment managing client connections.
- **ClientSocket**: An opaque reference representing a connected client.

7.4.3 Assumptions

- Client connections are established through the **RealtimeAdapter**.
- Game state transitions are processed exclusively through M15.
- Persistence and logging services (M5, M6) are available.

7.4.4 Access Routine Semantics

handleConnection(*socket*)

- transition: Registers the *socket* with the **RealtimeAdapter** and initializes session metadata.
- output: `void`.
- exception: `SessionError` if the connection cannot be established.

handleJoinGame(*socket, gameID*)

- transition:
  - Looks up the game state from **activeSessions**.
  - Associates the *socket* with the specified *gameID*.
  - Logs the join event using M6.
- output: Sends the current `GameState` to the client via `emitGameState`.
- exception: `NotFound` if the game session does not exist.

handleSubmitAction(*socket, action*)

- transition:
  - Retrieves the associated `GameState` from **activeSessions**.
  - Calls `M15.processTurn(action)` to validate and apply the action.
  - Updates the stored `GameState` with the returned result.
  - Persists significant state changes using M5.
  - Logs the action execution using M6.

- output: Broadcasts the updated `GameState` to all connected players.
- exception: `InvalidAction` if the action is rejected by M15.

registerGameSession(*gameID, state*)

- transition: Inserts a new entry into **activeSessions**.
- output: `void`.
- exception: `SessionError` if the gameID already exists.

7.4.5 Local Functions

None.

7.4.6 Considerations

- **Secret:** the mapping between client sockets and authoritative game sessions.
- This module is server-authoritative but delegates all rule enforcement and state transitions to M15.
- Networking concerns are isolated here to avoid coupling game logic with transport mechanisms.

8 MIS of Matchmaking Module (M3)

8.1 Module

Matchmaking

8.2 Uses

- M2 Real-time Gateway Module
- M5 Repository Module

8.3 Syntax

8.3.1 Exported Constants

None.

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
createLobby	hostID: UserID	LobbyID	LobbyError
joinLobby	lobbyID: LobbyID, userID: UserID	void	LobbyFull, LobbyNotFound
startMatch	lobbyID: LobbyID, hostID: UserID	GameID	LobbyNotFound, NotLobbyHost

8.4 Semantics

8.4.1 State Variables

- **lobbies**: Map¡LobbyID, Lobby¿ — Stores all active waiting lobbies.

8.4.2 Environment Variables

None.

8.4.3 Assumptions

- User identities are authenticated before invoking this module.
- M2 is available to register real-time game sessions.
- Persistent storage (M5) is available.

8.4.4 Access Routine Semantics

createLobby(*hostID*)

- transition:
  - Generates a unique `LobbyID`.
  - Creates a `Lobby` object containing `hostID` and initial player list.
  - Stores the lobby in **lobbies**.
  - Persists lobby metadata via M5.
- output: Returns the created `LobbyID`.
- exception: `LobbyError` if creation fails.

joinLobby(*lobbyID, userID*)

- transition:
  - Retrieves the lobby from **lobbies**.
  - Verifies lobby capacity and status.
  - Adds *userID* to the lobby player list.
  - Updates lobby state via M5.
- output: `void`.
- exception:
  - `LobbyFull` if maximum players reached.
  - `LobbyNotFound` if the lobby does not exist.

startMatch(*lobbyID, hostID*)

- transition:
  - Retrieves the lobby from **lobbies**.
  - Verifies *hostID* matches the lobby host.
  - Generates a `GameID`.
  - Calls `M2.registerGameSession(GameID, lobby.players)`.
  - Removes the lobby from **lobbies**.
  - Persists match metadata via M5.
- output: Returns the generated `GameID`.
- exception:
  - `LobbyNotFound` if the lobby does not exist.
  - `NotLobbyHost` if the caller is not the lobby host.

8.4.5 Local Functions

None.

8.4.6 Considerations

- **Secret:** lobby lifecycle management and session handoff logic.
- This module deliberately avoids any game rule or state logic.
- All real-time behavior is delegated to M2 after match creation.

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.
- **DBConnectionString**: A secure configuration value used to initialize database connectivity.

10.4.3 Assumptions

- **PostgreSQLServer** is running and reachable from the server-side runtime.
- **DBConnectionString** 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 $\leq$ **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	serverURL: string	void	ConnectionFailed
disconnect	–	void	–
emit	event: string, payload: Data	void	ConnectionFailed
on	event: string, handler: Callback	void	–

12.4 Semantics

12.4.1 State Variables

- **socket**: ClientSocket — Active real-time connection handle.
- **connected**: boolean — Indicates connection status.

12.4.2 Environment Variables

- **ClientRuntime**: Execution environment providing networking support (e.g., WebSocket).

12.4.3 Assumptions

- M2 is reachable at the provided server URL.
- Network connectivity may be intermittent.

12.4.4 Access Routine Semantics

connect(*serverURL*)

- transition:
  - Establishes a real-time connection to M2.
  - Initializes **socket**.
  - Sets **connected** := true on success.
- output: `void`.
- exception: `ConnectionFailed` if the server is unreachable.

disconnect()

- transition:
  - Terminates the active connection.
  - Sets **connected** := false.
  - Releases **socket**.
- output: `void`.

emit(*event, payload*)

- transition:
  - Serializes *payload*.
  - Sends the event and payload to M2 through **socket**.
- output: `void`.
- exception: `ConnectionFailed` if **connected** is false.

on(*event, handler*)

- transition:
  - Registers *handler* to be invoked when M2 emits *event*.
- output: `void`.

12.4.5 Local Functions

None.

12.4.6 Considerations

- **Secret:** connection lifecycle handling and event dispatch mechanism.
- This module abstracts real-time communication details from higher-level client modules.
- No game logic or UI logic is included in this module.

13 MIS of Application Shell Module (M8)

13.1 Module

Application Shell

13.2 Uses

- M9 Authentication Client 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 currently authenticated user (if any).
- **currentRoute**: string — The active route derived from the browser URL.

13.4.2 Environment Variables

- **BrowserRuntime**: The client browser environment that provides the DOM and URL/History APIs.

13.4.3 Assumptions

- Authentication state (logged in vs. guest vs. logged out) is obtained exclusively through M9.
- Route selection and access gating are based on **currentUser** and **currentRoute**.

13.4.4 Access Routine Semantics

Render(*props*)

- transition:
  - Reads the current URL path from **BrowserRuntime** and updates **currentRoute**.
  - Uses M9 (e.g., token presence / auth state) to update **currentUser**.
  - Selects which page/view to render based on **currentRoute** and **currentUser** (e.g., unauthenticated users are redirected to the login route).
- output: Returns a `JSX.Element` that renders the global layout (e.g., header/footer) and the active page contents.
- exception: `RouteNotFound` if **currentRoute** does not match any entry in the routing table.

13.4.5 Local Functions

None.

13.4.6 Considerations

- **Secret:** the application routing table and the layout composition strategy.
- **Service:** provides the top-level structure for the client application and enforces authentication gating by relying only on M9 for auth state.
- This module contains no gameplay logic; it only coordinates navigation and top-level rendering.

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` $\cup$ {`None`}.

15 MIS of Lobby View Module (M10)

15.1 Module

Lobby View

15.2 Uses

- M1 API Module
- M7 Real-time Client Module

15.3 Syntax

15.3.1 Exported Constants

None.

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ViewProps	ViewModel	None
handleCreateLobby	token: SessionToken	LobbyID	CreateLobbyError, NetworkError
handleJoinLobby	token: SessionToken, lobbyID: LobbyID	void	JoinLobbyError, NetworkError
handleStartMatch	token: SessionToken, lobbyID: LobbyID	GameID	StartMatchError, NetworkError
subscribeLobbyUpdates	lobbyID: LobbyID	void	ConnectionFailed

15.4 Semantics

15.4.1 State Variables

- **availableLobbies**: seq of LobbySummary — List of lobbies shown to the user.
- **selectedLobby**: LobbyID ∪ {None} — Currently selected lobby.
- **statusMessage**: string — UI feedback text (errors/loading).
- **isLoading**: boolean — True while an operation is in progress.

15.4.2 Environment Variables

- **ClientRuntime**: UI runtime responsible for rendering view models and handling user input events.

15.4.3 Assumptions

- The user is authenticated and a valid `SessionToken` is available when creating/joining/starting a match.
- M1 provides HTTP endpoints for lobby and match operations.
- M7 is able to connect to the real-time server and subscribe to lobby/game updates.

15.4.4 Access Routine Semantics

Render(*props*)

- transition: Updates the displayed lobby list and user interaction controls using current state variables.
- output: Returns a `ViewModel` representing the lobby UI (list of lobbies, create/join/start controls, status messages).
- exception: None.

handleCreateLobby(*token*)

- transition:
  - Sets **isLoading** := true.
  - Calls M1 to request lobby creation using *token*.
  - On success, sets **selectedLobby** to the returned `LobbyID`.
- output: Returns the created `LobbyID`.
- exception: `CreateLobbyError` if the server rejects the request; `NetworkError` if the request fails.

handleJoinLobby(*token, lobbyID*)

- transition:
  - Sets **isLoading** := true.
  - Calls M1 to request joining *lobbyID* using *token*.
  - On success, calls **subscribeLobbyUpdates**(*lobbyID*) to listen for real-time status changes.

- output: `void`.
- exception: `JoinLobbyError` if lobby is full/not found/rejected; `NetworkError` if request fails.

handleStartMatch(*token, lobbyID*)

- transition:
  - Calls M1 to start a match for *lobbyID*.
  - Receives a `GameID`.
  - Uses M7 to subscribe to real-time updates for the created game session (implementation-defined event name).
- output: Returns `GameID`.
- exception: `StartMatchError` if the server rejects start; `NetworkError` if request fails.

subscribeLobbyUpdates(*lobbyID*)

- transition:
  - Ensures M7 is connected (calls `M7.connect` if required).
  - Registers event handlers via `M7.on` to receive lobby status updates and update **availableLobbies**/**statusMessage**.
- output: `void`.
- exception: `ConnectionFailed` if M7 cannot connect.

15.4.5 Local Functions

None.

15.4.6 Considerations

- **Secret:** UI-level flow for lobby creation/joining/starting and how lobby updates are presented.
- This module performs no game logic; it only coordinates HTTP calls (M1) and real-time subscriptions (M7).

16 MIS of Game Board View Module (M11)

16.1 Module

Game Board View

16.2 Uses

- M12 Move Controller Module
- M13 Scoreboard View Module

16.3 Syntax

16.3.1 Exported Constants

None.

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	state: RoundState, playerId: PlayerID, scores: ScoreboardData	ViewModel	None
onSelectCard	card: Card	void	None
onPlaySelected	chosenSuit: Suit?	void	InvalidMoveUI, ConnectionFailed
onDraw	–	void	InvalidMoveUI, ConnectionFailed

16.4 Semantics

16.4.1 State Variables

- **selectedCard**: Card ∪ {None} — The card currently selected by the user.
- **validMoves**: seq of Card — Cached playable cards for highlighting.
- **statusMessage**: string — UI message for user feedback (errors/info).

16.4.2 Environment Variables

- **ClientRuntime**: UI runtime that renders the view model and delivers user input events.

16.4.3 Assumptions

- This module is a client-side view and does not modify authoritative game state.
- The current `RoundState` and player identity are provided by the client application flow.
- M12 is available for validating and emitting moves; M13 is available for formatting/displaying scores.

16.4.4 Access Routine Semantics

Render(*state, playerId, scores*)

- transition:
  - Calls `M12.getClientValidMoves(state, playerId)` to update **validMoves**.
  - Calls `M13.renderScoreboard(scores)` to obtain a scoreboard view component.
  - Constructs a `ViewModel` that displays: player hand, discard pile/top card, draw control, turn/status indicators, and the rendered scoreboard.
  - Highlights cards that are in **validMoves**.
- output: Returns the constructed `ViewModel`.
- exception: None. (Errors are represented via **statusMessage** when possible.)

onSelectCard(*card*)

- transition:
  - Sets **selectedCard** := *card*.
  - Clears or updates **statusMessage**.
- output: `void`.

onPlaySelected(*chosenSuit*)

- transition:
  - Requires **selectedCard** $\neq$ None.
  - Calls `M12.handlePlayCard(state, playerId, selectedCard, chosenSuit)`.
  - On failure, updates **statusMessage** with the error context.
- output: `void`.
- exception: `InvalidMoveUI`, `ConnectionFailed`.

onDraw()

- transition:
  - Calls `M12.handleDrawCard(state, playerId)`.
  - On failure, updates **statusMessage**.
- output: `void`.
- exception: `InvalidMoveUI`, `ConnectionFailed`.

16.4.5 Local Functions

None.

16.4.6 Considerations

- **Secret:** the UI layout for presenting the round state and mapping user interactions to controller calls.
- M11 is a presentation module; all move validation and move submission are delegated to M12.
- Score formatting and display are delegated to M13; M11 does not compute scores.

17 MIS of Move Controller Module (M12)

17.1 Module

Move Controller

17.2 Uses

- M7 Real-time Client Module
- M16 Rules Module

17.3 Syntax

17.3.1 Exported Constants

None.

17.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
handlePlayCard	state: RoundState, playerId: PlayerID, card: Card, chosenSuit: Suit?	void	InvalidMoveUI, ConnectionFailed
handleDrawCard	state: RoundState, playerId: PlayerID	void	InvalidMoveUI, ConnectionFailed
getClientValidMoves	state: RoundState, playerId: PlayerID	seq of Card	InvalidMoveUI

17.4 Semantics

17.4.1 State Variables

- **uiFeedback**: string — Latest user-facing feedback (error/info).

17.4.2 Environment Variables

- **ClientRuntime**: UI runtime that provides user input events and dispatches them to this controller.

17.4.3 Assumptions

- M7 is available and can emit actions to the server (M2).
- The provided `state` is the latest client-visible `RoundState`.
- M16 defines the authoritative rule checks used for client-side pre-validation.

17.4.4 Access Routine Semantics

getClientValidMoves(*state, playerId*)

- transition:
  - Calls `M16.getPlayableCards(state, playerId)`.
- output: Returns the sequence of playable cards for *playerId*.
- exception: `InvalidMoveUI` if the *state* or *playerId* is invalid for rule evaluation.

handlePlayCard(*state, playerId, card, chosenSuit*)

- transition:
  - Performs client-side pre-validation by checking `M16.isPlayable(state, playerId, card)`.
  - If invalid, sets **uiFeedback** and raises `InvalidMoveUI`.
  - If valid, emits a play action to the server via:

    ```
    M7.emit('submitMove', {action: 'play', card: card, chosenSuit: chosenSuit})
    ```

- output: `void`.
- exception:
  - `InvalidMoveUI` if pre-validation fails.
  - `ConnectionFailed` if M7 is not connected.

handleDrawCard(*state, playerId*)

- transition:
  - Performs client-side pre-validation by checking `M16.mustDraw(state, playerId)`.
  - If invalid, sets **uiFeedback** and raises `InvalidMoveUI`.
  - If valid, emits a draw action to the server via:

    ```
    M7.emit('submitMove', {action: 'draw'})
    ```

- output: `void`.
- exception:
  - `InvalidMoveUI` if drawing is not permitted.
  - `ConnectionFailed` if M7 is not connected.

17.4.5 Local Functions

None.

17.4.6 Considerations

- **Secret:** mapping user inputs (clicks, selections) into standardized move events sent to the server.
- Client-side validation is only for responsiveness; the server remains authoritative.
- This module contains no rendering logic and no game-state mutation; it only validates and emits actions.

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

19.2 Uses

- M1 API Module

19.3 Syntax

19.3.1 Exported Constants

None.

19.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ReactProps	JSX.Element	None
loadProfile	None	void	NetworkError, Unauthorized
updateProfile	data: ProfileUpdateData	void	NetworkError, InvalidInput, Unauthorized

19.4 Semantics

19.4.1 State Variables

- **profileData**: ProfileData — Cached profile information of the current user.
- **isLoading**: boolean — Indicates whether a profile request is in progress.
- **errorMessage**: string — None — Error message displayed in the UI.

19.4.2 Environment Variables

- **BrowserRuntime (M21)**: Provides DOM rendering and user interaction events.

19.4.3 Assumptions

- The user is authenticated and a valid session token is available on the client.
- Profile data is accessed exclusively through M1; this module does not access persistent storage directly.

19.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads **profileData**, **isLoading**, and **errorMessage** from state.
- output: Returns a `JSX.Element` that renders the user's profile information (e.g., username, avatar, statistics summary) and profile-editing controls.

loadProfile()

- transition: Sets **isLoading** to `true`. Sends an authenticated request to `M1.getProfile`. On success, stores the returned data in **profileData**. Clears **errorMessage**.
- output: `void`.
- exception: `Unauthorized` if the session token is invalid or expired. `NetworkError` if the API request fails.

updateProfile(*data*)

- transition: Validates *data* on the client. Sends an authenticated request to `M1.putProfile` with updated profile fields. On success, updates **profileData**.
- output: `void`.
- exception: `InvalidInput` if client-side validation fails. `Unauthorized` if the session token is invalid. `NetworkError` if the API request fails.

19.4.5 Local Functions

None.

19.4.6 Considerations

- **Secret:** UI layout decisions and client-side state handling for profile rendering and editing.
- **Service:** provides a user interface for viewing and updating profile information by delegating all data access to M1.
- This module contains no business logic and no persistence logic; it is a pure presentation and interaction layer.

20 MIS of Game Engine Module (M15)

20.1 Module

Game Engine

20.2 Uses

- M16 Rules Module
- M17 Scoring Module

20.3 Syntax

20.3.1 Exported Constants

- **MaxPlayers**: nat := 2

20.3.2 Exported Access Programs

Name	In	Out	Exceptions
createGame	players: seq of PlayerID, options: GameOptions	state: GameState	InvalidSetup
getState	None	GameState	NotInitialized
applyTurn	action: Action	GameState	InvalidAction, NotYourTurn, GameOver
isGameOver	None	boolean	NotInitialized
getWinner	None	PlayerID	NotInitialized, GameNotOver

20.4 Semantics

20.4.1 State Variables

- **gameState**: GameState (current authoritative game state)
- **isInitialized**: boolean := `false`

20.4.2 Environment Variables

- **RNG**: Source of randomness used for shuffling and dealing (implementation-defined).

20.4.3 Assumptions

- This module is server-side and acts as the source of truth for game state.
- The rules for legal moves and round transitions are provided by M16.
- Round scoring at the end of a round is provided by M17.
- A game is initialized with exactly **MaxPlayers** players.

20.4.4 Access Routine Semantics

createGame(*players, options*)

- transition: Validate *players* size and uniqueness. Initialize a fresh deck and shuffle using **RNG**. Call `M16.initRound(deck, players)` to create the initial round state. Initialize scores for players (0 in decimal, stored as project-defined representation). Set **gameState** to the newly created state and set **isInitialized** := `true`.
- output: Return the initialized `GameState`.
- exception: `InvalidSetup` if *players* is invalid or initialization fails.

getState()

- output: Return **gameState**.
- exception: `NotInitialized` if **isInitialized** is `false`.

applyTurn(*action*)

- transition:

  1. Require **isInitialized** = `true`.
  2. Check turn ownership from **gameState** (implementation-defined) and reject if *action.playerId* is not the active player.
  3. If *action* is a play-card action, call `M16.applyPlay` to update the round state.
  4. If *action* is a draw-card action, call `M16.applyDraw` to update the round state.
  5. If `M16.isRoundOver(updatedRoundState)` is `true`, compute round score via `M17.computeRound loserHand)` and update cumulative scores in **gameState**. Re-initialize the next round (policy-defined) using `M16.initRound`.
  6. If the overall game end condition is satisfied (policy-defined), mark game over and store winner.

  Update **gameState** with the new round state, scores, and turn.

- output: Return the updated `GameState`.

- exception: `NotYourTurn` if *action.playerId* is not the active player. `InvalidAction` if the action is malformed or illegal under M16. `GameOver` if the game has already ended.

isGameOver()

- output: Return `true` iff **gameState** is marked as game-over; otherwise `false`.
- exception: `NotInitialized` if **isInitialized** is `false`.

getWinner()

- output: Return the stored winner `PlayerID`.
- exception: `NotInitialized` if **isInitialized** is `false`. `GameNotOver` if the game is not over yet.

20.4.5 Local Functions

None.

20.4.6 Considerations

- **Secret:** orchestration of round lifecycle, turn sequencing, and how cumulative score/state is stored in `GameState`.
- **Service:** maintains the authoritative state machine for Crazy Eights gameplay while delegating legality checks to M16 and scoring computation to M17.
- This module should remain deterministic given the same shuffled deck seed and action sequence to support testing and debugging.

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* $\neq$ 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* = **DozenalTargetDecimal**

- 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 **isPlayable**(*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* := 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* := **getPlayableCards**(*state, 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, loserHand: 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:

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

  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	InvalidNumberFormat
dozenalToDecimal	s: string	int	InvalidNumberFormat
normalizeDozenal	s: string	string	InvalidNumberFormat
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 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), it 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)