Contents

[High-Level Design Document 1](#_Toc182572953)

[1. Document Overview 1](#_Toc182572954)

[2. Project Overview 2](#_Toc182572955)

[Objective 2](#_Toc182572956)

[Scope 2](#_Toc182572957)

[3. System Architecture 2](#_Toc182572958)

[3.1 Architecture Diagram 2](#_Toc182572959)

[3.2 Component Breakdown 3](#_Toc182572960)

[5. Key Technologies 4](#_Toc182572961)

[6. Database Design 4](#_Toc182572962)

[7. API Specifications 5](#_Toc182572963)

[8. Security 6](#_Toc182572964)

[9. Monitoring & Logging 6](#_Toc182572965)

[10. Deployment Strategy 6](#_Toc182572966)

[11. Assumptions & Constraints 6](#_Toc182572967)

[12. User Stories and Tasks 6](#_Toc182572968)

[13. OpenAPI Specification 8](#_Toc182572969)

# High-Level Design Document

Here's a High-Level Design (HLD) document outline for the Player Match Management system, which leverages Azure Functions, Cosmos DB, SQL Server, and a .NET Web API to manage player and match data. This HLD includes system architecture, key components, data flow, and integration points, structured to meet the requirements you've specified.

## 1. Document Overview

- Title: High-Level Design for Player Match Management System

- Date: [15-11-2024]

- Version: 1.0

- Author: [Sankar B]

## 2. Project Overview

### Objective

The Player Match Management System is designed to store, update, and retrieve data for players and matches. This system involves:

- Storing player information, including basic details and statistics (matches played, won, lost, and win/loss ratio).

- Capturing match information to keep player statistics dynamically updated.

- Exposing APIs to interact with this data.

### Scope

The system will consist of:

- Azure Functions for creating and managing match data, and updating player statistics.

- A .NET Web API to interface with the Azure Functions and expose endpoints for external access.

- Cosmos DB and SQL Server databases for data storage and management.

## 3. System Architecture

### 3.1 Architecture Diagram

The architecture includes the following components:

1. Frontend/Application Layer: An external application or client interacts with the system through the Web API.

2. Web API (ASP.NET Core): Acts as a middleware layer that:

- Sends requests to Azure Functions for match creation.

- Queries player data through Azure Functions.

3. Azure Functions:

- Function 1: Handles REST API requests to create a new match record in the SQL database.

- Function 2: Implements Change Data Capture (CDC) from the Matches table to dynamically update the player statistics.

- Function 3: Fetches player details by player ID from the SQL database.

4. Data Storage:

- SQL Server: Stores player and match data in two tables.

- Cosmos DB (optional if needed for high scalability or further expansion).

5. Monitoring and Logging: Azure Application Insights or Azure Monitor for capturing logs, errors, and system performance.

### 3.2 Component Breakdown

#### 1. .NET Web API:

- Provides an interface for external clients.

- Handles requests for creating a match and fetching player information.

- Acts as an API gateway to Azure Functions.

#### 2. Azure Functions:

- CreateMatchFunction: Adds new match data to SQL Server.

- UpdatePlayerStatsFunction: Triggers on match creation to update player stats.

- GetPlayerInfoFunction: Retrieves player details by ID, returning key statistics.

#### 3. SQL Server:

- Players Table: Holds player data such as player ID, name, gender, nationality, matches played, won, lost, and win/loss percentage.

- Matches Table: Contains match records with player IDs, match ID, nationality, and match result.

#### 4. Data Flow

##### 4.1 Sequence of Events

1. Creating a Match:

- Client sends a POST request to the Web API’s `/api/matches` endpoint.

- Web API calls the `CreateMatchFunction` Azure Function, which inserts match data into the SQL database.

- `UpdatePlayerStatsFunction` is triggered to update the player statistics.

2. Updating Player Statistics (CDC):

- `UpdatePlayerStatsFunction` listens for changes in the Matches table.

- When a new match is inserted, it retrieves the updated data and calculates the updated win/loss stats for the involved players.

- Updates the relevant player records in the Players table in SQL Server.

3. Fetching Player Information:

- Client sends a GET request to `/api/players/{playerId}`.

- Web API calls `GetPlayerInfoFunction`, which retrieves player details from SQL Server and returns them.

## 5. Key Technologies

1. Azure Functions: Serverless compute service for lightweight, scalable functions.

2. ASP.NET Core Web API: Used as a centralized access layer for the Azure Functions.

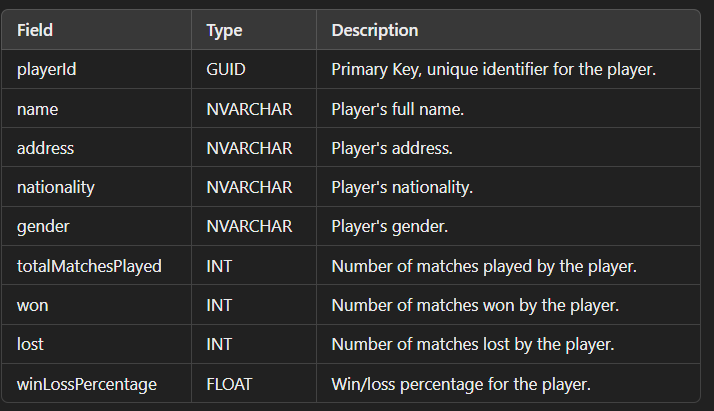
3. SQL Server: Manages structured player and match data.

4. Cosmos DB (optional): Provides globally distributed, multi-model database capabilities (useful if the application needs NoSQL support).

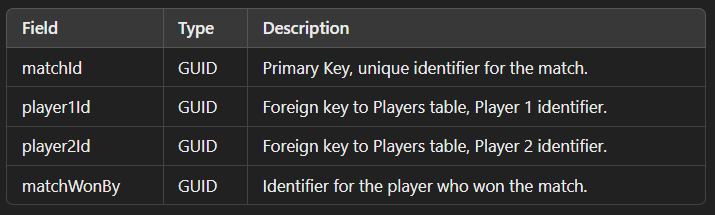
5. Application Insights: Monitors and tracks logs, errors, and performance.

## 6. Database Design

Players Table



Matches Table



## 7. API Specifications

Endpoints

1. POST `/api/matches`

- Creates a match record in SQL Server.

- Request Body: `Match` model with player IDs, names, nationalities, and winner ID.

- Response: Success or failure message.

2. GET `/api/players/{playerId}`

- Fetches player details.

- Response: Player data with fields like name, nationality, matches played, won, lost, and win/loss percentage.

## 8. Security

- Authentication: Use Azure AD or API keys to authenticate requests to the Web API.

- Authorization: Implement role-based access control (RBAC) if needed.

- Data Encryption: Use HTTPS for data transmission and encrypt sensitive data in SQL Server.

## 9. Monitoring & Logging

- Application Insights: Track function performance, API response times, and errors.

- Azure Monitor: Set alerts for unusual patterns like high error rates or response latency.

## 10. Deployment Strategy

- Environment: Set up separate environments (e.g., Development, Staging, Production) using Azure resource groups and configurations.

- Continuous Integration: Use GitHub Actions or Azure DevOps pipelines to automate build and deployment of the Azure Functions and Web API.

- Continuous Deployment: Set up release pipelines for each environment to deploy changes efficiently.

## 11. Assumptions & Constraints

- Assumes Azure Functions and SQL Server can handle the anticipated load.

- Constraints may include function execution time limits or rate limits in Azure.

## 12. User Stories and Tasks

**User Story 1:**

As a developer, I want to create an Azure Function to add a new match to the database, so I can update match records via a REST API.

- Task 1.1: Set up the Azure Function project.

- Task 1.2: Implement the `CreateMatch` function.

- Subtask 1.2.1: Define HTTP trigger with REST API endpoint.

- Subtask 1.2.2: Validate incoming request payload for match data.

- Subtask 1.2.3: Write logic to insert match data into the SQL database `Matches` table.

- Task 1.3: Test `CreateMatch` function with sample data to ensure data is added to the SQL database correctly.

**User Story 2:**

As a developer, I want to create an Azure Function that updates player statistics when a match is added or updated.

- Task 2.1: Configure the Azure Function for Change Data Capture (CDC).

- Subtask 2.1.1: Set up CDC on the SQL `Matches` table.

- Subtask 2.1.2: Create triggers or event listeners on `Matches` for changes in data.

- Task 2.2: Implement `UpdatePlayerStatistics` function.

- Subtask 2.2.1: Fetch relevant players based on the updated match data.

- Subtask 2.2.2: Calculate matches played, wins, and losses for each affected player.

- Subtask 2.2.3: Update player statistics in the SQL `Players` table.

- Task 2.3: Test with data updates in `Matches` and verify that statistics update correctly in `Players`.

**User Story 3:**

As a developer, I want a REST API endpoint to retrieve player information based on player ID.

- Task 3.1: Implement the `GetPlayerInfo` Azure Function.

- Subtask 3.1.1: Define HTTP trigger with REST API endpoint for retrieving player information.

- Subtask 3.1.2: Write logic to query the `Players` table using player ID.

- Subtask 3.1.3: Format and return player information as a JSON response.

- Task 3.2: Test the `GetPlayerInfo` endpoint with various player IDs to ensure accurate data retrieval.

**User Story 4:**

As a developer, I want a .NET Web API to call the Azure Functions to manage matches and players.

- Task 4.1: Create the .NET Web API project.

- Task 4.2: Implement HTTP client service to call Azure Function endpoints.

- Subtask 4.2.1: Set up HttpClient with configuration for Azure Function URLs.

- Subtask 4.2.2: Create methods in the service for `CreateMatch` and `GetPlayerInfo`.

- Task 4.3: Implement `MatchesController` with endpoints for creating a match.

- Task 4.4: Implement `PlayersController` with endpoint to retrieve player information.

- Task 4.5: Test Web API endpoints by calling Azure Functions and verifying the data flow.

**User Story 5:**

As a developer, I want to set up configuration and deployment scripts for local and cloud environments.

- Task 5.1: Configure local development settings.

- Subtask 5.1.1: Create `local.settings.json` for Azure Functions with local configuration.

- Subtask 5.1.2: Set up `launchSettings.json` for local debugging in Visual Studio.

- Task 5.2: Write deployment scripts to deploy Azure Functions and Web API.

- Subtask 5.2.1: Define GitHub Actions or Azure Pipelines for CI/CD deployment to Azure.

- Subtask 5.2.2: Configure Azure resources in the scripts, including Cosmos DB, SQL Database, and Azure Functions.

- Task 5.3: Test deployment process to confirm smooth CI/CD flow.

## 13. OpenAPI Specification

OpenAPI documentation outline for the .NET Web API project that interacts with the Azure Functions for managing player and match data. This covers the endpoints for creating a match and retrieving player information, including request/response models and parameter details.

Info Section



**Paths**

1. Create Match Endpoint

 2. Get Player Information Endpoint



Components

Schemas

 Example Requests and Responses

1. Create Match

- POST `/api/matches`

- Request Body:

*```json*

*{*

*"matchId": "guid-here",*

*"player1Id": "player1-guid-here",*

*"player1Name": "Player One",*

*"player1Nationality": "Country A",*

*"player2Id": "player2-guid-here",*

*"player2Name": "Player Two",*

*"player2Nationality": "Country B",*

*"matchWonBy": "player1-guid-here"*

*}*

- Response:

- 200 OK: `{"message": "Match created successfully."}`

- 500 Internal Server Error: `{"error": "Failed to create match."}`

**2. Get Player Information**

- GET `/api/players/{playerId}`

- Path Parameter: `playerId`: `player-guid-here`

- Response:

- 200 OK:

*```json*

*{*

*"playerId": "player-guid-here",*

*"name": "Player Name",*

*"address": "123 Example St.",*

*"nationality": "Country",*

*"gender": "Male",*

*"totalMatchesPlayed": 50,*

*"won": 30,*

*"lost": 20,*

*"winLossPercentage": 60.0*

*}*

*```*

- 404 Not Found: `{"error": "Player not found."}`