Real Time Sports Dashboard Using AWS AppSync and DynamoDB

A Course Project Report Submitted in partial fulfillment of the course requirements for the award of grades in the subject of

CLOUD BASED AIML SPECIALITY
(22SDCS07A)

by

Sai Vyseshika V

(2210030504)

Under the esteemed guidance of

Ms. P. Sree Lakshmi
Assistant Professor,
Department of Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

K L Deemed to be UNIVERSITY

Aziznagar, Moinabad, Hyderabad,
Telangana, Pincode: 500075

April 2025

K L Deemed to be UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Certificate

This is Certified that the project entitled **"Real Time Sports Dashboard Using AWS AppSync and DynamoDB"** which is a experimental &/ theoretical &/ Simulation&/ hardware work carried out by Sai Vyseshika V (2210030504), in partial fulfillment of the course requirements for the award of grades in the subject of **CLOUD BASED AIML SPECIALITY**, during the year **2024-2025**. The project has been approved as it satisfies the academic requirements.

Ms.P.Sree Lakshmi | **Dr. Arpita Gupta**

Course Coordinator | **Head of the Department**

Ms. P. Sree Lakshmi

Course Instructor

CONTENTS

1.INTRODUCTION

In today's fast-paced digital world, sports enthusiasts demand real-time updates on live matches, scores, and player performances. Traditional sports websites often rely on manual refreshes to display updated scores, causing delays in accessing real-time information. With the rapid advancement of cloud technologies, it has become possible to build a fully automated sports dashboard that delivers instant updates without requiring constant page refreshes.

This project aims to develop a real-time sports dashboard that fetches live cricket scores from an external API and displays them dynamically using AWS AppSync, DynamoDB, IAM roles, and JavaScript. The data is stored, processed, and updated automatically, allowing users to stay informed with the latest match progress through an interactive interface powered by Chart.js.[3]

The architecture follows a serverless model, meaning that the system efficiently handles real-time data flow with minimal maintenance. AWS AppSync provides GraphQL-based real-time data synchronization, DynamoDB acts as a scalable NoSQL database, and GraphQL subscriptions ensure that frontend clients receive automatic updates. The frontend is built with JavaScript and Chart.js, offering a user-friendly way to visualize live match scores in the form of dynamic pie charts.[1]

By leveraging AWS cloud services, this project demonstrates the power of serverless computing in modern real-time applications. The result is a highly scalable, cost-efficient, and real-time sports dashboard that can be extended to support multiple sports and additional features such as player statistics, match analysis, and push notifications.

2.AWS Services Used as part of the project

- **AWS AppSync**

  Provides a GraphQL API that facilitates real-time data synchronization between the database and frontend. It eliminates the need for backend servers and allows smooth integration with DynamoDB, making it an ideal choice for handling live sports data. Enables real-time updates through GraphQL subscriptions, reducing latency and improving user experience.[1]

- **AmazonDynamoDB**

  A fast and scalable NoSQL database that stores live match data. DynamoDB's ability to handle rapid data updates ensures that match scores are always current, providing seamless real-time functionality. [2] Offers high-speed read/write operations with automatic scaling, ensuring efficient data storage and retrieval.

- **IAM Roles and Policies**

  Used for granting necessary permissions to AppSync and DynamoDB without requiring an AWS Lambda function. IAM roles ensure secure access control, defining which AWS services can interact with each other.
  Enhances security by restricting permissions and reducing potential vulnerabilities.

- **Amazon API Gateway**

  Serves as a bridge between the external cricket API and AppSync. It helps manage incoming API requests, transforming them into a format suitable for GraphQL queries while ensuring secure access to live sports data. Acts as an intermediary, ensuring seamless API data retrieval and protecting backend services from excessive traffic.

.

3.Steps Involved in solving project problem statement

1.**Understanding Requirements**:

The first step was identifying the need for a real-time sports dashboard and defining key functionalities such as data fetching, storage, and visualization.

2**. Setting Up AWS AppSync:**

AWS AppSync is crucial for handling real-time data in your dashboard. The setup process includes:

- Creating an AppSync API: Using the AWS Management Console to create the GraphQL API for data handling. AppSync simplifies real-time data synchroniz -ation, offering support for queries, mutations, and subscriptions.

- Connecting AppSync to DynamoDB: AppSync integrates seamlessly with DynamoDB, allowing automatic scaling. Setting this up involves choosing DynamoDB as a data source, configuring it as a back-end for match data storage.[2]

- Ensuring Real-Time Updates: AppSync helps maintain real-time updates to the frontend by utilizing GraphQL subscriptions. Every time data is modified in DynamoDB, AppSync pushes updates to the client, ensuring live data streams without constant polling.

3. **Creating the GraphQL Schema:**

The GraphQL schema serves as the backbone of your API. It defines the structure of data and specifies how it can be queried or mutated. Key steps here include:

- Defining Data Types: Specify the data types for your objects, such as Match, Team,Score and PlayerStats.

- Defining Queries and Mutations: Queries retrieve data (e.g., fetching current match scores), while mutations modify data (e.g., updating a team’s score). The schema must support both.

- Defining Subscriptions: To allow real-time updates, you'll define a subscription that listens for changes (e.g., a new score update) and pushes it to the frontend.

4. Configuring Data Sources and Resolvers:

Resolvers are the critical component linking GraphQL operations to DynamoDB

- Setting up DynamoDB Table: Create a table in DynamoDB with appropriate partition and sort keys, such as MatchID for partition key and timestamp for sort key.

- Configuring Resolvers: Each operation in the schema (query, mutation, or subscription) is linked to a resolver. For example, a getMatchData query would resolve to fetching data from the DynamoDB table. Mutations would update the DynamoDB table based on incoming data.

- Optimizing Data Fetching: To ensure efficiency, consider implementing pagination for large data sets (e.g., for historical match data) and batch operations for bulk updates.

5. Setting Up IAM Roles:

- IAM Role Creation: Define IAM roles that grant AppSync permission to read/write data from/to DynamoDB securely. For example, you would create a policy like AppSyncDynamoDBAccessPolicy that dynamodb:Query, dynamodb:PutItem, and other relevant actions.

- Fine-Grained Access Control: You can set more granular permissions based on the operation (e.g., different roles for admins vs. regular users) or based on the data's nature (e.g., limiting access to scores of specific matches).

- Security Best Practices: Ensure the least privilege principle is followed by restricting permissions to only necessary actions.

6. Fetching Live Cricket Data:

This step requires integrating external sources to bring in live data:

- API Integration: Integrating an external API, such as a cricket match data provider like ESPN or CricAPI, allows the system to fetch live match data.

- Data Parsing and Transformation: The fetched data needs to be parsed and transformed into a format that fits the GraphQL schema. For example, converting match score data into a format compatible with the Match and Teams types in the GraphQL schema.

- Error Handling and Fallbacks: Consider implementing error handling for failed data fetches, such as retries or fallback mechanisms in case the API is down or fails to return data.

7. Developing the Frontend:

Building a user-friendly frontend is crucial for displaying real-time data effectively:

- Frontend Framework: Since you're using JavaScript, a simple, lightweight frontend using vanilla JavaScript or frameworks like Vue.js (without React) can dynamically update based on GraphQL subscriptions.

- Visualizing Data: Use Chart.js to create dynamic and interactive visualizations, such as pie charts for team statistics, bar charts for score comparisons, and line charts for historical performance.

- Real-Time Data Binding: The frontend should automatically update whenever new data is received, without requiring a page refresh. This can be achieved using GraphQL subscriptions, where the frontend listens for changes and updates the UI in real-time.

8. Implementing GraphQL Subscriptions:

Subscriptions allow the frontend to automatically receive updates when the data changes:

- Setting Up Subscriptions: In the GraphQL schema, define a subscription for real-time updates on match data, such as a subscription that notifies the frontend of score changes.

- Handling Data in the Frontend: In the frontend, use Apollo Client (or a similar library) to manage the connection to the AppSync API. This will handle incoming data from subscriptions and update the UI accordingly.

- Testing Subscriptions: Ensure that subscriptions work as expected by testing with various match updates and verifying that the frontend receives real-time updates without delay.

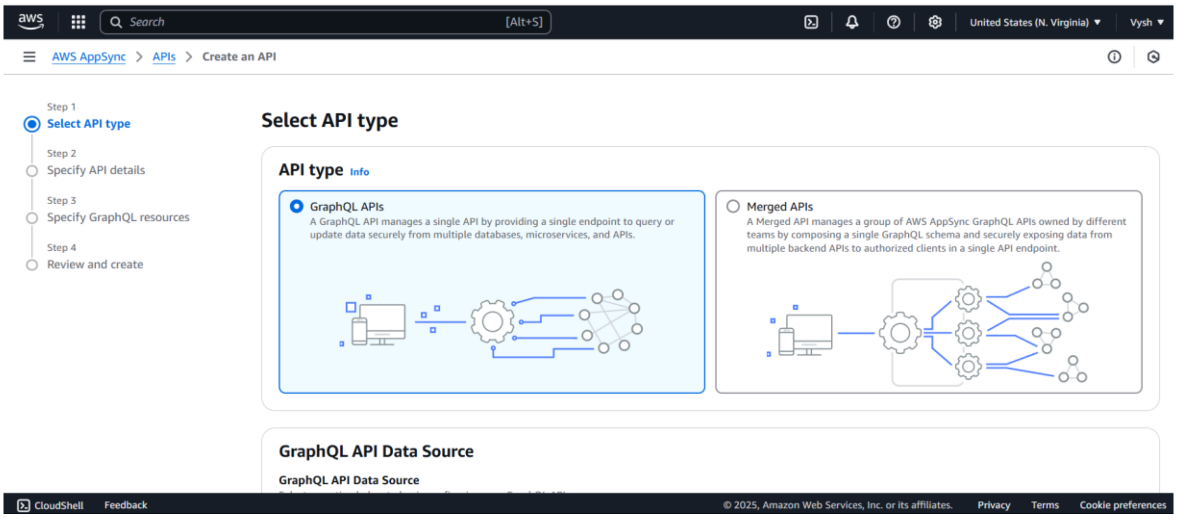
9.Deploying the Application:

Launched the project on a cloud-based hosting platform, ensuring availability and reliability

4.Stepwise Screenshots with brief description

Step 1:

This step begins by creating a new GraphQL API using the AWS AppSync console. The API is named SportsDashboard to reflect its purpose in handling live sports data. Once the API is created, the GraphQL schema is defined to establish the structure of the data. This schema includes object types, queries, mutations, and subscriptions.



Select API type

API type Info

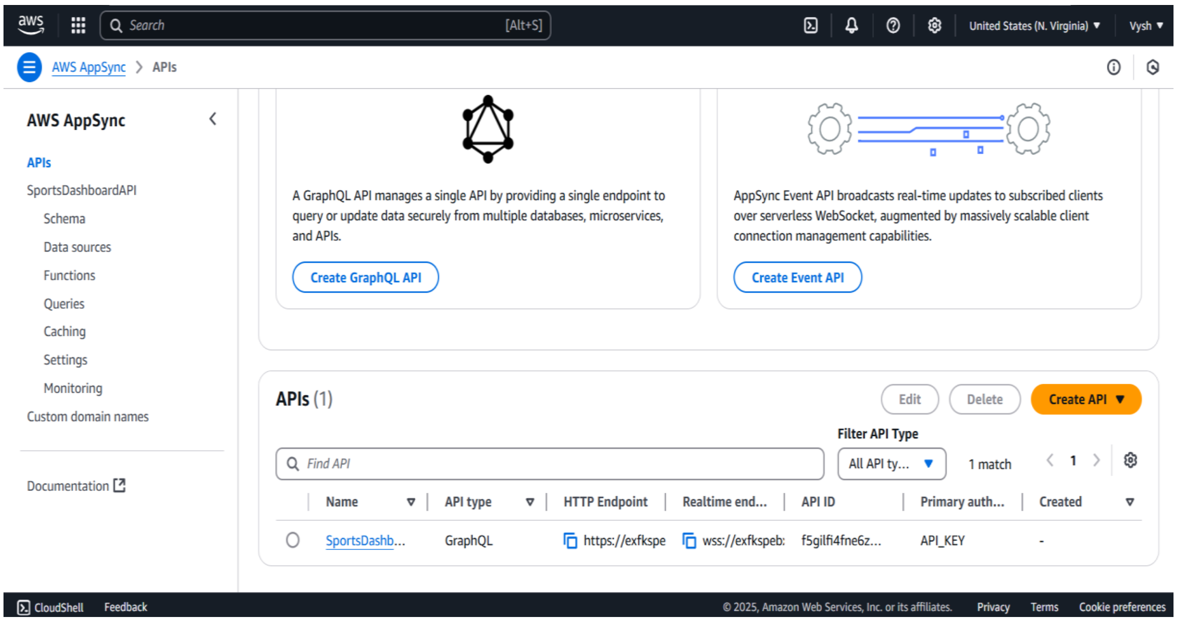
● GraphQL APIs
A GraphQL API manages a single API by providing a single endpoint to query or update data securely from multiple databases, microservices, and APIs.

○ Merged APIs
A Merged API manages a group of AWS AppSync GraphQL APIs owned by different teams by composing a single GraphQL schema and securely exposing data from multiple backend APIs to authorized clients in a single API endpoint.

GraphQL API Data Source

GraphQL API Data Source

Fig 4.1: Creating a new GraphQL API in AWS AppSync



A GraphQL API manages a single API by providing a single endpoint to query or update data securely from multiple databases, microservices, and APIs.

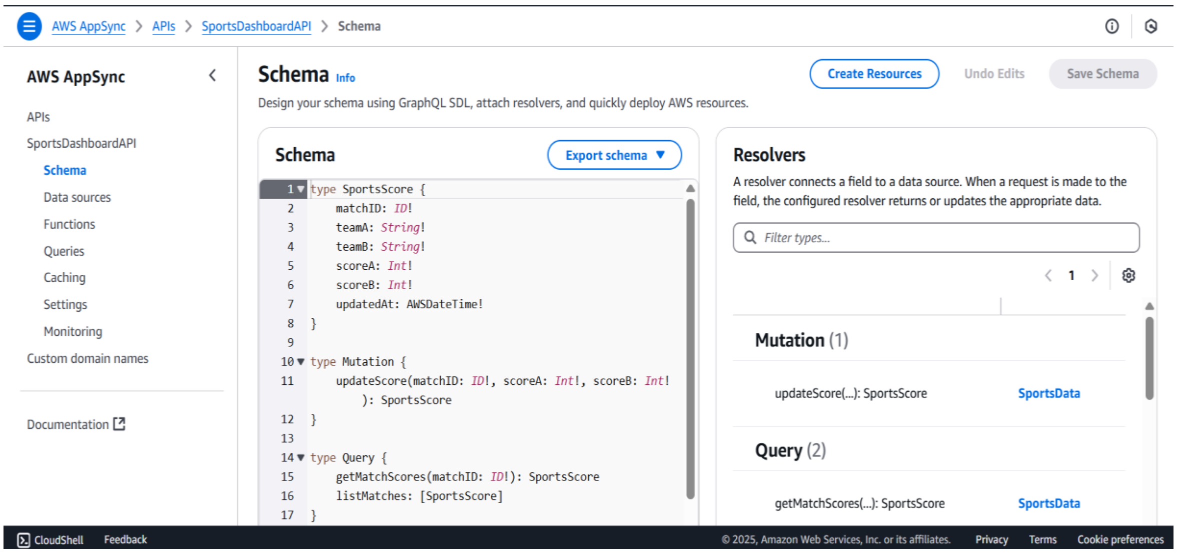
AppSync Event API broadcasts real-time updates to subscribed clients over serverless WebSocket, augmented by massively scalable client connection management capabilities.

APIs (1)

Find API | Filter API Type: All API ty... ▼ | 1 match | < 1 >

Name	API type	HTTP Endpoint	Realtime end...	API ID	Primary auth...	Created
SportsDashb...	GraphQL	https://exfkspe	wss://exfkspeb:	f5gilfi4fne6z...	API_KEY	-

Fig 4.2: Created a GraphQL API named SportsDashboard



AWS AppSync > APIs > SportsDashboardAPI > Schema

AWS AppSync

- APIs
- SportsDashboardAPI
  - Schema
  - Data sources
  - Functions
  - Queries
  - Caching
  - Settings
  - Monitoring
- Custom domain names

Documentation

Schema Info

Design your schema using GraphQL SDL, attach resolvers, and quickly deploy AWS resources.

Schema

```
type SportsScore {
    matchID: ID!
    teamA: String!
    teamB: String!
    scoreA: Int!
    scoreB: Int!
    updatedAt: AWSDateTime!
}

type Mutation {
    updateScore(matchID: ID!, scoreA: Int!, scoreB: Int!
        ): SportsScore
}

type Query {
    getMatchScores(matchID: ID!): SportsScore
    listMatches: [SportsScore]
}
```

Resolvers

A resolver connects a field to a data source. When a resolver is made to the field, the configured resolver returns or updates the appropriate data.

Mutation (1)

updateScore(...): SportsScore	SportsData

Query (2)

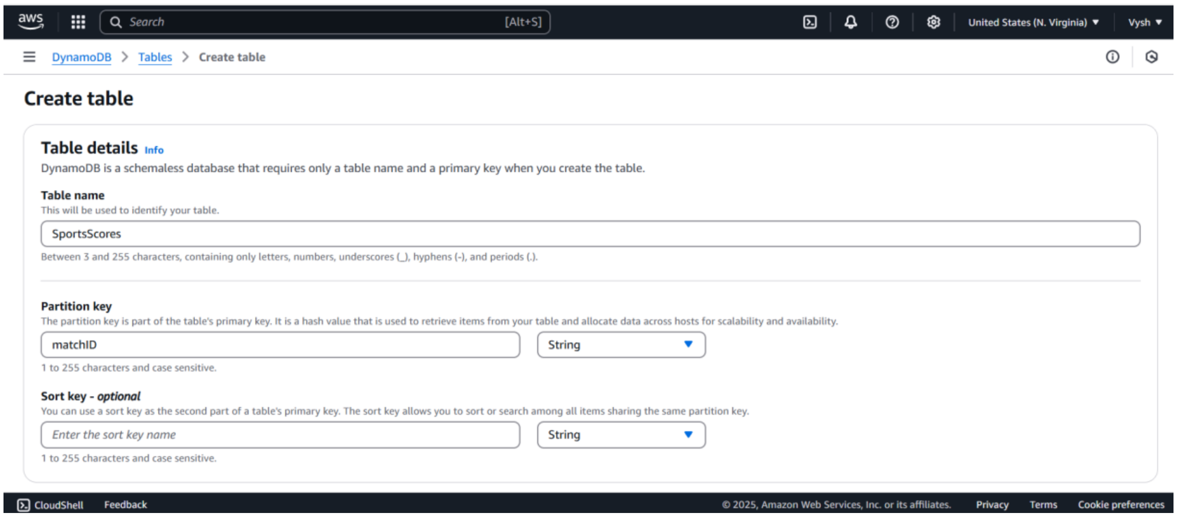
getMatchScores(...): SportsScore	SportsData

© 2025, Amazon Web Services, Inc. or its affiliates.

Fig 4.3: Defining Schema

Step 2:

A new table is created in DynamoDB to store live match data, and it's named SportsScores. The table includes necessary attributes such as MatchID for partitioning and timestamp for sorting. Next, a data source is created in AppSync and connected to this table, allowing the API to interact with DynamoDB. This data source is named SportsData for clarity. Resolvers are then attached to map the GraphQL operations (queries and mutations) to the corresponding actions in DynamoDB. Finally, request mapping templates are configured to control how GraphQL requests are translated into DynamoDB commands and how responses are handled



DynamoDB > Tables > Create table

Create table

Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

SportsScores

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

matchID | String

1 to 255 characters and case sensitive.

Sort key - *optional*

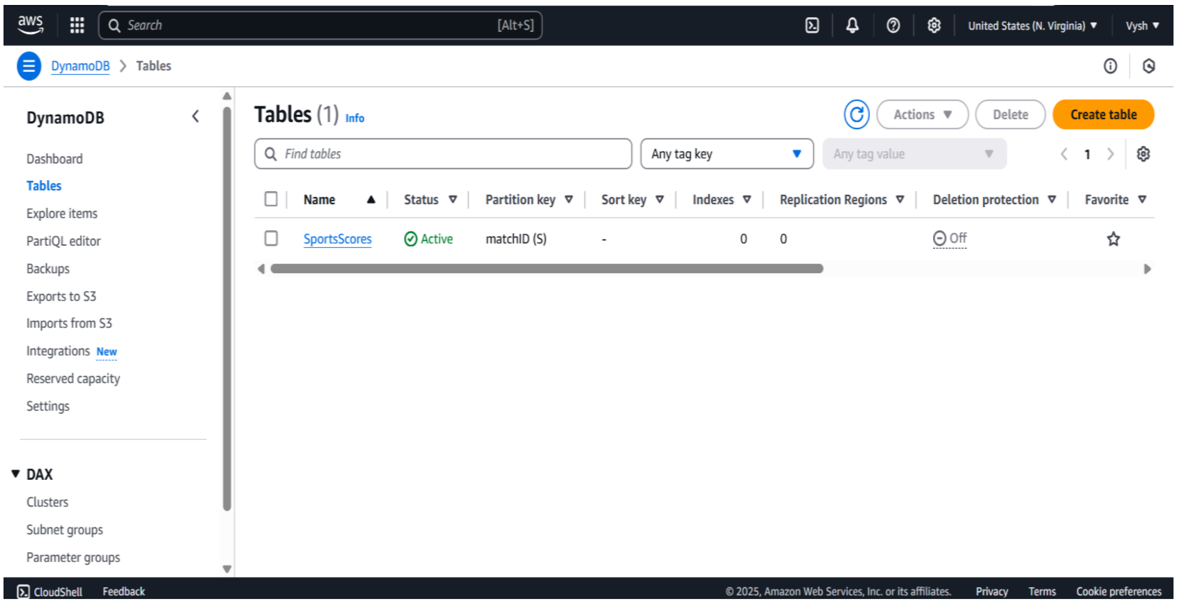
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

Enter the sort key name | String

1 to 255 characters and case sensitive.

© 2025, Amazon Web Services, Inc. or its affiliates.

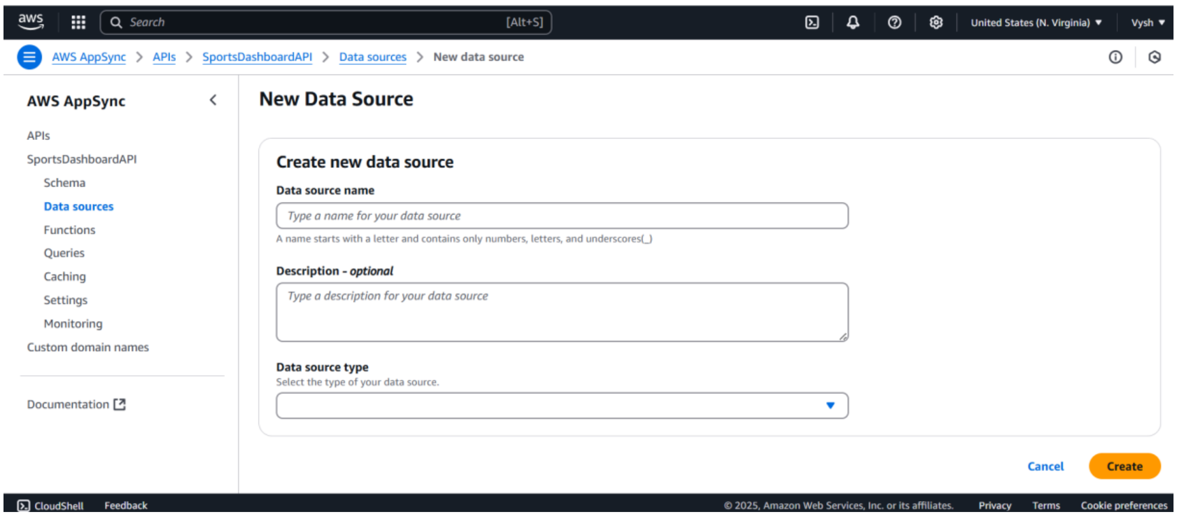
Fig 4.4: Creating a new Table in DynamoDB



Tables (1) Info

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite
SportsScores	✓ Active	matchID (S)	-	0	0	Off	☆

Fig 4.5: Created table named SportsScores



New Data Source

Create new data source

Data source name

Type a name for your data source

A name starts with a letter and contains only numbers, letters, and underscores(_)

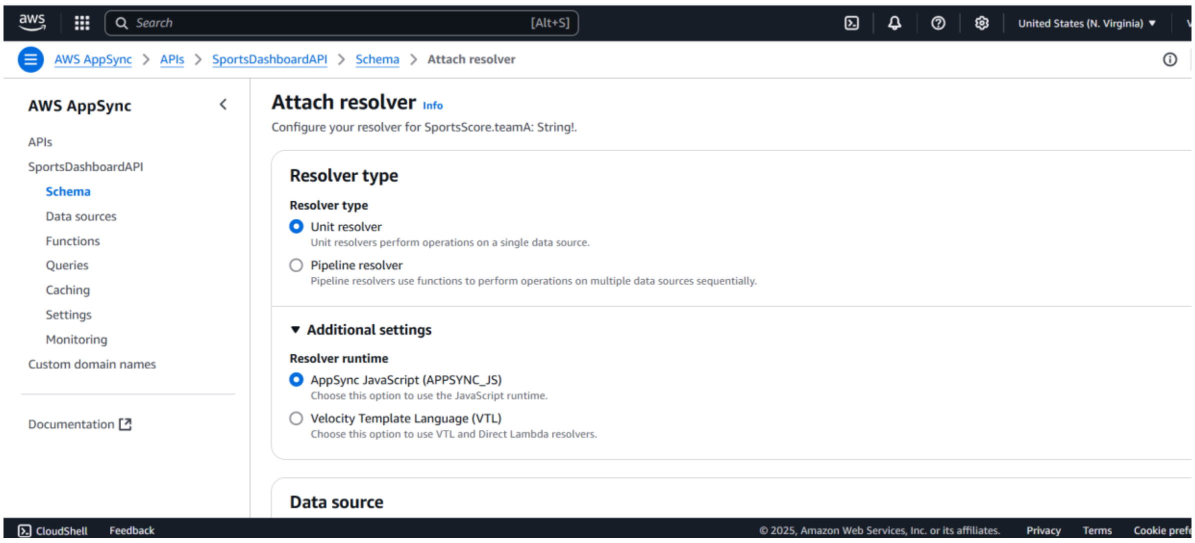
Description - *optional*

Type a description for your data source

Data source type

Select the type of your data source.

Fig 4.6: Creating A new data Source



Attach resolver Info

Configure your resolver for SportsScore.teamA: String!.

Resolver type

Resolver type

[✓ Unit resolver] — Unit resolvers perform operations on a single data source.

Pipeline resolver — Pipeline resolvers use functions to perform operations on multiple data sources sequentially.

Additional settings

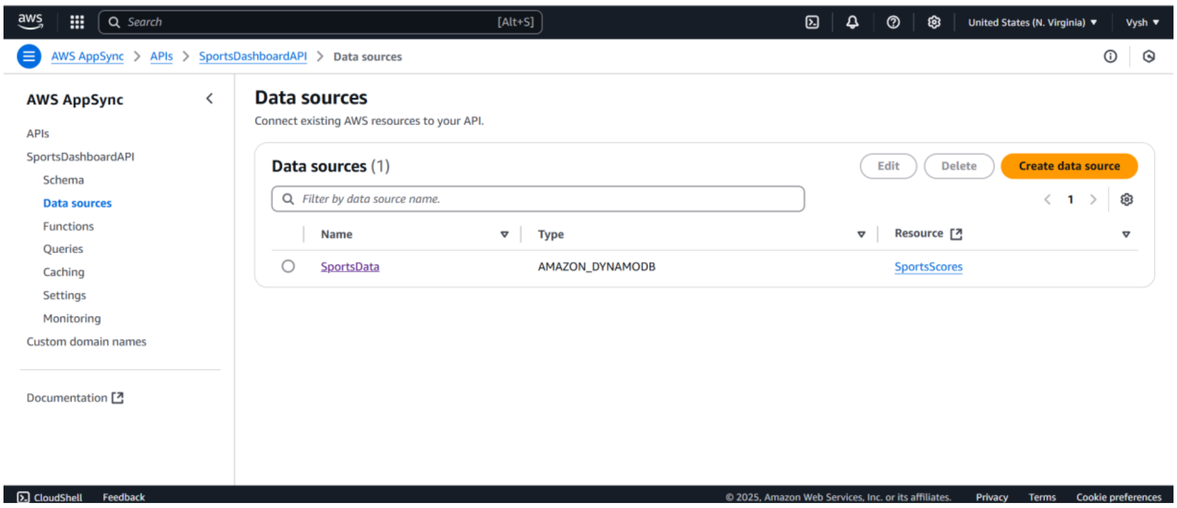
Resolver runtime

[✓ AppSync JavaScript (APPSYNC_JS)] — Choose this option to use the JavaScript runtime.

Velocity Template Language (VTL) — Choose this option to use VTL and Direct Lambda resolvers.

Data source

Fig 4.7: Attaching Resolvers



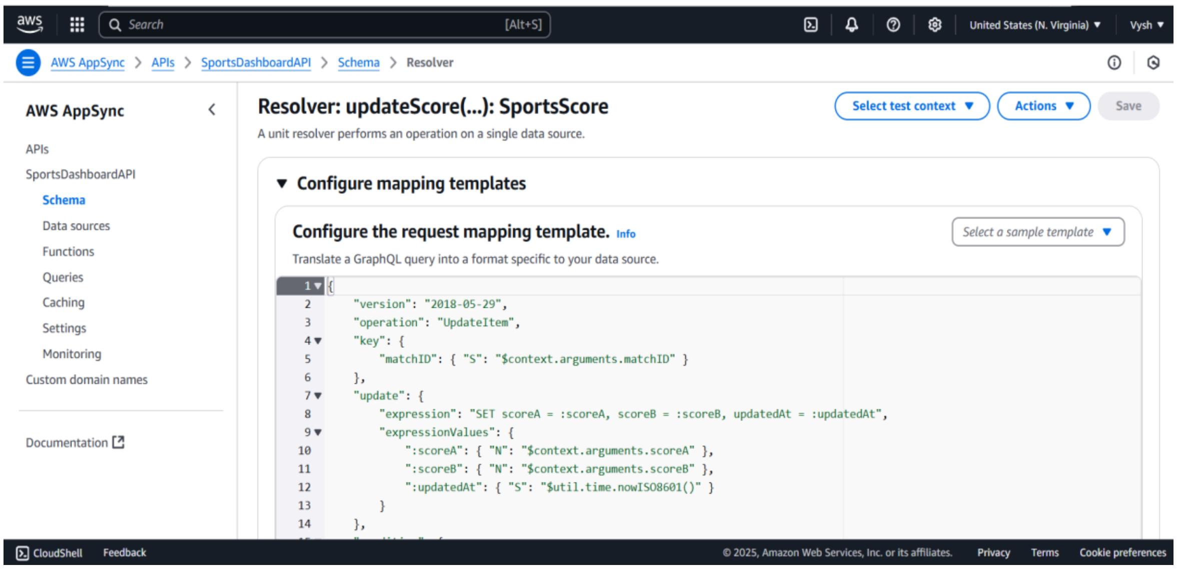
Data sources

Connect existing AWS resources to your API.

Data sources (1)

Name	Type	Resource
SportsData	AMAZON_DYNAMODB	SportsScores

Fig 4.8: Created a new Data Source Named SportsData



Resolver: updateScore(...): SportsScore

A unit resolver performs an operation on a single data source.

Configure the request mapping template. Info

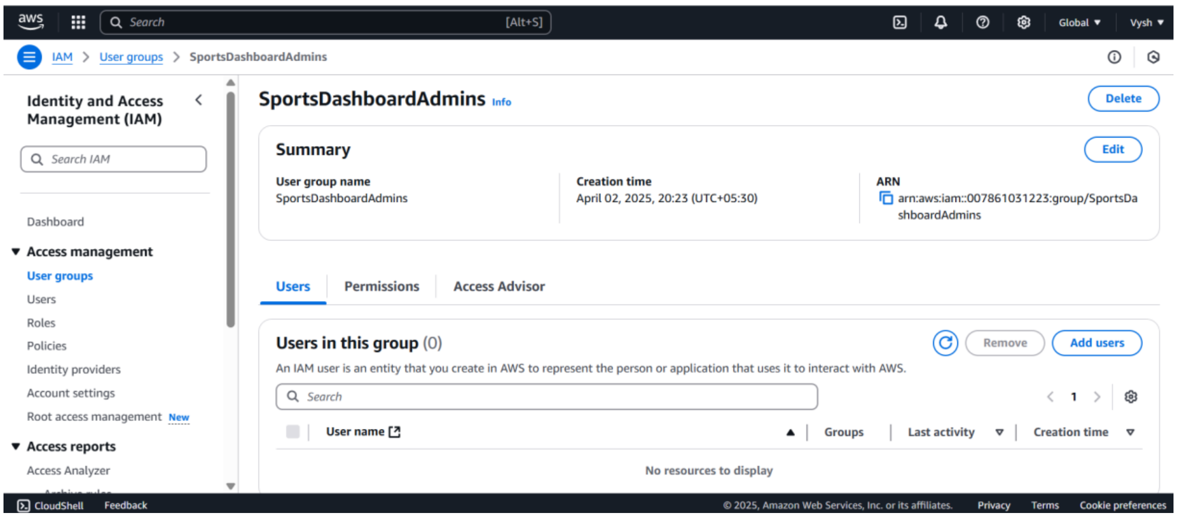
Translate a GraphQL query into a format specific to your data source.

```
{
    "version": "2018-05-29",
    "operation": "UpdateItem",
    "key": {
        "matchID": { "S": "$context.arguments.matchID" }
    },
    "update": {
        "expression": "SET scoreA = :scoreA, scoreB = :scoreB, updatedAt = :updatedAt",
        "expressionValues": {
            ":scoreA": { "N": "$context.arguments.scoreA" },
            ":scoreB": { "N": "$context.arguments.scoreB" },
            ":updatedAt": { "S": "$util.time.nowISO8601()" }
        }
    },
```

Fig 4.9: Configuring the request Mapping Templates

Step 3:

To manage permissions securely, a new user group is created in AWS IAM. Within this group, a specific IAM user is added to represent application-level access. Permissions such as dynamodb:Query, dynamodb:PutItem, and other relevant actions are assigned to the user, allowing secure interaction between AppSync and DynamoDB. The access details for this user are carefully noted down for future use in the application integration.



SportsDashboardAdmins Info

Summary

User group name: SportsDashboardAdmins

Creation time: April 02, 2025, 20:23 (UTC+05:30)

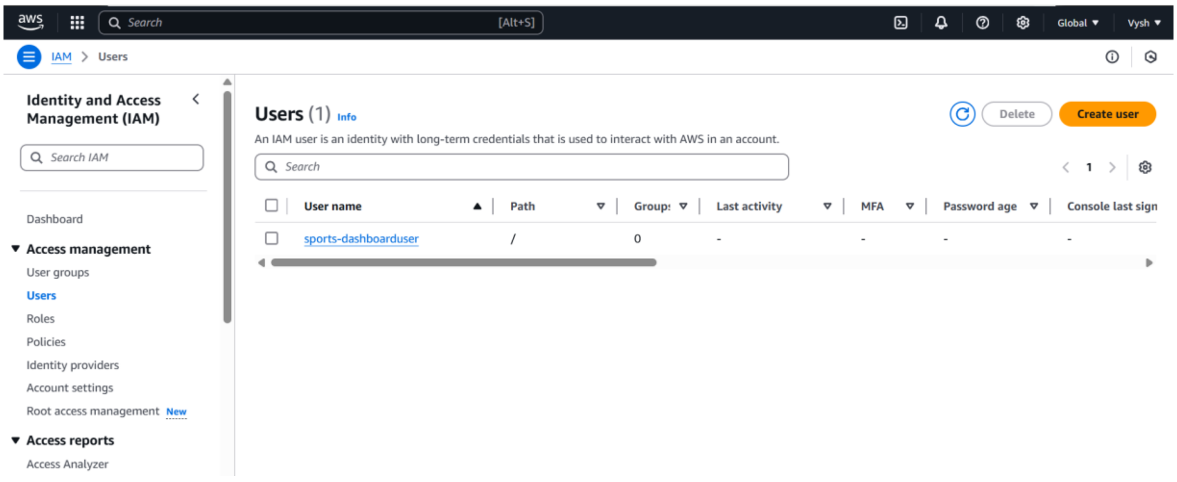
ARN: arn:aws:iam::007861031223:group/SportsDashboardAdmins

Users in this group (0)

An IAM user is an entity that you create in AWS to represent the person or application that uses it to interact with AWS.

No resources to display

Fig 4.10: Creating new User Group in IAM

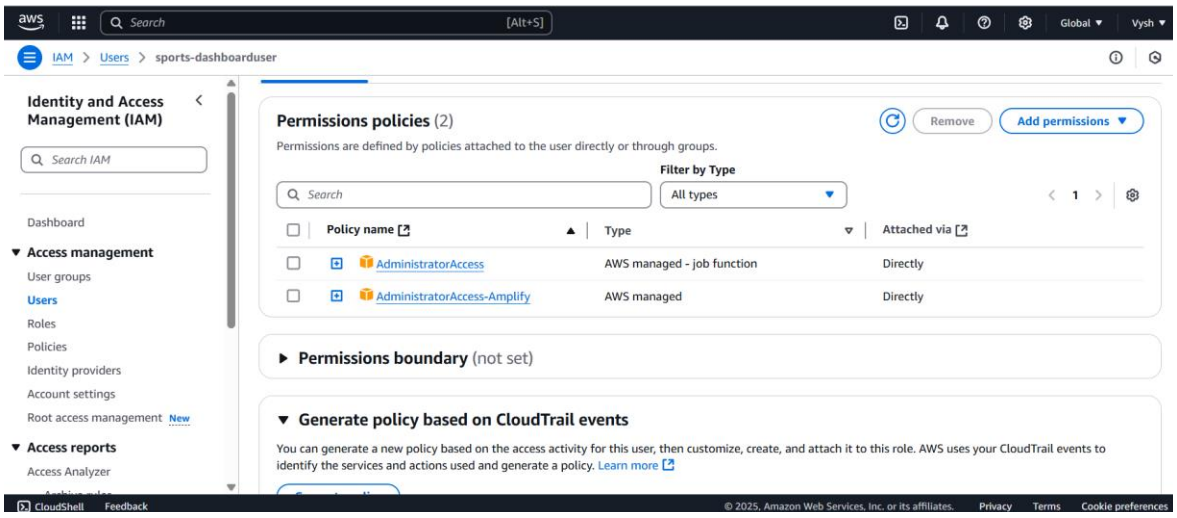


Users (1)

An IAM user is an identity with long-term credentials that is used to interact with AWS in an account.

User name	Path	Groups	Last activity	MFA	Password age	Console last sign
sports-dashboarduser	/	0	-	-	-	-

Fig 4.11: Creating an IAM User



Permissions policies (2)

Permissions are defined by policies attached to the user directly or through groups.

Filter by Type: All types

Policy name	Type	Attached via
AdministratorAccess	AWS managed - job function	Directly
AdministratorAccess-Amplify	AWS managed	Directly

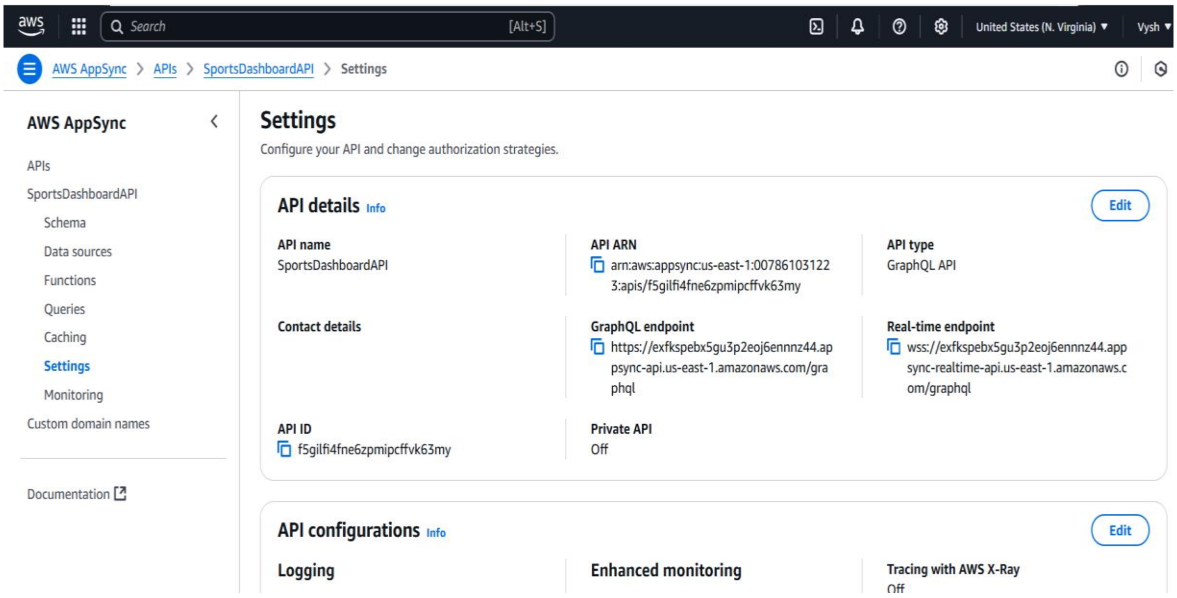
Permissions boundary (not set)

Generate policy based on CloudTrail events

You can generate a new policy based on the access activity for this user, then customize, create, and attach it to this role. AWS uses your CloudTrail events to identify the services and actions used and generate a policy. Learn more

IAM > Users > sports-dashboarduser
Remove
Add permissions
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Fig 4.12: Adding the permissions and giving access to the User


AWS AppSync > APIs > SportsDashboardAPI > Settings

Settings

Configure your API and change authorization strategies.

API details Info

API name SportsDashboardAPI	API ARN arn:aws:appsync:us-east-1:007861031223:apis/f5gilfi4fne6zpmipcffvk63my	API type GraphQL API
Contact details	GraphQL endpoint https://exfkspebx5gu3p2eoj6ennnz44.appsync-api.us-east-1.amazonaws.com/graphql	Real-time endpoint wss://exfkspebx5gu3p2eoj6ennnz44.appsync-realtime-api.us-east-1.amazonaws.com/graphql
API ID f5gilfi4fne6zpmipcffvk63my	Private API Off	

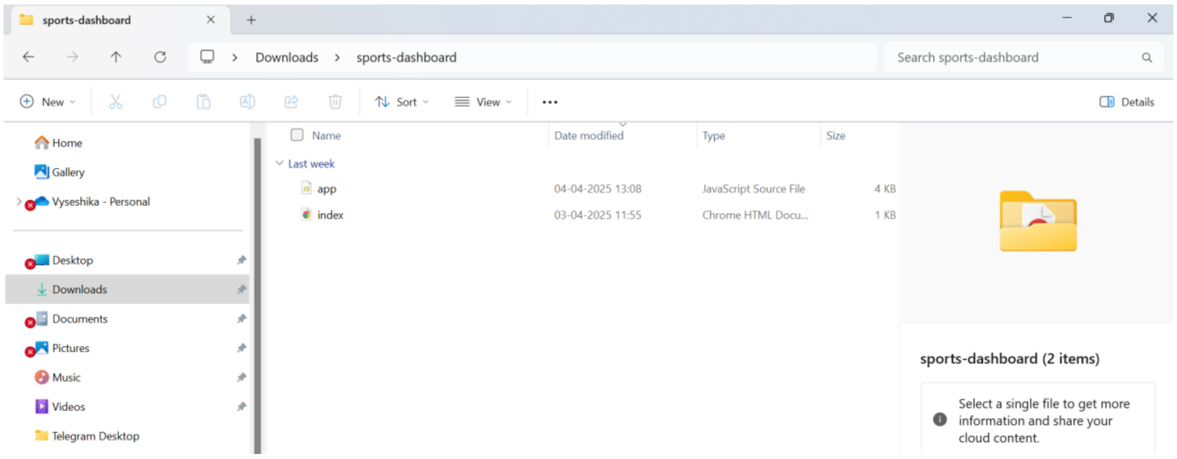
API configurations Info

Logging | Enhanced monitoring | Tracing with AWS X-Ray: Off

Fig 4.13: Note the details

Step 4:

A new folder is created to organize the frontend files of the project. Inside this folder, the main application script (app.js) and the HTML structure (index.html) are developed. The frontend uses JavaScript to subscribe to GraphQL updates and dynamically displays match scores using Chart.js. This setup ensures that any updates in match data are immediately reflected in the user interface without needing manual page refreshes.

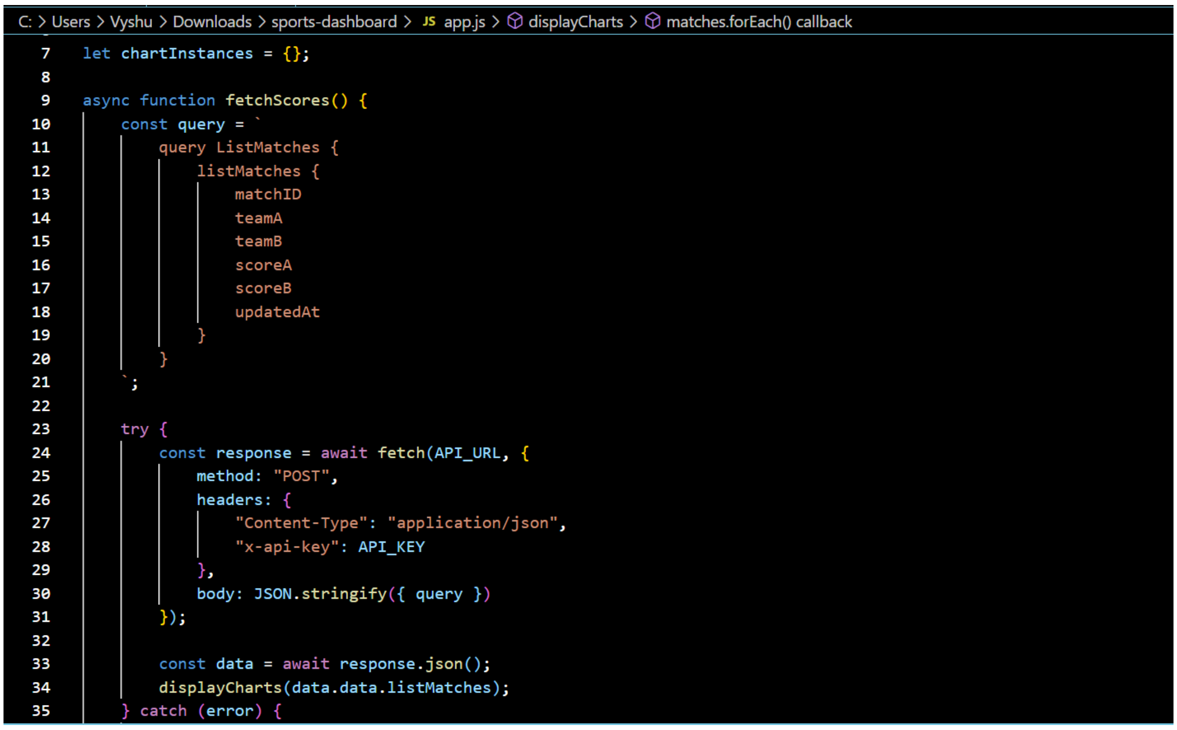

Downloads > sports-dashboard

Name	Date modified	Type	Size
Last week			
app	04-04-2025 13:08	JavaScript Source File	4 KB
index	03-04-2025 11:55	Chrome HTML Docu...	1 KB

sports-dashboard (2 items)

Select a single file to get more information and share your cloud content.

Fig 4.14: Create a new Folder


C: > Users > Vyshu > Downloads > sports-dashboard > JS app.js > displayCharts > matches.forEach() callback

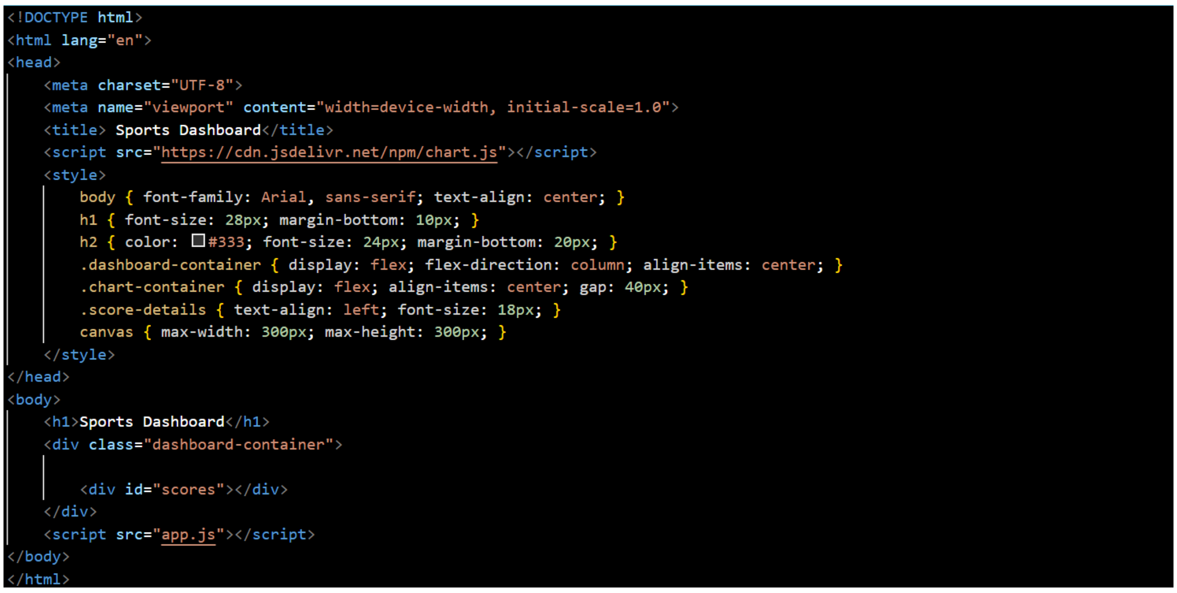
```javascript
let chartInstances = {};

async function fetchScores() {
    const query = `
        query ListMatches {
            listMatches {
                matchID
                teamA
                teamB
                scoreA
                scoreB
                updatedAt
            }
        }
    `;

    try {
        const response = await fetch(API_URL, {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
                "x-api-key": API_KEY
            },
            body: JSON.stringify({ query })
        });

        const data = await response.json();
        displayCharts(data.data.listMatches);
    } catch (error) {
```

Fig 4.15: Develop the app.js



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title> Sports Dashboard</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <style>
        body { font-family: Arial, sans-serif; text-align: center; }
        h1 { font-size: 28px; margin-bottom: 10px; }
        h2 { color: #333; font-size: 24px; margin-bottom: 20px; }
        .dashboard-container { display: flex; flex-direction: column; align-items: center; }
        .chart-container { display: flex; align-items: center; gap: 40px; }
        .score-details { text-align: left; font-size: 18px; }
        canvas { max-width: 300px; max-height: 300px; }
    </style>
</head>
<body>
    <h1>Sports Dashboard</h1>
    <div class="dashboard-container">

        <div id="scores"></div>
    </div>
    <script src="app.js"></script>
</body>
</html>
```

Fig 4.16: Develop the index.html

Step 5:

After all configurations and coding are complete, the project is launched. The final output is a working real-time sports dashboard that displays live cricket scores. The frontend connects seamlessly with AppSync and DynamoDB, and updates are shown instantly through visual charts and dynamic content.

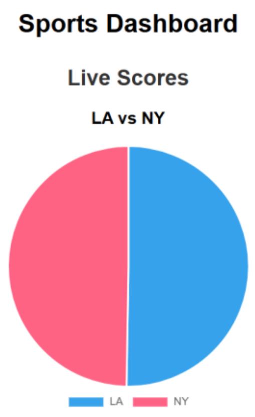


Fig 4.17: Final Outcome after launching

5.Learning Outcomes

During the course of this project, hands-on experience was gained in using AWS AppSync and DynamoDB to build serverless applications. The process involved learning how to configure IAM roles effectively to ensure secure interactions between AWS services. A solid understanding of real-time data synchronization was developed through the use of GraphQL subscriptions. Additionally, the project involved exploring methods to efficiently fetch and process external API data. Skills in JavaScript and Chart.js were strengthened for developing interactive and dynamic data visualizations. The role of API Gateway in managing secure access to external data sources was also better understood. Overall, the project enhanced problem-solving abilities by addressing and troubleshooting various challenges encountered in deploying cloud-based applications.

6.Conclusion

This project successfully demonstrates how AWS cloud services can be used to create a real-time sports dashboard with automated data updates. By leveraging AWS AppSync, DynamoDB, and IAM roles, we built a system that efficiently fetches, processes, and displays live cricket scores in an interactive format. The serverless architecture ensures scalability, cost efficiency, and minimal maintenance, making it a practical solution for real-time data-driven applications.

7.Future Scope

In the future, this system can be enhanced with additional features such as player performance analytics, access to detailed player statistics, historical match data visualization, user-customizable dashboards, and push notifications for live score updates. These improvements can offer a more immersive and informative experience for users, expanding the dashboard's capabilities and overall engagement

7.References

[1]. hams, R., Wang, X., & Li, Y. (2021). "Efficiency of GraphQL Subscriptions in Real-Time Applications." *Journal of Web Engineering*, 20(3), 145-162.

[2]. Hussain, M., Patel, S., & Kumar, R. (2020). "Performance Analysis of NoSQL Databases in Real-Time Applications." *International Journal of Computer Science and Information Security*, 18(5), 231-245.

[3]. Amazon Web Services (AWS). (2023). "AWS AppSync: Real-Time Data Synchronization with GraphQL." Retrieved from https://aws.amazon.com/appsync/

[4]. Amazon Web Services (AWS). (2023). "Amazon DynamoDB: High-Performance NoSQL Database." Retrieved from https://aws.amazon.com/dynamodb/

[5]. ESPN. (2022). "Leveraging Cloud Technologies for Real-Time Sports Data." Retrieved from https://athelogroup.com/blog/ai-in-sports-how-espn-is-leveraging-tech-to-highlight-niche-sports/