

Real-Time Chat Application using AWS AppSync & DynamoDB

Project Overview

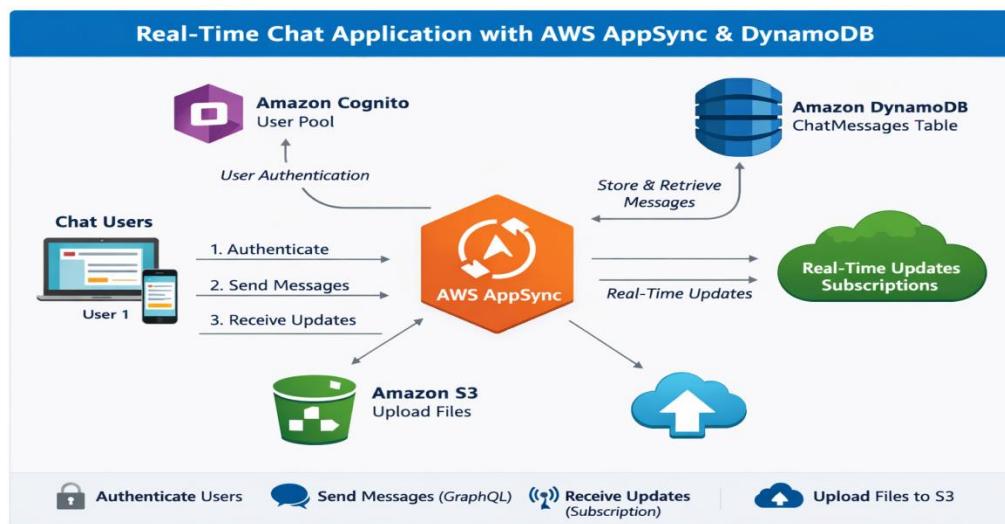
Services Used:

- **AWS AppSync** – Managed GraphQL service with real-time subscriptions
- **Amazon DynamoDB** – NoSQL database for chat messages
- **Amazon Cognito** – User authentication & authorization
- **Amazon S3** – File/image uploads in chat

Goal: Build a secure real-time chat backend using only the AWS Console.

Architecture (High Level)

1. Users sign up / sign in using **Cognito User Pool**
2. Frontend connects to **AppSync GraphQL API**
3. Messages are stored in **DynamoDB**
4. Real-time updates via **GraphQL Subscriptions**
5. Files/images uploaded to S3



STEP 1: Create DynamoDB Table (Chat Messages)

1. Go to **AWS Console** → **DynamoDB**
2. Click **Create table**
3. Table details:
 - Table name: ChatMessages
 - Partition key: roomId (String)
 - Sort key: timestamp (String)
4. Table settings: Choose **Default settings**
5. Click **Create table**

This allows storing messages per chat room in time order

The screenshot shows the AWS DynamoDB console with the ChatMessages table selected. The table is active and has 0 items. The capacity mode is On-demand. The Point-in-time recovery (PITR) policy is set to Off. The ARN of the table is arn:aws:dynamodb:ap-south-1:162308490504:table/ChatMessages.

STEP 2: Create Cognito User Pool (Authentication)

1. Go to AWS Console → Cognito
2. Click **Create user pool**
3. Select **Cognito User Pool**=_Real-Time Chat Application

Configure sign-in experience

- Authentication provider: **Cognito User Pool**
- Sign-in options: **Email**

Security requirements

- Password policy: Default
- MFA: Optional (Disable for demo)

Sign-up experience

- Enable self sign-up: **Yes**
- Required attributes: Email

App integration

- User pool name: ChatUserPool
- App client name: ChatAppClient
- Client secret: **X**Do NOT generate

4. Click **Create user pool**

Overview: ChatUserPool

User pool information

- User pool name: ChatUserPool
- User pool ID: ap-south-1_1M27OnDDF
- ARN: arn:aws:cognito-idp:ap-south-1:162308490504:userpool/ap-south-1_1M27OnDDF
- Token signing key URL: https://cognito-idp.ap-south-1.amazonaws.com/ap-south-1_1M27OnDDF/well-known/jwks.json
- Estimated number of users: 0
- Feature plan: Essentials

Recommendations

App client: Real-Time Chat Application

App client information

- App client name: Real-Time Chat Application
- Client ID: 3v1tl6t84khmfqavha0aqu6vim
- Client secret: (Show client secret)
- Authentication flows: Choice-based sign-in, Secure remote password (SRP), Get user tokens from existing authenticated sessions
- Authentication flow session duration: 3 minutes
- Refresh token expiration: 5 day(s)
- Access token expiration: 60 minutes
- ID token expiration: 60 minutes
- Advanced authentication settings: Enable token revocation, Enable prevent user existence errors

Quick setup guide | Attribute permissions | Login pages | Threat protection | Analytics

STEP 3: Create S3 Bucket (File Uploads)

1. Go to AWS Console → S3
2. Click **Create bucket**
3. Bucket name: chat-app-uploads-appasync
4. Region: Same as AppSync(Mumbai)
5. Block Public Access: Uncheck all
6. Click **Create bucket**

Enable CORS

1. Open bucket → Permissions → CORS
2. Add:

```
[
  {
    "AllowedHeaders": ["*"],
    "AllowedMethods": ["PUT", "GET"],
    "AllowedOrigins": ["*"],
    "ExposeHeaders": []
  }
]
```

Cross-origin resource sharing (CORS)

```
[{"AllowedHeaders": ["*"], "AllowedMethods": ["PUT", "GET"], "AllowedOrigins": ["*"], "ExposeHeaders": []}]
```

STEP 4: Create AppSync GraphQL API

1. Go to AWS Console → AppSync
2. Click **Create API**
3. Choose **Build from scratch**
4. API name: ChatAppAPI
5. Authorization type: **Amazon Cognito User Pool**
6. Select User Pool: ChatUserPool
7. Click **Create API**

Name	API type	HTTP Endpoint	Realtime endpoint	API ID	Primary auth type	Created
ChatAppAPI	GraphQL	https://g62ysr4yj	wss://g62ysr4yzf	wowsnre77bde...	API_KEY	-

STEP 5: Define GraphQL Schema

In AppSync → Schema → Edit schema

```
type Message {
  roomId: ID!
  timestamp: String!
  sender: String!
  content: String
  fileUrl: String
}
```

```
input MessageInput {
  roomId: ID!
  timestamp: String!
  sender: String!
  content: String
  fileUrl: String
}
```

```

}

type Query {
  getMessages(roomId: ID!): [Message]
}

type Mutation {
  sendMessage(input: MessageInput!): Message
}

type Subscription {
  onNewMessage(roomId: ID!): Message
    @aws_subscribe(mutations: ["sendMessage"])
}

schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

```

Click **Save schema**

The screenshot shows the AWS AppSync Schema editor interface. On the left, a sidebar lists navigation options: AWS AppSync, APIs, ChatAppAPI, Schema (selected), Data sources, Functions, Queries, Caching, Settings, Monitoring, and Custom domain names. Below this is a Documentation link. At the bottom of the sidebar are CloudShell, Feedback, and Console Mobile App buttons. The main workspace has a header with a search bar, account information (Account ID: 1623-0849-0504, Region: Asia Pacific (Mumbai)), and a user name (kavyanjali). The top right features a 'Create Event API' button. A blue banner at the top says 'New: Introducing AWS AppSync Events! Create fully-managed WebSocket APIs powered by AppSync. Publish, broadcast, and subscribe to events with ease. Get started now!' Below it, a green banner says 'API key was successfully created.' The central area is titled 'Schema Info' and contains the GraphQL schema. To the right is a 'Resolvers' panel with a 'Message (5)' section showing fields for roomId and timestamp with 'Attach' buttons. The bottom right of the workspace includes copyright information (© 2025, Amazon Web Services, Inc. or its affiliates.) and links for Privacy, Terms, and Cookie preferences.

STEP 6: Create DynamoDB Data Source

1. AppSync → Data Sources
2. Click **Create data source**
3. Type: **Amazon DynamoDB**
4. Name: ChatMessagesDS
5. Table: ChatMessages
6. IAM Role: Create new role
7. Click **Create**

The screenshot shows the AWS AppSync Data sources page. On the left, there's a sidebar with options like AWS AppSync, APIs, Schema, Data sources (which is selected), Functions, Queries, Caching, Settings, Monitoring, and Custom domain names. Below the sidebar is a link to Documentation. The main area has a blue header bar with a message about AWS AppSync Events and a green success message about an API key being created. Below this is a section titled 'Data sources' with a sub-section 'Data sources (1/2)'. It lists two entries: 'Message' (Type: AMAZON_DYNAMODB, Resource: Message) and 'ChatMessagesDS' (Type: AMAZON_DYNAMODB, Resource: ChatMessages). There are buttons for Edit, Delete, and Create data source.

STEP 7: Attach Resolvers

Mutation Resolver (sendMessage)

1. Schema → Mutation → sendMessage
2. Attach resolver → DynamoDB
3. Use **Unit Resolver**

Request Mapping Template:

```
{
  "version": "2018-05-29",
  "operation": "PutItem",
  "key": {
    "roomId": {"S": "$ctx.args.input.roomId"},
    "timestamp": {"S": "$ctx.args.input.timestamp"}
  },
  "attributeValues": {
    "sender": {"S": "$ctx.args.input.sender"},
    "content": {"S": "$ctx.args.input.content"},
    "fileUrl": {"S": "$ctx.args.input.fileUrl"}
  }
}
```

Response Mapping Template:

```
$util.toJson($ctx.args.input)
```

Query Resolver (getMessages)

1. Query → getMessages
2. Attach resolver → DynamoDB

Request Template:

```
{
  "version": "2018-05-29",
  "operation": "Query",
  "query": {
```

```
        "expression": "roomId = :roomId",
        "expressionValues": {
            ":roomId": {"S": "$ctx.args.roomId"}
        }
    }
}
```

Response Template:

```
$util.toJson($ctx.result.items)
```

STEP 8: Test GraphQL API

1. AppSync → Queries
2. Login using Cognito user

Test Mutation

```
mutation {
  sendMessage(input: {
    roomId: "room1",
    timestamp: "2025-01-01T10:00:00",
    sender: "user1",
    content: "Hello!"
  }) {
    roomId
    content
  }
}
```

Test Subscription

```
subscription {
  onNewMessage(roomId: "room1") {
    content
    sender
  }
}
```

STEP 9: File Upload Flow (S3)

1. User uploads file to S3 (via frontend)
 2. S3 returns file URL
 3. URL stored in DynamoDB using sendMessage
-

STEP 10: Security & IAM Notes

- Cognito secures AppSync access
 - AppSync IAM role allows DynamoDB read/write
 - S3 access controlled via bucket policy or pre-signed URLs
-

Summary

In this project, a **Real-Time Chat Application** is designed and implemented using **AWS managed services**, focusing on scalability, security, and real-time communication.

The application uses **AWS AppSync** as the core service to provide a **GraphQL API** that handles all client communication. AppSync enables **queries** for fetching chat messages, **mutations** for sending messages, and **subscriptions** for real-time message delivery without polling.

Amazon Cognito manages user authentication and authorization. Users sign up and sign in using a Cognito User Pool, and only authenticated users are allowed to access the AppSync API. This ensures secure communication between clients and backend services.

Amazon DynamoDB is used as the backend database to store chat messages. Messages are stored using a partition key (roomId) and sort key (timestamp), which allows efficient querying of messages per chat room in chronological order. DynamoDB's serverless and highly scalable nature makes it suitable for real-time chat workloads.

Amazon S3 is used for file and media uploads such as images or documents shared in chat messages. Uploaded file URLs are stored along with chat messages in DynamoDB, enabling users to access shared files securely.

The entire system is implemented using **AWS Console only**, without the need for server management or infrastructure provisioning, making it a fully serverless architecture.

Conclusion

This Real-Time Chat Application demonstrates how modern cloud-native applications can be built using **serverless AWS services**. By combining AppSync, DynamoDB, Cognito, and S3, the application achieves:

- **Real-time communication** using GraphQL subscriptions
- **Secure user authentication** using Amazon Cognito
- **Highly scalable data storage** using DynamoDB
- **Efficient file handling** using Amazon S3
- **Low operational overhead** with a fully managed, serverless architecture

The project effectively showcases key cloud computing concepts such as **API-driven design, real-time data streaming, authentication, NoSQL database modeling, and managed cloud services**.