

Real-Time Chat Application using WebSockets and Serverless

Project Overview

Project Name: Real-Time Chat App using WebSockets and Serverless
Goal: Build a scalable real-time messaging platform without managing servers
Architecture Type: Fully Serverless, Event-Driven
AWS Region: Choose one region (example: ap-south-1)

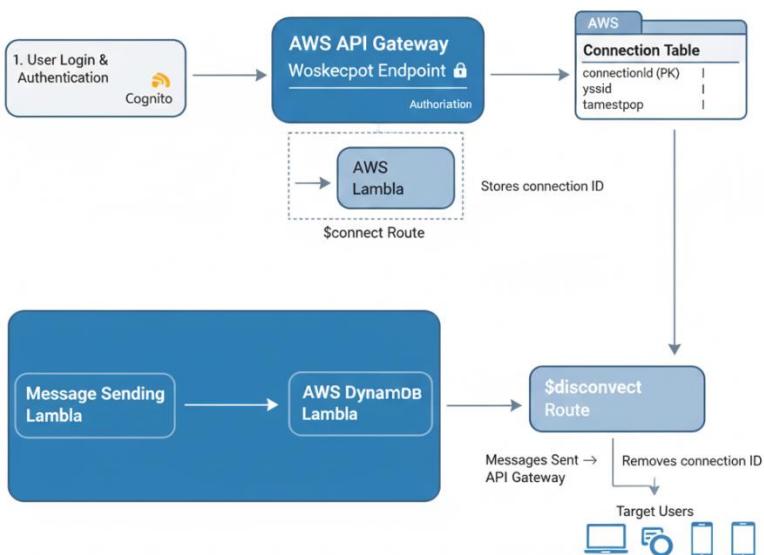
AWS Services Used

Service	Purpose
API Gateway (WebSocket API)	Real-time bidirectional communication
AWS Lambda	Handle connect, disconnect, and messages
DynamoDB	Store active WebSocket connections
Amazon Cognito	User authentication & identity federation
IAM	Secure permissions between services
CloudWatch	Logs and monitoring

High-Level Architecture Flow

1. User logs in via **Cognito**
2. Client connects to **WebSocket API**
3. \$connect Lambda stores connection ID in DynamoDB
4. Messages sent → Lambda → API Gateway → Target users
5. \$disconnect Lambda removes connection ID
- 6.

AWS Serverless Woskeskot Architecture



STEP-BY-STEP AWS CONSOLE INSTRUCTIONS

STEP 1: Create DynamoDB Table (Store WebSocket Connections)

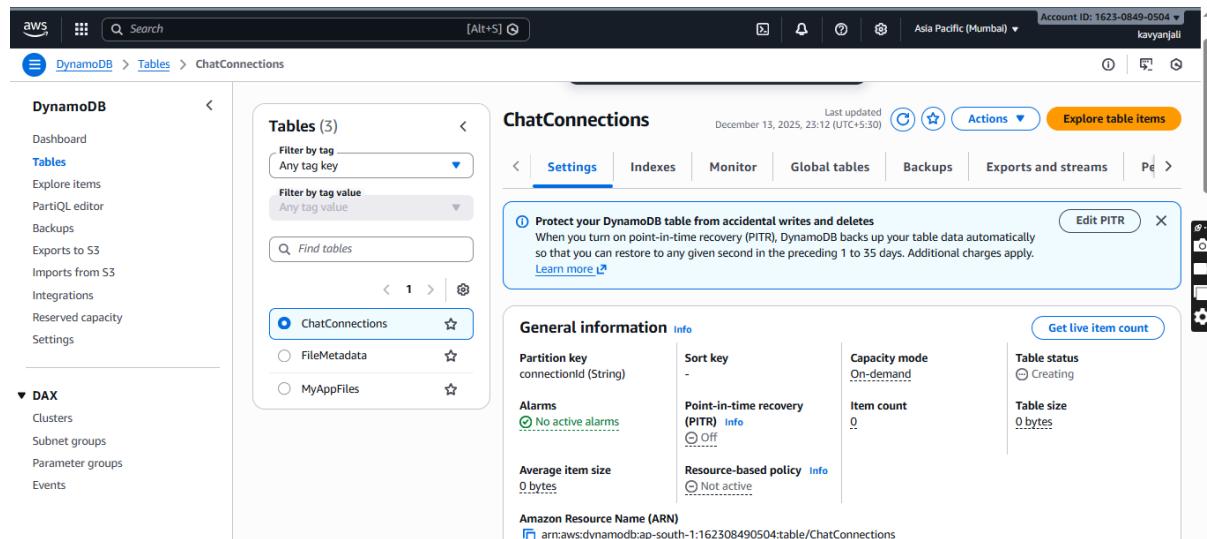
Console Steps:

1. Open AWS Console
2. Go to Services → DynamoDB
3. Click **Create table**

Configuration:

- **Table name:** ChatConnections
- **Partition key:**
 - Name: connectionId
 - Type: String
- **Table settings:** Default
- Click **Create table**

Purpose: Store active WebSocket connection IDs



The screenshot shows the AWS DynamoDB console interface. On the left, the navigation sidebar is visible with options like Dashboard, Tables, Explore items, PartiQL editor, Backups, Exports to S3, Imports from S3, Integrations, Reserved capacity, Settings, DAX, Clusters, Subnet groups, Parameter groups, and Events. The main area shows a list of tables under the 'Tables' section, with 'ChatConnections' selected. The 'ChatConnections' table details page is displayed, showing general information such as Partition key (connectionId, String), Sort key (-), Capacity mode (On-demand), Item count (0), Point-in-time recovery (PITR) status (Off), Average item size (0 bytes), Resource-based policy (Not active), and Table status (Creating). The ARN of the table is also shown as arn:aws:dynamodb:ap-south-1:162308490504:table/ChatConnections.

Step 1.2 Create Messages Table

The screenshot shows the AWS DynamoDB console with the 'MessagesTable' settings page. The table has 4 items. Key settings include:

- Partition key:** ChatRoomID (String)
- Sort key:** Timestamp (Number)
- Point-in-time recovery (PITR):** Enabled (Info)
- Capacity mode:** On-demand
- Item count:** 0
- Table status:** Active
- Table size:** 0 bytes

STEP 2: Create Cognito User Pool (Authentication)

Console Steps:

1. Go to Services → Cognito
2. Click **Create user pool**
3. Choose **Cognito User Pool**
4. Click **Next**

Configure Sign-in:

- Sign-in options: **Email**
- Password policy: Default
- MFA: Optional (disable for demo)

App Client:

- App client name: ChatAppClient
- Disable client secret

Click **Create user pool**

Purpose: Authenticate users before chat access

Screenshot of the Amazon Cognito User Pools Overview page for 'MyUserpool'.

User pool information:

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

Recommendations:

- Manage or create applications
- Apply branding to your managed login pages

Screenshot of the Amazon Cognito App clients page for 'real-time'.

App client: ChatUserPool

App client information:

- App client name: ChatUserPool
- Client ID: 2rineriggug7lftf9ns4tc7416p
- Client secret: ****
- Show client secret:
- Authentication flows: Username and password, 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

Screenshot of the Amazon Cognito User pools page for 'MyUserpool'.

Users:

User "kavyakavyanajali@gmail.com" has been created successfully.

User name	Email address	Email verified	Confirmation status	Status
c163cdba-90c1-7006-21...	kavyakavyanajali@gmail....	No	Force change password	<input checked="" type="checkbox"/> Enabled

Import users (0):

No user import jobs found.

STEP 3: Create IAM Role for Lambda

Console Steps:

1. Go to **IAM → Roles**
2. Click **Create role**
3. Trusted entity: **AWS service**
4. Service: **Lambda**
5. Click **Next**

Attach Policies:

- **AWSLambdaBasicExecutionRole**
 - **AmazonDynamoDBFullAccess**
 - **AmazonAPIGatewayInvokeFullAccess**
6. Role name: **ChatLambdaRole**
 7. Click **Create role**

Identity and Access Management (IAM)

Role ChatLambdaRole created.

Permissions policies (3)

Policy name	Type	Attached entities
AmazonAPIGatewayInvokeFull...	AWS managed	1
AmazonDynamoDBFullAccess	AWS managed	1
AWSLambdaBasicExecutionRole	AWS managed	3

Identity and Access Management (IAM)

Role APIGatewayCloudWatchLogsRole created.

Last activity

Maximum session duration

Permissions

Permissions policies (1)

Policy name	Type	Attached entities
AmazonAPIGatewayPushToClo...	AWS managed	1

STEP 4: Create Lambda Functions

You need **3 Lambda functions**.

4.1 Lambda: Connect Handler

Console Steps:

1. Go to Services → Lambda
2. Click **Create function**
3. Author from scratch

Configuration:

- Function name: ChatConnectHandler
- Runtime: **Python 3.12**
- Execution role: **Use existing role**
- Role: ChatLambdaRole
- Click **Create function**

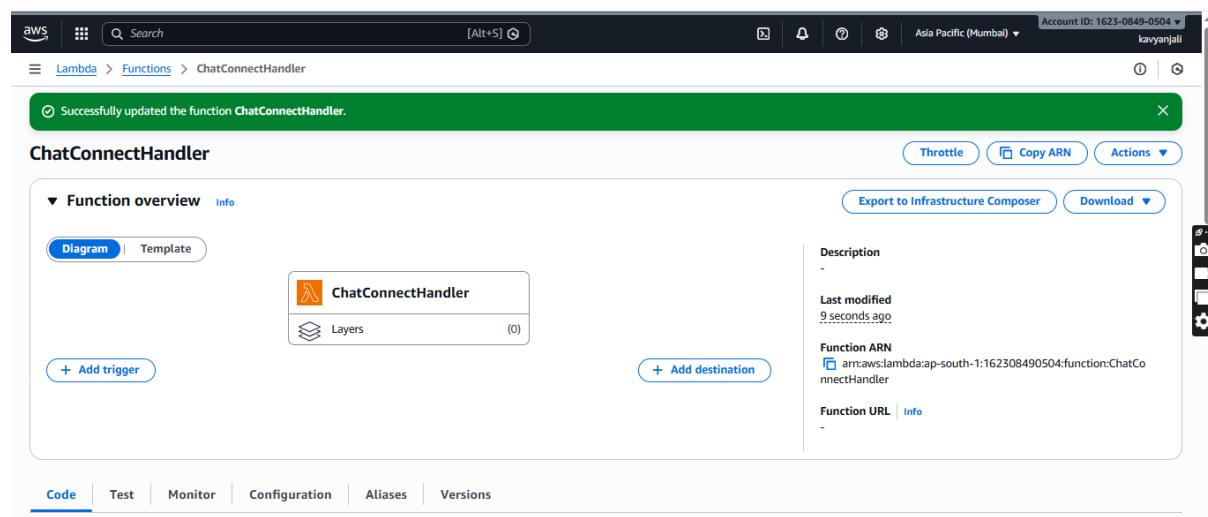
Code :

```
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('ChatConnections')

def lambda_handler(event, context):
    connection_id = event['requestContext']['connectionId']
    table.put_item(Item={'connectionId': connection_id})
    return {'statusCode': 200}
```

Click Deploy



4.2 Lambda: Disconnect Handler

Repeat steps above.

- **Function name:** ChatDisconnectHandler

Code:

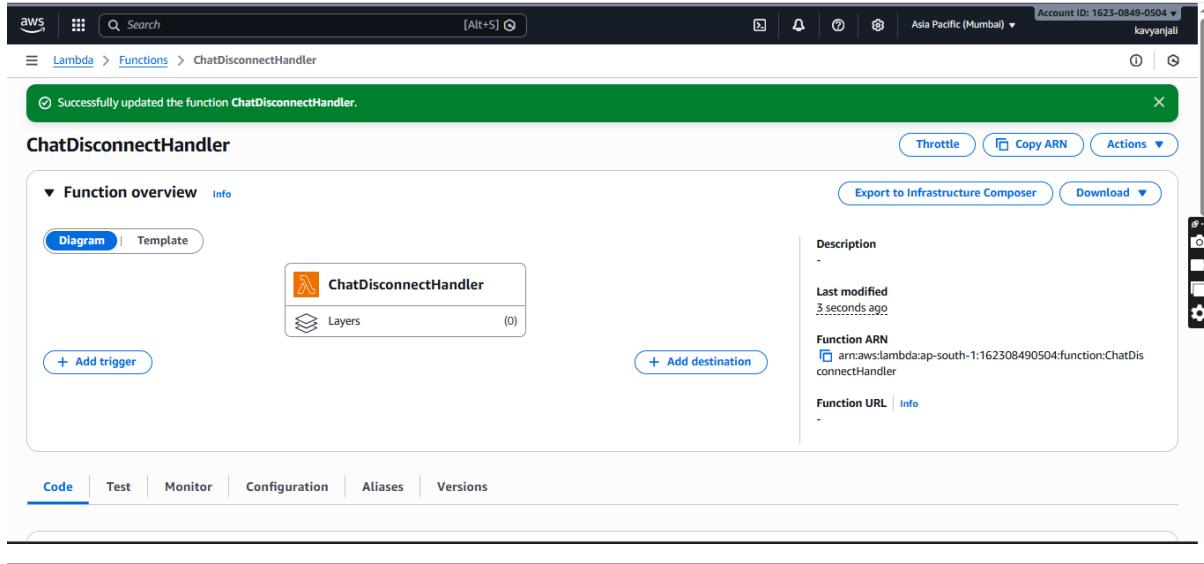
```

import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('ChatConnections')

def lambda_handler(event, context):
    connection_id = event['requestContext']['connectionId']
    table.delete_item(Key={'connectionId': connection_id})
    return {'statusCode': 200}

```



4.3 Lambda: Message Handler

- **Function name:** ChatMessageHandler

Code:

```

import boto3, json

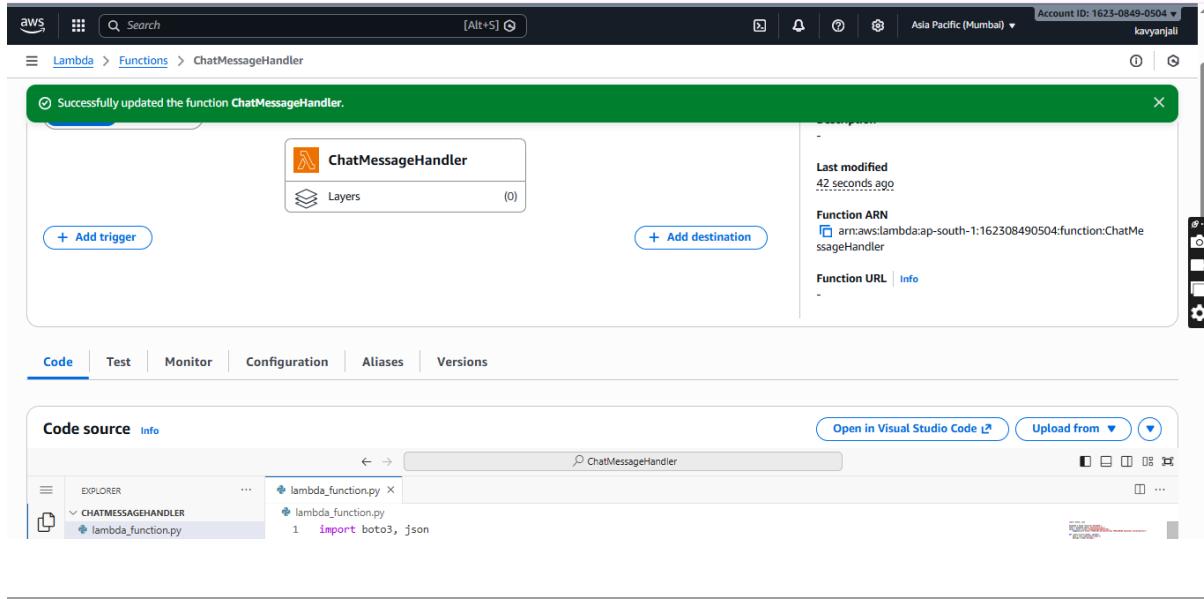
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('ChatConnections')
apigw = boto3.client('apigatewaymanagementapi',
    endpoint_url='wss://lgzv7q7w97.execute-api.ap-south-1.amazonaws.com/production/')

def lambda_handler(event, context):
    body = json.loads(event['body'])
    message = body['message']

    connections = table.scan()['Items']
    for conn in connections:
        apigw.post_to_connection(
            ConnectionId=conn['connectionId'],
            Data=message.encode('utf-8')
        )

    return {'statusCode': 200}

```



STEP 5: Create WebSocket API (API Gateway)

Console Steps:

1. Go to **Services → API Gateway**
2. Click **Create API**
3. Choose **WebSocket API**
4. Click **Build**

WebSocket Settings:

- API name: ChatWebSocketAPI
- Route selection expression: \$request.body.action
- Click **Create API**

WebSocket URL: wss://lgzv7q7w97.execute-api.ap-south-1.amazonaws.com/production/

STEP 6: Configure Routes

Routes to Create:

Route	Lambda
\$connect	ChatConnectHandler
\$disconnect	ChatDisconnectHandler
sendMessage	ChatMessageHandler

Steps:

1. Open **Routes**
2. Click **Create route**
3. Enter route key → Save

4. Attach Lambda integration

The screenshot shows the AWS API Gateway console. A modal window titled "Creating ChatWebSocketAPI..." indicates the process is 100% complete. The main interface shows a "Routes" section with a "connect" route selected. A flow diagram illustrates the request path from a "Client" to a "Route request" step, then to an "Integration request" step, and finally to a "Lambda integration". Below the diagram, tabs for "Route request", "Integration request" (which is selected), "Integration response", and "Route response" are visible.

The screenshot shows the AWS Lambda console. A modal window confirms the successful creation of the "CognitoWebSocketAuthorizer" function. The main interface displays the function overview for "CognitoWebSocketAuthorizer", which has no triggers or destinations. It includes fields for "Description", "Last modified" (7 minutes ago), "Function ARN" (arn:aws:lambda:ap-south-1:162308490504:function:CognitoWebSocketAuthorizer), and a "Function URL" link.

The screenshot shows the AWS API Gateway console under the "Authorizers" section. A modal window titled "CognitoAuthorizer" displays "Authorizer details". Key information includes the "Authorizer ID" (kc0yej), "Lambda function" (CognitoWebSocketAuthorizer (ap-south-1)), and "Identity sources" (Authorization (Header)). Other sections like "Lambda invoke role - optional" and "Lambda event payload" are also shown.

STEP 7: Deploy WebSocket API

1. Go to **Deployments**
2. Click **Create deployment**
3. Stage name: production
4. Click **Deploy**

Copy WebSocket URL:

wss://lgzv7q7w97.execute-api.ap-south-1.amazonaws.com/production/

The screenshot shows the AWS API Gateway console. In the left sidebar, under 'APIs', there's a section for 'API: ChatWebSocketAPI' which includes 'Stages'. The 'production' stage is selected. On the right, the 'Stage details' panel shows the 'Stage name' as 'production', 'Rate Info' as '10000', and 'Burst Info' as '5000'. Below this, the 'WebSocket URL' is listed as 'wss://lgzv7q7w97.execute-api.ap-south-1.amazonaws.com/production/' with a green 'Copied' message next to it. Other sections like '@connections URL' and 'Active deployment' are also visible.

STEP 8: Enable Cognito Authorizer

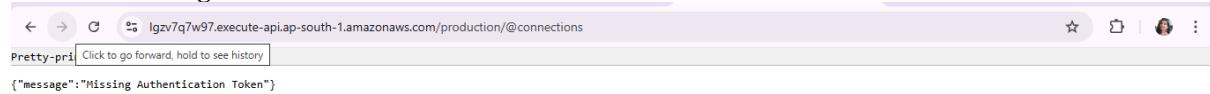
1. Go to **API Gateway → Authorizers**
2. Create **JWT Authorizer**
3. Identity source: Authorization
4. Issuer URL: Cognito User Pool URL
5. Audience: App Client ID
6. Attach to \$connect route

STEP 9: Enable Logging (CloudWatch)

1. Go to **API Gateway → Stages**
2. Select production
3. Enable **CloudWatch Logs**
4. Log level: INFO

The screenshot shows the AWS API Gateway 'Settings' page. A green success message at the top states 'Successfully updated CloudWatch log role ARN to 'arn:aws:iam::162308490504:role/APIGatewayCloudWatchLogsRole''. The 'Logging' section contains a 'CloudWatch log role ARN' field with the value 'arn:aws:iam::162308490504:role/APIGatewayCloudWatchLogsRole'. The 'Throttling' section shows an account-level throttling rate of 10000 requests per second with a burst of 5000 requests. The left sidebar includes sections for APIs, Usage plans, and Developer portals.

STEP 10: Testing



Pretty-print Click to go forward, hold to see history

```
{"message": "Missing Authentication Token"}
```

Security & Scalability Notes

- Serverless auto-scales
 - No server management
 - Cognito secures users
 - DynamoDB handles millions of connections
-

Skills Demonstrated

- WebSocket lifecycle management
 - Event-driven architecture
 - Identity federation with Cognito
 - Stateless Lambda design
 - Real-time serverless communication
-

Conclusion

This project implements a fully serverless real-time chat application using AWS WebSocket APIs, Lambda, DynamoDB, and Cognito. It demonstrates scalable event-driven design, secure identity management, and efficient state handling without maintaining servers.
