# Software Cloud Computing - AWS Task Management System Implementation Documentation


| | |
|---|---|
| Maryam Kashif | 11001929 |
| Ahmed Hegab | 10005393 |
| Ammar Hassona | 10006003 |
| Ahmed Ashraf | 10002329 |
| Ali Sherif | 10000719 |
| Marina Emil | 10000709 |
| Sandra Narmer | 10002592 |
| Clara Amir | 10002591 |


23/05/2025

## 1 Cognito Setup Implementation

### 1.1 User Pool

A Cognito User Pool was created to manage user accounts and handle sign-up and sign-in flows.

- **Pool Name:** User pool - pla9hw
- **Attributes:** Email, Password, Username
- **Password Policy:** Minimum length of 8 characters, with requirements for numbers and special characters.

### 1.2 App Client

- **App Client Name:** Task Manager Application
- **Generate Client Secret:** Yes
- **Callback URLs:** Set for frontend application
- **Sign-out URLs:** Defined for logout redirects

### 1.3 User Attributes

- `email` – required and must be unique
- `username` – custom
- `sub` – a UUID automatically generated for each user upon registration

# 2 Integration with RDS

## 2.1 Cognito Sub in RDS

Each Cognito user has a unique identifier called `sub` (UUID). This identifier is stored in the RDS `users` table to map Cognito users to application-specific user data.

```
CREATE TABLE users (
    cognito_sub UUID PRIMARY KEY,
    username VARCHAR(100),
    email VARCHAR(50) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

## 2.2 Authentication Flow

1. The user signs up or signs in via the web interface using Amazon Cognito.
2. Cognito returns an authentication token and a UUID (`sub`) for the user.
3. A Lambda function sends the UUID, email, and username to the RDS database.
4. All subsequent user-related data references this UUID.

# 3 Security and Best Practices

- Use HTTPS for all API communications with Cognito.
- Never store passwords manually—Cognito manages them securely.
- Validate JWT tokens on the backend to ensure authenticated access.
- Set token expiration and refresh settings appropriately.
- Use IAM roles and policies to restrict access based on identity claims.

# 4 RDS and PostgreSQL Documentation

## 4.1 Overview

This section describes the setup of a PostgreSQL database hosted on Amazon RDS, the connection using pgAdmin, and the creation of two relational tables: `users` and `tasks`.

# 5 AWS RDS Setup

## 5.1 Database Instance

Amazon RDS was used to host a PostgreSQL database instance. Configuration:

- Database Engine: PostgreSQL

- Instance Class: db.t4g.micro
- Storage: 20 GB
- VPC: Ensure EC2 and RDS are in the same VPC
- Private Access: To ensure database security
- Availability Zone: Ensure EC2 and RDS are on the same AZ to avoid extra charges
- Security Group: EC2 and RDS are connected using an EC2 connection with an RDS-EC2 security group that allows inbound PostgreSQL protocol from the RDS

# 6 Connecting to RDS with pgAdmin

## 6.1 pgAdmin Configuration

To connect to the RDS database:

- Launch pgAdmin.
- Create a new server connection.
- Enter the RDS endpoint as the host.
- Use the master username and password created in the AWS RDS setup.
- Confirm that the port is 5432 and the database name matches the RDS database name.

# 7 Database Schema

## 7.1 Users Table

The `users` table stores user profile information.

```
CREATE TABLE users (
    cognito_sub UUID PRIMARY KEY,
    username VARCHAR(100),
    email VARCHAR(100) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

## 7.2 Tasks Table

The `tasks` table stores task information associated with a user. Each task is linked to a user via a foreign key.

```
CREATE TABLE tasks (
    task_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    title VARCHAR(255) NOT NULL,
    description TEXT,
    status VARCHAR(50) DEFAULT 'pending',
```

```
    attachments TEXT[],
    priority VARCHAR(50),
    cognito_sub UUID REFERENCES users(cognito_sub) ON DELETE CASCADE,
);
```

### 7.3 Design Choices

- **UUID Primary Keys:** Universally unique identifiers are used for both users and tasks to prevent collisions and as an additional security measure.
- **Foreign Key with Cascade Delete:** Tasks are automatically deleted if the corresponding user is deleted.

## 8 DynamoDB Overview

This report describes the creation and structure of a DynamoDB table named `Tasks`, designed to store task-related data in a serverless environment. It supports scalable performance and flexible schema design while maintaining high availability.

## 9 AWS DynamoDB Setup

### 9.1 Table Configuration

The DynamoDB table was created with the following configuration:

- **Table Name:** Tasks
- **Primary Key:** `taskId` (Partition key)
- **Billing Mode:** Pay-per-request (on-demand)
- **Region:** eu-north-1
- **Encryption:** AWS owned CMK
- **Stream:** Disabled
- **Backup:** Enabled (on-demand backup)

## 10 Task Entry Sample

| Attribute Name | Type | Description |
|---|---|---|
| `taskId` | String (PK) | Unique identifier for the task. |
| `activity_log` | Map/List | Record of user activity on the task. |
| `attachments` | List | Array of URLs or filenames attached to task. |
| `last_viewed` | String | ISO timestamp of the last view. |
| `priority_score` | Number | A numeric value representing task priority. |
| `tags` | List | Tags or labels associated with the task. |
| `view_count` | Number | Number of times the task was viewed. |
| `created_at` | String | ISO timestamp of when the task was created. |
| `updated_at` | String | ISO timestamp of the last update. |

**Example entry:**

```
{
  "taskId": "T12345",
  "activity_log": [],
  "attachments": [],
  "last_viewed": "2025-05-09T13:50:00Z",
  "priority_score": 0.92,
  "tags": [],
  "view_count": 12,
  "created_at": "2025-05-01T09:00:00Z",
  "updated_at": "2025-05-09T13:50:00Z"
}
```

# 11 Design Choices

- **Flexible Schema:** Allows dynamic addition of attributes per item.
- **ISO Timestamps:** Simplifies sorting, comparison, and display.
- **Partition Key Only:** Optimized for single-key lookup operations.
- **Scalable Access:** Suitable for workloads with unpredictable access patterns.

# 12 SQS and SNS Implementation Documentation

## 12.1 Overview

This document provides a comprehensive overview of the implementation of Amazon SQS (Simple Queue Service) and SNS (Simple Notification Service) in the Task Management System project.

## 12.2 SNS (Simple Notification Service)

### Objective

SNS is used to broadcast task update notifications to subscribers, which in this case is an SQS queue.

### Steps Implemented

- **Topic Creation:**
  - SNS Topic named `TaskNotificationTopic` was created to handle task update notifications
  - Topic ARN: `arn:aws:sns:eu-north-1:183631305334:TaskNotificationTopic`

## 12.3 SQS (Simple Queue Service)

### Objective

SQS is used to queue task update notifications and ensure reliable message processing.

**Steps Implemented**

- **Queue Creation:**
  - SQS Queue named `TaskNotificationQueue` was created
  - Queue ARN: `arn:aws:sqs:eu-north-1:183631305334:TaskNotificationQueue`
- **Configuration Settings:**
  - **Visibility Timeout:** 30 seconds – Duration message remains hidden after retrieval
  - **Message Retention Period:** 4 days – Automatic deletion after retention period
  - **Delivery Delay:** 0 seconds – Immediate message delivery
  - **Maximum Message Size:** 256 KB – Size limit per message
  - **Receive Message Wait Time:** 0 seconds – Short polling enabled

# 13 S3-Bucket Implementation Documentation

## 13.1 Overview

Amazon S3 bucket named `task-manager-file-storage` serves as a centralized, durable, and scalable file storage solution. It is used to store, manage, and retrieve files uploaded by the task management application, including:

- Task-related attachments or documents
- User-uploaded files
- System-generated exports or logs

## Properties

- **Bucket name:** task-manager-file-storage
- **AWS region:** Europe (Stockholm) eu-north-1
- **ARN:** arn:aws:s3:::task-manager-file-storage
- **Creation date:** May 12, 2025, 19:40:11 (UTC+03:00)

## 13.2 Objects

- **Name:** tasks/
- **Type:** Folder

## 13.3 Properties

**Bucket Overview**

- **Encryption:** Server-side encryption with Amazon S3 managed keys (SSE-S3)

### 13.4 Bucket Policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::task-manager-file-storage/*"
        }
    ]
}
```

Listing 1: S3 Bucket Policy

### 13.5 CORS Policy

```
[
    {
        "AllowedHeaders": ["*"],
        "AllowedMethods": ["GET", "PUT", "POST"],
        "AllowedOrigins": ["http://51.21.128.110"],
        "ExposeHeaders": ["ETag"],
        "MaxAgeSeconds": 3000
    }
]
```

Listing 2: CORS Configuration

## 14 Lambda Functions Implementation

### 14.1 Overview

This section provides a comprehensive overview of the implementation of Amazon Lambda functions and their corresponding API gateways in the **Task Management System** project using AWS services.

All task Lambda functions include their respective `package.json`, `package-lock.json`, `node_modules` folder, and a `database` folder containing configuration modules for AWS services such as the aws-config.js file and the rds-config.js.

### 14.2 AWS Configuration Module: `aws-config.js`

The `aws-config.js` file serves as a centralized configuration and utility module that streamlines integration with AWS services using the AWS SDK v3 for JavaScript. It enhances secure and scalable cloud operations by leveraging environment variables and modular service clients.

**1. Environment and Utilities**

- `require('dotenv').config();` – Loads environment variables from a `.env` file into `process.env`.

- `uuidv4` – Imported from the `uuid` package to generate universally unique identifiers (UUIDs).

**2. Amazon S3**

- `s3Client` – Initialized with a specified AWS region to interact with S3 buckets.
- `PutObjectCommand` – Used to upload files or data to S3.

**3. Amazon SQS**

- `sqsClient` – Configured to connect to SQS for message queuing.
- Commands: `ReceiveMessageCommand`, `DeleteMessageCommand`.

**4. Amazon SNS**

- `snsClient` – Interacts with SNS for notifications and messaging.
- Commands: `PublishCommand`, `SubscribeCommand`, `ListSubscriptionsByTopicCommand`.

**5. Amazon DynamoDB**

- `dynamoDB` – Instance of `DynamoDBDocumentClient` for simplified access to DynamoDB.
- Commands: `PutCommand`, `ScanCommand`, `GetCommand`, `UpdateCommand`, `DeleteCommand`.

**6. Amazon SES**

- `sesClient` – Used to send emails via Simple Email Service.
- Command: `SendEmailCommand`.

**Module Snippet**

```
require('dotenv').config();
const { v4: uuidv4 } = require('uuid');

const { S3Client, PutObjectCommand } = require('@aws-sdk/client-s3');
const { SQSClient, ReceiveMessageCommand, DeleteMessageCommand } =
   require('@aws-sdk/client-sqs');
const { SNSClient, PublishCommand, SubscribeCommand,
   ListSubscriptionsByTopicCommand } = require('@aws-sdk/client-sns');
const { DynamoDBClient } = require('@aws-sdk/client-dynamodb');
const { DynamoDBDocumentClient, PutCommand, ScanCommand, GetCommand,
   UpdateCommand, DeleteCommand } = require('@aws-sdk/lib-dynamodb');
const { SESClient, SendEmailCommand } = require('@aws-sdk/client-ses');

const region = process.env.AWS_REGION;

module.exports = {
  uuidv4,
  s3Client: new S3Client({ region }),
```

```
  PutObjectCommand,
  sqsClient: new SQSClient({ region }),
  ReceiveMessageCommand,
  DeleteMessageCommand,
  snsClient: new SNSClient({ region }),
  PublishCommand,
  SubscribeCommand,
  ListSubscriptionsByTopicCommand,
  dynamoDB: DynamoDBDocumentClient.from(new DynamoDBClient({ region }))
     ,
  PutCommand,
  ScanCommand,
  GetCommand,
  UpdateCommand,
  DeleteCommand,
  sesClient: new SESClient({ region }),
  SendEmailCommand,
};
```

Listing 3: aws-config.js

## 14.3 PostgreSQL RDS Configuration: `rds-config.js`

The `rds-config.js` file configures secure access to a PostgreSQL database hosted on Amazon RDS using the `pg` library. It defines a reusable connection pool that supports efficient query execution throughout the application.

**1. Environment Configuration**

- `require('dotenv').config();` – Loads secure environment variables for database connection.

**2. Pool Initialization**

- Uses the `Pool` class from `pg` to manage PostgreSQL connections.
- Parameters:
  - `host` – RDS endpoint address.
  - `port` – Database listening port.
  - `user, password` – Credentials for access.
  - `database` – Name of the database, defaults to `TaskManagerDB`.
  - `ssl` – Enables encrypted connections; disables certificate rejection for development.

**Module Snippet**

```
require('dotenv').config();
const { Pool } = require('pg');

const pool = new Pool({
  host:     process.env.DB_HOST,
```

```
  port:     Number(process.env.DB_PORT),
  user:     process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME || 'TaskManagerDB',
  ssl:      { rejectUnauthorized: false }
});

module.exports = pool;
```

Listing 4: rds-config.js

## 14.4 Lambda Function: `createTask.js`

### 14.4.1 Overview

The `createTask.js` Lambda function is responsible for handling the creation of a new task in the Task Management System. It integrates multiple AWS services including S3, PostgreSQL (via RDS), DynamoDB, and SNS to support file storage, metadata logging, database persistence, and notification broadcasting.

### 14.4.2 Function ARN

- Function ARN: arn:aws:lambda:eu-north-1:183631305334:function:createTask

### 14.4.3 Code Properties

- Package size: 16.7 MB
- SHA256 hash: `I7b6OfAtpX17QnKl6CQ169GMI/UrBtgWfD20LuCcex8=`
- Description: A starter AWS Lambda function.
- Timeout: 0 min 15 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.4.4 Runtime Settings

- Runtime: Node.js 18.x
- Handler: `createTask.handler`
- Architecture: `x86_64`
- Runtime version ARN: `arn:aws:lambda:eu-north-1::runtime:bcd7c0a83c1c6bc9b4982403f8200084d0db1fe61d7c4c`

#### 14.4.5 VPC Configuration

- VPC: `vpc-0b4346ff737a35d6a (10.1.0.0/16)` — `taskmanager-dev-vpc-01-vpc`
- IPv6 Traffic Allowed: `false`
- Subnets:
  - `subnet-0c207a76a62a126e9 (10.1.102.0/24)` — eu-north-1b, `taskmanager-dev-vpc-01-`
  - `subnet-0ed2c4940f659476a (10.1.101.0/24)` — eu-north-1a, `taskmanager-dev-vpc-01-`

#### 14.4.6 Inbound Rules

- Security Group ID: `sg-021b9a2b45c93ddf2`
- Protocol: Custom TCP
- Ports: 443
- Source: `sg-021b9a2b45c93ddf2`

#### 14.4.7 Permissions

- Role Name: `taskmanager-exec-role`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`

#### 14.4.8 Trigger: API Gateway

- API Name: `TasksAPI`
- ARN: `arn:aws:execute-api:eu-north-1:183631305334:w0dkkllt18/*/POST/tasks`
- API Endpoint: `https://w0dkkllt18.execute-api.eu-north-1.amazonaws.com/pre-prod/tas`
- **API Type:** REST
- Authorization: NONE
- isComplexStatement: No
- Method: POST
- Resource Path: `/tasks`

- Service Principal: `apigateway.amazonaws.com`
- Stage: pre-prod
- Statement ID: `62181ade-c891-5ac2-b748-c240141a6da9`

#### 14.4.9 Functionality Workflow

The Lambda function follows these main steps:

1. **CORS Preflight Handling:** The function returns an appropriate response if the HTTP method is `OPTIONS`, enabling CORS support.
2. **Input Validation:** The function parses the incoming JSON payload to extract required fields such as `title` and `userId`, and throws an error if they are missing.
3. **Attachment Upload (Amazon S3):**
   - Each attachment is decoded from Base64 format.
   - Files are uploaded to a predefined S3 bucket using a unique key generated via UUID.
   - Public URLs are returned for each uploaded file.
4. **PostgreSQL Insertion (Amazon RDS):**
   - A new task record is inserted into the `tasks` table under the appropriate schema.
   - The record includes task metadata and the S3 URLs of any attachments.
5. **DynamoDB Logging and SNS Notification:**
   - The user's email is retrieved from the PostgreSQL `users` table.
   - A task metadata record is created in DynamoDB, which includes an activity log, view count, and tags.
   - A notification is published to an SNS topic to inform subscribers of the new task creation.
6. **Response:** A JSON response is returned containing the created task's details, including the task ID and file URLs.

#### 14.4.10 Environment Variables

The function uses the following environment variables:

- `S3_BUCKET`: The name of the S3 bucket used for file storage.
- `DB_TASK`: The name of the PostgreSQL task table.
- `DB_SCHEMA_TASK`: The schema in which the task table resides.
- `SNS_TOPIC_ARN`: The ARN of the SNS topic to publish task notifications.
- `DYNAMODB_TASK_TABLE`: The name of the DynamoDB table storing task metadata.

#### 14.4.11 Key AWS Services Utilized

- **Amazon S3:** Stores uploaded attachments with secure URLs.
- **Amazon RDS (PostgreSQL):** Main relational data store for task metadata.
- **Amazon DynamoDB:** Stores task activity logs and metadata for fast, scalable access.
- **Amazon SNS:** Publishes notifications to subscribed clients or systems.

#### 14.4.12 Error Handling

All operations are wrapped in a `try-catch` block to capture runtime errors. In case of failure, a `500 Internal Server Error` response is returned along with the error message.

#### 14.4.13 CORS Configuration

CORS headers are set for all responses to allow cross-origin POST and OPTIONS requests. This ensures the Lambda function can be safely invoked from frontend applications hosted on different domains.

#### 14.4.14 Security Considerations

- Environment variables are loaded securely from a environment variables configuration..
- Uploaded files are not made public by default.
- SQL queries use parameterized inputs to prevent SQL injection.

### 14.5 Lambda Function: `getAllTasks.js`

#### 14.5.1 Overview

The `getAllTasks.js` Lambda function retrieves all tasks associated with a specific user from the Task Management System. It fetches task data from a PostgreSQL database and enriches each task with metadata from DynamoDB. The function is secured using AWS Cognito-based authorization and returns a structured JSON response.

#### 14.5.2 Function ARN

- arn:aws:lambda:eu-north-1:183631305334:function:getAllTasks

#### 14.5.3 Code Properties

- Package size: 22.1 MB
- SHA256 hash: `/yoxuM/aOnXvjvcAdkRWnAXox5v+VbFzaloIgHssyws=`
- Timeout: 0 min 10sec

- Memory: 128 MB
- Ephemeral storage: 512 MB

#### 14.5.4 Runtime Settings

- Runtime: Node.js 22.x
- Handler: `getAllTasks.handler`
- Architecture: `x86_64`
- Runtime version ARN:
  `arn: arn:aws:lambda:eu-north-1::runtime:fd2e05b324f99edd3c6e17800b2535deb79bcce`

#### 14.5.5 VPC Configuration

- VPC: `vpc-0b4346ff737a35d6a (10.1.0.0/16)` — `taskmanager-dev-vpc-01-vpc`
- IPv6 Traffic Allowed: `false`
- Subnets:
  - `subnet-0c207a76a62a126e9 (10.1.102.0/24)` — eu-north-1b, `taskmanager-dev-vpc-01-`
  - `subnet-0ed2c4940f659476a (10.1.101.0/24)` — eu-north-1a, `taskmanager-dev-vpc-01-`

#### 14.5.6 Inbound Rules

- Security Group ID: `sg-021b9a2b45c93ddf2`
- Protocol: Custom TCP
- Ports: 443
- Source: `sg-021b9a2b45c93ddf2`

#### 14.5.7 Permissions

- Role Name: `taskmanager-exec-role`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`

#### 14.5.8 Trigger: API Gateway

- API Name: `TasksAPI`
- ARN: `arn:aws:execute-api:eu-north-1:183631305334:w0dkkllt18/*/GET/tasks`
- API Endpoint: : `https://w0dkkllt18.execute-api.eu-north-1.amazonaws.com/pre-prod/`
- **API Type:** REST
- Authorization: Cognito JWT (via `event.requestContext.authorizer`)
- isComplexStatement: No
- Method: GET
- Resource Path: `/tasks`
- Service Principal: `apigateway.amazonaws.com`
- Stage: pre-prod
- Statement ID: `7ea2edb2-2fc0-57e7-89a7-0a84ca3fc646`

#### 14.5.9 Functionality Workflow

The Lambda function follows these main steps:

1. **User Authorization:**
   - Extracts the user's Cognito `sub` identifier from the request context.
   - Returns a `401 Unauthorized` response if the identity is not found.
2. **Task Retrieval (Amazon RDS):**
   - Queries the PostgreSQL `tasks` table under the specified schema.
   - Filters tasks by the authenticated user's `cognito_sub`.
3. **Metadata Enrichment (DynamoDB):**
   - Fetches metadata for each task from the DynamoDB table.
   - Merges the metadata with task details.
4. **Response:**
   - Returns a structured JSON array of enriched tasks.
   - Handles empty arrays gracefully.

#### 14.5.10 Environment Variables

- `DB_TASK`: The PostgreSQL task table name.
- `DB_SCHEMA_TASK`: The schema where the task table resides.
- `DYNAMODB_TASK_TABLE`: The name of the DynamoDB table storing task metadata.

#### 14.5.11 Key AWS Services Utilized

- **Amazon RDS (PostgreSQL):** Source of all task data.
- **Amazon DynamoDB:** Source of task metadata including activity logs and tags.
- **Amazon Cognito:** Used for authenticating and identifying users via JWT tokens.

#### 14.5.12 Error Handling

All operations are wrapped in a `try-catch` block to manage runtime exceptions. In the event of failure, a `500 Internal Server Error` response is returned with a message describing the error.

#### 14.5.13 CORS Configuration

While commented out by default, CORS headers can be enabled to support cross-origin access from web applications.

#### 14.5.14 Security Considerations

- Only authenticated users can access their own tasks.
- Data is fetched from DynamoDB using the task's unique ID, ensuring least privilege access.

### 14.6 Lambda Function: `getTask.js`

#### 14.6.1 Overview

The `getTask.js` Lambda function retrieves the details of a single task by its unique `taskId`. It validates the input, queries both Amazon RDS (PostgreSQL) and DynamoDB, and returns a unified object containing task data and metadata. It also updates the metadata to record the view count and timestamp of the latest access.

#### 14.6.2 Function ARN

- Function ARN: arn:aws:lambda:eu-north-1:183631305334:function:GetTask

#### 14.6.3 Code Properties

- Package size: 21.2 MB
- SHA256 hash: `M+8lKJW07fpzn2R7xBkuvW2e1NF2F5sefwrjMf+MeRo=`
- Description: Retrieves a task by `taskId`, updates its metadata, and returns enriched task info.
- Timeout: 0 min 3 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

#### 14.6.4 Runtime Settings

- Runtime: Node.js 18.x
- Handler: `getTask.handler`
- Architecture: `x86_64`
- Runtime version ARN: `arn:aws:lambda:eu-north-1::runtime:5ab9f87f341aef8dc321668fd30e7a28cd061413c8b91a`

#### 14.6.5 VPC Configuration

- VPC: `vpc-0b4346ff737a35d6a (10.1.0.0/16)` — `taskmanager-dev-vpc-01-vpc`
- IPv6 Traffic Allowed: `false`
- Subnets:
  - `subnet-0c207a76a62a126e9 (10.1.102.0/24)` — eu-north-1b, `taskmanager-dev-vpc-01-`
  - `subnet-0ed2c4940f659476a (10.1.101.0/24)` — eu-north-1a, `taskmanager-dev-vpc-01-`

#### 14.6.6 Inbound Rules

- Security Group ID: `sg-021b9a2b45c93ddf2`
- Protocol: Custom TCP
- Ports: 443
- Source: `sg-021b9a2b45c93ddf2`

#### 14.6.7 Permissions

- Role Name: `GetTask-role-41lkiwyp`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`

#### 14.6.8 Trigger: API Gateway

- API Name: `TasksAPI`
- ARN: `arn:aws:execute-api:eu-north-1:183631305334:w0dkkllt18/*/GET/tasks/*`
- API Endpoint: `https://w0dkkllt18.execute-api.eu-north-1.amazonaws.com/pre-prod/tas`
- **API Type:** REST
- Authorization: None (assumes public or protected by other means)
- Method: GET
- Resource Path: `/tasks/{id}`
- Service Principal: `apigateway.amazonaws.com`
- Stage: pre-prod
- Statement ID: `8fdfc05f-8ee6-592a-8645-95802103cb82`

#### 14.6.9 Functionality Workflow

The Lambda function performs the following steps:

1. **Path Parameter Validation:**
   - Extracts `taskId` from the API Gateway path parameters.
   - Returns `400 Bad Request` if the parameter is missing.
2. **Metadata Fetch (DynamoDB):**
   - Checks if the task exists in DynamoDB.
   - Returns `404 Not Found` if metadata is not found.
3. **Task Fetch (PostgreSQL):**
   - Queries RDS for task details using the provided `taskId`.
   - Returns `404 Not Found` if no task is located.
4. **Metadata Update (DynamoDB):**
   - Updates the view count, last viewed timestamp, and appends a log entry to the activity log.
   - Returns updated metadata with the final response.
5. **Return Response:**
   - Combines PostgreSQL task data with updated metadata.
   - Returns `200 OK` with the result as a JSON object.

### 14.6.10 Environment Variables

- `DB_TASK`: Name of the PostgreSQL task table.
- `DB_SCHEMA_TASK`: Schema name for the task table.
- `DYNAMODB_TASK_TABLE`: DynamoDB table name used to store task metadata.

### 14.6.11 Key AWS Services Utilized

- **Amazon RDS (PostgreSQL):** Primary store for core task details.
- **Amazon DynamoDB:** Holds metadata like view count, timestamps, and activity logs.
- **Amazon API Gateway:** REST interface exposing this function at the route `/tasks/{id}`.

### 14.6.12 Error Handling

All operations are enclosed in a `try-catch` block. Expected error responses include:

- `400 Bad Request`: Missing `taskId`.
- `404 Not Found`: Task not found in either DynamoDB or PostgreSQL.
- `500 Internal Server Error`: On unexpected failures.

## 14.7 CORS Configuration

CORS headers are enabled explicitly to allow browser-based access.

### 14.7.1 Security Considerations

- Minimal privilege access: DynamoDB updates and RDS queries are scoped to specific taskId.
- Input is validated for existence and correct structure.
- All dynamic expressions are parameterized to mitigate injection attacks.

## 14.8 Lambda Function: `deleteTask.js`

### 14.8.1 Overview

The `deleteTask.js` Lambda function deletes a task by its unique `taskId`. It validates the input, checks the existence of the task in both Amazon RDS (PostgreSQL) and DynamoDB, deletes the task from both data sources, and sends a notification through Amazon SNS to inform downstream services of the deletion.

### 14.8.2 Function ARN

- Function ARN: arn:aws:lambda:eu-north-1:183631305334:function:deleteTask

### 14.8.3 Code Properties

- Package size: 22 MB
- SHA256 hash: `2Or2XTYtSNIicG75Lk6qiHweBIeqi/2N0ARMRghVft4=`
- Description: Deletes a task from both PostgreSQL and DynamoDB and notifies via SNS.
- Timeout: 0 min 15 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.8.4 Runtime Settings

- Runtime: Node.js 22.x
- Handler: `deleteTask.handler`
- Architecture: `x86_64`
- Runtime version ARN: `arn:aws:lambda:eu-north-1::runtime:fd2e05b324f99edd3c6e17800b2535deb79bcce74b750`

### 14.8.5 VPC Configuration

- VPC: `vpc-0b4346ff737a35d6a (10.1.0.0/16)` — `taskmanager-dev-vpc-01-vpc`
- IPv6 Traffic Allowed: `false`
- Subnets:
  - `subnet-0c207a76a62a126e9 (10.1.102.0/24)` — eu-north-1b, `taskmanager-dev-vpc-01-`
  - `subnet-0ed2c4940f659476a (10.1.101.0/24)` — eu-north-1a, `taskmanager-dev-vpc-01-`

### 14.8.6 Inbound Rules

- Security Group ID: `sg-021b9a2b45c93ddf2`
- Protocol: Custom TCP
- Ports: 443
- Source: `sg-021b9a2b45c93ddf2`

#### 14.8.7 Permissions

- Role Name: `taskmanager-exec-role`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`

#### 14.8.8 Trigger: API Gateway

- API Name: `TasksAPI`
- ARN: `arn:aws:execute-api:eu-north-1:183631305334:w0dkkllt18/*/DELETE/tasks/*`
- API Endpoint: `https://w0dkkllt18.execute-api.eu-north-1.amazonaws.com/pre-prod/t`
- **API Type:** REST
- Authorization: None
- Method: DELETE
- Resource Path: `/tasks/{id}`
- Service Principal: `apigateway.amazonaws.com`
- Stage: pre-prod
- Statement ID: `a5987b33-50a3-5df6-bbe7-834b48d0f764`

#### 14.8.9 Functionality Workflow

The Lambda function performs the following steps:

1. **Path Parameter Validation:**
   - Extracts `taskId` from the API Gateway path parameters.
   - Returns `400 Bad Request` if the parameter is missing.
2. **Cognito Sub Lookup (PostgreSQL):**
   - Queries RDS to find the `cognito_sub` associated with the task.
   - Returns `404 Not Found` if task does not exist.
3. **Optional Email Fetch:**

- Queries the users table to retrieve the email address linked to the `cognito_sub`.

4. **Existence Check (DynamoDB):**
   - Confirms the task exists in the DynamoDB metadata table.
   - Returns `404 Not Found` if missing.
5. **Publish Deletion Event (SNS):**
   - Publishes a JSON message to the SNS topic including the user ID, email, and task ID.
6. **Delete Task (DynamoDB and PostgreSQL):**
   - Deletes the task from DynamoDB using `DeleteCommand`.
   - Deletes the task from RDS using `DELETE FROM`.
7. **Return Response:**
   - Returns `200 OK` with a confirmation message.

#### 14.8.10 Environment Variables

- `DB_TASK`: Name of the PostgreSQL task table.
- `DB_SCHEMA_TASK`: Schema name for the task table.
- `DYNAMO_TABLE`: DynamoDB table name used to store task metadata.
- `SNS_TOPIC_ARN`: Amazon SNS topic used to notify task deletions.

#### 14.8.11 Key AWS Services Utilized

- **Amazon RDS (PostgreSQL):** Stores task data and user metadata.
- **Amazon DynamoDB:** Stores metadata for tasks.
- **Amazon SNS:** Sends task deletion notifications to subscribed systems.
- **Amazon API Gateway:** REST interface exposing this function at the route `/tasks/{id}`.

#### 14.8.12 Error Handling

All operations are enclosed in a `try-catch` block. Expected error responses include:

- `400 Bad Request`: Missing `taskId`.
- `404 Not Found`: Task not found in DynamoDB or RDS.
- `500 Internal Server Error`: On unexpected failures.

#### 14.8.13 CORS Configuration

CORS headers are enabled explicitly to allow browser-based access.

## 14.9 Security Considerations

- Uses parameterized queries to avoid SQL injection.
- Validates input strictly.
- Dynamically verifies existence before attempting deletions.
- Publishes audit messages to SNS for traceability.

## 14.10 Lambda Function: `updateTask.js`

### 14.10.1 Overview

The `updateTask.js` Lambda function updates an existing task by its unique `taskId`. It supports multipart form submissions (for file uploads) and JSON bodies, dynamically merges new attachments with existing ones, updates both Amazon RDS (PostgreSQL) and DynamoDB, and publishes an update event through Amazon SNS.

### 14.10.2 Function ARN

- Function ARN: arn:aws:lambda:eu-north-1:183631305334:function:updateTask

## 14.11 Code Properties

- Package size: 16.6 MB
- SHA256 hash: `KpWa9BSDierb17ESGpn5Fm4KuZ00tR7x0bI0zj8AckU=`
- Description: Updates a task in PostgreSQL and DynamoDB, supports file uploads, and sends an SNS update notification.
- Timeout: 0 min 11 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.11.1 Runtime Settings

- Runtime: Node.js 18.x
- Handler: `updateTask.handler`
- Architecture: `x86_64`
- Runtime version ARN: `arn:aws:lambda:eu-north-1::runtime:bcd7c0a83c1c6bc9b4982403f8200084d0db1fe61d7c4c`

#### 14.11.2 VPC Configuration

- VPC: `vpc-0b4346ff737a35d6a (10.1.0.0/16)` — `taskmanager-dev-vpc-01-vpc`
- IPv6 Traffic Allowed: `false`
- Subnets:
  - `subnet-0c207a76a62a126e9 (10.1.102.0/24)` — eu-north-1b, `taskmanager-dev-vpc-01-`
  - `subnet-0ed2c4940f659476a (10.1.101.0/24)` — eu-north-1a, `taskmanager-dev-vpc-01-`
- Security Group ID: `sg-021b9a2b45c93ddf2`
- Protocol: Custom TCP
- Ports: 443
- Source: `sg-021b9a2b45c93ddf2`

#### 14.11.3 Permissions

- Role Name: `updateTask-role-r2e8ko3n`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`

#### 14.11.4 Trigger: API Gateway

- API Name: `TasksAPI`
- ARN: `arn:aws:execute-api:eu-north-1:183631305334:w0dkkllt18/*/PUT/tasks/*`
- API Endpoint: `https://w0dkkllt18.execute-api.eu-north-1.amazonaws.com/pre-prod/t`
- **API Type:** REST
- Authorization: None
- Method: PUT
- Resource Path: `/tasks/{id}`
- Service Principal: `apigateway.amazonaws.com`
- Stage: pre-prod
- Statement ID: `5df1a624-951e-5e62-a12f-4da012eef5a6`

### 14.11.5 Functionality Workflow

The Lambda function performs the following steps:

1. **CORS Preflight:**
   - Handles `OPTIONS` requests with appropriate CORS headers.
2. **Path Parameter Validation:**
   - Extracts `taskId` from the API path.
   - Returns `400 Bad Request` if not present.
3. **Request Parsing:**
   - Parses either JSON or multipart form-data including file attachments.
   - Returns `415 Unsupported Media Type` if format is invalid.
4. **RDS Lookup:**
   - Fetches existing task to retrieve current attachments and `cognito_sub`.
   - Returns `404 Not Found` if task does not exist.
5. **File Upload (if any):**
   - Uploads new files to S3 and appends URLs to attachment list.
6. **Update Construction:**
   - Merges form fields and new attachments into update payload.
   - Returns `400 Bad Request` if no fields are provided.
7. **Database Updates:**
   - Updates task in RDS using `UPDATE` statement.
   - Optionally updates DynamoDB metadata with `UpdateCommand`.
8. **Publish Update Event (SNS):**
   - Sends update metadata (task ID, user ID, etc.) via SNS.
9. **Return Response:**
   - Returns `200 OK` with confirmation of update.

### 14.11.6 Environment Variables

- `DB_TASK`: Name of the PostgreSQL task table.
- `DB_SCHEMA_TASK`: Schema name for the task table.
- `DYNAMO_TABLE`: Name of the DynamoDB table.
- `SNS_TOPIC_ARN`: Amazon SNS topic for task update events.
- `S3_BUCKET`: S3 bucket for storing attachments.

### 14.11.7 Key AWS Services Utilized

- **Amazon RDS (PostgreSQL):** Stores core task data and metadata.
- **Amazon DynamoDB:** Hosts task metadata for fast access.
- **Amazon SNS:** Publishes task update notifications.
- **Amazon S3:** Stores uploaded attachments linked to tasks.
- **Amazon API Gateway:** Routes HTTP requests to the function.

### 14.11.8 Error Handling

All major operations are enclosed in a `try-catch` block. Expected error responses include:

- `400 Bad Request`: Missing task ID or no fields to update.
- `404 Not Found`: Task not found in RDS.
- `415 Unsupported Media Type`: If content type is not JSON or multipart.
- `500 Internal Server Error`: On unexpected failures.

### 14.11.9 CORS Configuration

CORS headers are enabled to allow cross-origin requests.

### 14.11.10 Security Considerations

- Validates inputs and file types.
- Avoids SQL injection via parameterized queries.
- Prevents overwrites of existing attachments unless explicitly specified.
- Publishes task update event for auditing and downstream system consistency.

## 14.12 Lambda Function: `taskNotifierFunction`

### 14.12.1 Overview

The `taskNotifierFunction` Lambda function processes incoming task notification messages from an Amazon SQS queue. For each message, it extracts the task ID, action performed (e.g., created, updated, deleted), and the recipient's email address, then sends an email notification via Amazon SES to inform the user about the task status update.

### 14.12.2 Function ARN

- Function ARN: `arn:aws:lambda:eu-north-1:183631305334:function:TaskNotifierFunctio`

### 14.12.3 Code Properties

- Package size: 40.7 kB
- SHA256 hash: `H60c2fP0k01stvLodGQkpaMTcAzzMBejuSiEyJtA/1s=`
- Description: Sends task update notifications via Amazon SES based on messages from an SQS queue.
- Timeout: 0 min 10 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.12.4 Runtime Settings

- Runtime: Python 3.9
- Handler: $\texttt{lambda}_{function.lambda_{handler}}Architecture$ : `x86_64`
- Runtime version ARN: arn:aws:lambda:eu-north-1::runtime:c4127de9acf600313fc596a22245bc413de98744c008a2b8a38abb334f663c06

### 14.12.5 Permissions

Role Name: `TaskNotifierFunction-role-b0hr5yp1`

- Role Name: `taskNotifierFunction-role`
- Allowed Actions:
  - `ses:SendEmail`
  - `sqs:ReceiveMessage`
  - `sqs:DeleteMessage`
  - `logs:CreateLogGroup`
  - `logs:CreateLogStream`
  - `logs:PutLogEvents`

### 14.12.6 Trigger: Amazon SQS

- Queue Name: `TaskNotificationQueue`
- Queue ARN: `arn:aws:sqs:eu-north-1:183631305334:TaskNotificationQueue`
- Batch Size: 10
- Event Source Mapping ARN: arn:aws:lambda:eu-north-1:183631305334:event-source-mapping:c233d14e-62d5-436c-8850-ddc0d1655e1d
- Event Source Mapping Enabled: true
- Metrics: None

- On-failure destination: None
- Report batch item failures: No
- UUID: c233d14e-62d5-436c-8850-ddc0d1655e1d

#### 14.12.7 Functionality Workflow

The Lambda function performs the following steps for each SQS message:

1. **Message Parsing:**
   - Parses the JSON message body from the SQS event record.
   - Extracts `email`, `taskId`, and `action` fields.
2. **Email Validation:**
   - Checks if the `email` field is present; if missing, logs a warning and skips sending.
3. **Email Preparation and Sending:**
   - Constructs an email subject indicating the task ID and action.
   - Composes an HTML email body notifying the user of the task update.
   - Sends the email through Amazon SES to the specified recipient.
4. **Logging:**
   - Logs each notification attempt and its success.
5. **Return:**
   - Returns HTTP status code 200 after processing all records.

#### 14.12.8 Environment Variables

- `AWS_REGION`: AWS region for SES client (e.g., `eu-north-1`)
- `SES_SOURCE_EMAIL`: Verified email address used as the source for SES emails (e.g., `ahmed.hegab@student.giu-uni.de`)

#### 14.12.9 Key AWS Services Utilized

- **Amazon SES:** Sends email notifications to users about task status changes.
- **Amazon SQS:** Provides the message queue triggering the Lambda function with task update events.
- **AWS Lambda:** Executes the notification logic on message receipt.

#### 14.12.10 Error Handling

- Logs warnings for messages missing the `email` field and continues processing.
- Exceptions during email sending will cause the Lambda invocation to fail and the message to remain in the queue for retries, depending on the queue's redrive policy.

### 14.12.11 Security Considerations

- The source email used in SES must be verified to prevent spoofing.
- Permissions are scoped to only allow sending emails and reading from the configured SQS queue.
- Input messages are assumed trusted.

## 14.13 Lambda Function: `SESverifier.js`

### 14.13.1 Overview

The `SESverifier.js` Lambda function verifies Amazon SES identities, which can be either email addresses or domains. It checks if the specified identity is already verified and, if not, initiates the appropriate verification process via SES. The function handles verification status queries and triggers new verifications, returning success or error messages accordingly.

### 14.13.2 Function ARN

- Function ARN: `arn:aws:lambda:eu-north-1:183631305334:function:SESVerifier`

### 14.13.3 Code Properties

- Package size: 749 byte
- SHA256 hash: `hCg5IPSkpfONS59BrlD/hXEfn3+J7x8fx5A+NgscaGw=`
- Description: Verifies SES email or domain identities, triggering verification if not already verified.
- Timeout: 0 min 3 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.13.4 Runtime Settings

- Runtime: Python 3.9
- Handler: `lambda_function.lambda_handler`
- Architecture: `x86_64`
- Runtime version ARN: `arn:aws:lambda:eu-north-1::runtime:c4127de9acf600313fc596a22245bc413de98744c008a2`

### 14.13.5 Permissions

- Role Name: `SESverifier-role`
- Allowed Actions:
  - `secretsmanager:GetSecretValue`

### 14.13.6 Functionality Workflow

The Lambda function executes the following steps:

1. **Extract Identity and Type:**
   - Reads `identity` and optional `type` (default "email") from the event payload.
2. **Check Verification Status:**
   - Calls `GetIdentityVerificationAttributes` to determine if the identity is verified.
   - If verified with status `Success`, returns a confirmation message.
3. **Initiate Verification:**
   - If not verified, triggers verification:
     - For `email`, calls `VerifyEmailIdentity`.
     - For `domain`, calls `VerifyDomainIdentity`.
4. **Return Response:**
   - On success, returns `200 OK` with verification initiation details.
   - On error, returns `500 Internal Server Error` with error details.

### 14.13.7 Environment Variables

- `AWS_REGION`: AWS region, e.g., `eu-north-1`

### 14.13.8 Key AWS Services Utilized

- **Amazon SES:** For identity verification operations.
- **AWS CloudWatch Logs:** For logging Lambda execution details.

### 14.13.9 Error Handling

- Returns HTTP 500 with error message on any exceptions.
- Logs errors to CloudWatch Logs for troubleshooting.

### 14.13.10 Security Considerations

- Uses minimal SES permissions scoped to verification actions.
- Handles input carefully to avoid injection or malformed requests.
- Logs sensitive information cautiously.

## 14.14 Lambda Function: `CreateUser.js`

### 14.14.1 Overview

The `CreateUser.js` Lambda function is triggered by an Amazon Cognito Post Confirmation event. It inserts or updates the user's data in a PostgreSQL RDS database upon successful user sign-up confirmation. The function uses a connection pool from the `pg` Node.js package to execute a SQL `INSERT` or `UPDATE` query on the `users_data.users` table.

### 14.14.2 Function ARN

- Function ARN: `arn:aws:lambda:eu-north-1:183631305334:function:CognitoRDS`

### 14.14.3 Code Properties

- Package size: 20.5 MB
- SHA256 hash: `00tX2lRK/Qq6+aAewW08/yGriXyWBf2oHHSt2wMdFko=`
- Description: Inserts or updates Cognito user data in PostgreSQL database after user confirmation.
- Timeout: 0 min 5 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.14.4 Runtime Settings

- Runtime: Node.js 18.x
- Handler: `createUser.handler`
- Architecture: `x86_64`
- Runtime version ARN: `arn:aws:lambda:eu-north-1::runtime:5ab9f87f341aef8dc321668fd30e7a28cd061413c8b91a`

### 14.14.5 Environment Variables

- `DB_HOST` — Database host endpoint
- `DB_PORT` — Database port (default PostgreSQL port 5432)
- `DB_USERNAME` — Database username
- `DB_PASSWORD` — Database password
- `DB_NAME` — Database name (default: `TaskManagerRDS_DB`)

#### 14.14.6 VPC Configuration

- VPC: `vpc-0b4346ff737a35d6a (10.1.0.0/16)` — `taskmanager-dev-vpc-01-vpc`
- IPv6 Traffic Allowed: `false`
- Subnets:
  - `subnet-0c207a76a62a126e9 (10.1.102.0/24)` — eu-north-1b, `taskmanager-dev-vpc-01-`
  - `subnet-0ed2c4940f659476a (10.1.101.0/24)` — eu-north-1a, `taskmanager-dev-vpc-01-`

#### 14.14.7 Inbound Rules

- Security Group ID: `sg-021b9a2b45c93ddf2`
- Protocol: Custom TCP
- Ports: 443
- Source: `sg-021b9a2b45c93ddf2`

#### 14.14.8 Permissions

- Role Name: `GetTask-role-41lkiwyp`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`
- Resource Policy:
  - CSI_PostConfirmation_eu-north-1U8WCGtv8c_CSI_PostConfirmation

#### 14.14.9 Functionality Workflow

1. **Receive Cognito event:**
   - Extracts user attributes `sub`, `email`, and `name` from the event.
2. **Connect to PostgreSQL database:**
   - Establishes a connection pool to the RDS database using environment variables.
3. **Insert or update user record:**

- Executes an `INSERT` query with `ON CONFLICT (email) DO UPDATE` clause.
- Updates `cognito_sub`, `username`, and sets `created_at` timestamp to current time on conflict.

4. **Release database connection:**
   - Properly releases client back to the pool after query execution.
5. **Return event object:**
   - Returns the original event object to allow Cognito to continue processing.
6. **Error handling:**
   - Logs error details and throws error to halt the confirmation process if insertion fails.

#### 14.14.10 Security Considerations

- Use environment variables to securely store database credentials.
- Secure database access with VPC and security groups if applicable.
- Limit IAM role permissions to only necessary CloudWatch and RDS actions.
- Sanitize and parameterize SQL queries (already done using parameterized queries).

#### 14.14.11 Logging

- Logs successful event JSON payload.
- Logs any errors during database operations.

### 14.15 Lambda Function: `exchangeCode.js`

#### 14.15.1 Overview

The `exchangeCode.js` Lambda function serves as an HTTP endpoint (typically behind an API Gateway) to validate Amazon Cognito ID tokens and asynchronously trigger an email identity verification process using another Lambda function (`SESVerifier`). It ensures only valid and authenticated users can proceed by verifying their JWT against Cognito's JSON Web Key Set (JWKS) public keys.

#### 14.15.2 Function ARN

- Function ARN: `arn:aws:lambda:eu-north-1:183631305334:function:BackendLambdaCognito`

### 14.15.3 Code Properties

- Package size: 20.9 MB
- SHA256 hash: `GvZjmdtD4Fpo207JqdqlDbysVqLjlUBYmA7SBZkRdvE=`
- Description: Verifies Cognito JWT tokens and triggers SESVerifier Lambda if successful.
- Timeout: 0 min 50 sec
- Memory: 128 MB
- Ephemeral storage: 512 MB

### 14.15.4 Runtime Settings

- Runtime: Node.js 18.x
- Handler: `exchangeCode.handler`
- Architecture: `x86_64`
- Runtime ARN: `arn:aws:lambda:eu-north-1::runtime:bcd7c0a83c1c6bc9b4982403f8200084c`

### 14.15.5 Permissions

- Role Name: `taskmanager-exec-role`
- Allowed Actions:
  - `application-autoscaling:DeleteScalingPolicy`
  - `application-autoscaling:DeregisterScalableTarget`
  - `application-autoscaling:DescribeScalableTargets`
  - `application-autoscaling:DescribeScalingActivities`
  - `application-autoscaling:DescribeScalingPolicies`
  - `application-autoscaling:PutScalingPolicy`
  - `application-autoscaling:RegisterScalableTarget`

  Resource Policy:

  - `lambda:InvokeFunction`

### 14.15.6 Environment Dependencies

- `jsonwebtoken` — for decoding and verifying JWT tokens.
- `jwk-to-pem` — to convert JWKS keys to PEM format for token validation.
- `aws-sdk` — to invoke the SESVerifier Lambda.

#### 14.15.7 Constants

- REGION: eu-north-1
- USER_POOL_ID: eu-north-1_U8WCGtv8c
- CLIENT_ID: 6thkk9j96oa02djeccritml1gr
- COGNITO_ISSUER: https://cognito-idp.eu-north-1.amazonaws.com/eu-north-1_U8WCGtv8c

#### 14.15.8 CORS Configuration

- Allows origin: http://localhost:3000
- Allows headers: Content-Type, Authorization
- Allows methods: POST, OPTIONS
- Allows credentials: true

#### 14.15.9 Permissions

- Role Name: taskmanager-exec-role
- Allowed Actions:
  - lambda:InvokeFunction on the SESVerifier Lambda

#### 14.15.10 Trigger: API Gateway

- API Name: Cognito
- ARN: rn:aws:execute-api:eu-north-1:183631305334:xppo00vwz7/*/*/auth/callback
- API Endpoint: https://xppo00vwz7.execute-api.eu-north-1.amazonaws.com/prod/auth/ca
- **API Type:** HTTP
- Authorization: None
- Method: ANY
- Resource Path: /auth/callback
- Service Principal: apigateway.amazonaws.com
- Stage: prod
- Statement ID: d73a0ef4-c5b4-55a4-b8fb-a9b072f529a2

#### 14.15.11 Functionality Workflow

1. **CORS Preflight:**
   - Returns a 200 OK response with CORS headers if the method is `OPTIONS`.
2. **JWT Authorization:**
   - Extracts Bearer token from the `Authorization` header.
   - Fetches and caches Cognito JWKS keys (if not already cached).
   - Converts the relevant JWK to PEM and verifies the JWT using the public key.
3. **Token Validation:**
   - Validates the token issuer and audience (`CLIENT_ID`).
   - If valid, extracts user information (`sub`, `email`, `name`).
4. **Invoke SESVerifier Lambda:**
   - Asynchronously invokes the `SESVerifier` Lambda with the decoded email as payload.
5. **Return Success:**
   - Returns a 200 OK response with the authenticated user's data.
6. **Error Handling:**
   - Returns a 401 or 403 error if the token is invalid or SES invocation fails.
   - Logs the error message for debugging.

#### 14.15.12 Security Considerations

- Uses verified JWKS from Cognito for token validation.
- Tokens must match expected issuer and audience (CLIENT_ID).
- Caches JWKS in-memory to reduce HTTP calls.
- Validates token signature and expiry.

#### 14.15.13 Logging

- Logs decoded email and SES invocation status.
- Logs error details for unauthorized or failed operations.

# 15 API Gateway Documentation

## 15.1 Overview

This report describes the setup and configuration of Amazon API Gateway used in the Task Management System to expose backend functionality over HTTP. It supports routing HTTP requests to AWS Lambda functions, integrates with Cognito for user authentication, and exposes RESTful endpoints for managing tasks.

## 15.2 API Gateway Configuration

### 15.2.1 Gateway Type

- **Type:** HTTP API
- **Integration:** AWS Lambda
- **Authorization:** Amazon Cognito Authorizer for secure endpoints
- **CORS:** Enabled for all methods and routes

## 15.3 Routes and Endpoints

### 15.3.1 1. Cognito API (/auth/callback)

Used for handling user authentication flows via Cognito.

- **Route:** `/auth/callback`
- **Methods:** `GET, POST, OPTIONS`
- **Integration:** Lambda function that handles OAuth callback or JWT validation
- **Authorization:** None (Cognito-managed)

## 15.4 2. Task API (/tasks)

Used for creating, retrieving, updating, and deleting tasks.

**Base Path:** `/tasks`

- **GET:** Retrieve a list of tasks for the authenticated user.
- **POST:** Create a new task.
- **OPTIONS:** Preflight support for CORS.
- **Authorization:** Cognito Authorizer required.

**Path with ID:** `/tasks/{id}`

- **GET:** Retrieve a specific task by ID.
- **PUT:** Update a task

- **DELETE:** Delete a task.
- **OPTIONS:** Preflight support for CORS.
- **Authorization:** Cognito Authorizer required.

### 15.5 Integration with Lambda

Each method on the API routes triggers a corresponding AWS Lambda function. These Lambda functions perform business logic (e.g., reading from or writing to RDS or DynamoDB) and return responses back to API Gateway.

### 15.6 CORS Configuration

CORS (Cross-Origin Resource Sharing) is enabled to allow requests from the frontend application.

- **Allowed Origins:** `*` (or restricted to frontend domain)
- **Allowed Methods:** `GET, POST, PUT, DELETE, OPTIONS`
- **Allowed Headers:** `Content-Type, Authorization`

### 15.7 Security

- **Authentication:** API Gateway integrates with Amazon Cognito to enforce secure access to protected endpoints.
- **Token Validation:** JWT access tokens issued by Cognito are validated by the Cognito Authorizer.

## 16 Frontend Implementation and Deployment

### 16.1 Overview

This EC2 instance (`frontend-app-02`) serves as the primary deployment host for the Task Management System's frontend application.

### 16.2 EC2 Instance Properties

- **Instance ARN:** `arn:aws:ec2:eu-north-1:183631305334:instance/i-06e6d026bdca5b09b`
- **Instance ID:** `i-06e6d026bdca5b09b`
- **AMI ID:** `ami-00f34bf9aeacdf007`
- **AMI Name:** al2023-ami-2023.7.20250512.0-kernel-6.1-x86$_6$4**Key Pair:**$Hegab - EC2 - TaskManagerProject$

- **AZ:** eu-north-1a
- **Instance Name:** frontend-app-02

- **Security Groups:**
  - **Inbound Traffic:**
    - `sgr-058d1c216fba0dc0b`
    - `sgr-0d8ebb4be06c2da93`
    - `sgr-0fc9ae25e263946f6`
  - **Outbound Traffic:**
    - `sgr-04bb2736751e32f20`
- **VPC ID:** `vpc-0b4346ff737a35d6a`
- **Subnet ID:** `subnet-07848965f33b9f5db`

## 16.3 File Structure

```
AVIS-TASKMANAGER/
          my-bootstrap-app2/
                node_modules/
                public/
                src/
                      components/
                            Navbar.js
                      pages/
                            Callback.jsx
                            CreateTask.jsx
                            Home.jsx
                            LoginErrorPage.js
                            TaskDetails.jsx
                            Tasks.jsx
                      App.js
                      App.css
                      App.test.js
                      aws-exports.js
                      index.css
                      index.js
                      logo.svg
                      reportWebVitals.js
                      setupTests.js
                .env
                .gitignore
                package-lock.json
                package.json
                README.md
```

## 16.4 Key Frontend Components

### 16.4.1 API Service Integration

```
// src/api.js
import axios from 'axios';

const api = axios.create({
  baseURL: process.env.REACT_APP_API_URL || '',
  // withCredentials: true,  // enable if you later want to send
     cookies automatically
});

api.interceptors.request.use((config) => {
  console.log('    API Request       ', {
    url: `${config.baseURL}${config.url}`,
    method: config.method,
    headers: { ...config.headers }
  });

  // Get the token from cookies or localStorage
  let token;
  const cookieMatch = document.cookie
    .split('; ')
    .find(row => row.startsWith('idToken='));

  if (cookieMatch) {
    token = cookieMatch.split('=')[1];
  } else {
    // Fallback to localStorage if cookie not found
    token = localStorage.getItem('idToken');
  }

  if (token) {
    config.headers = config.headers || {};
    config.headers.Authorization = `Bearer ${token}`;
    console.log('       attached Authorization:', config.headers.
       Authorization);
  } else {
    console.warn('          no idToken found in cookies or localStorage
       ');
    // Don't throw here, let the server respond with 401
  }

  return config;
}, err => Promise.reject(err));

export default api;
```

## 16.5 Authentication Flow

1. User signs up via Cognito-hosted UI
2. JWT tokens stored in localStorage and cookies
3. Subsequent API requests include Authorization header
4. Session management handled by Amplify Auth

# 17 Conclusion

The implemented solution provides:

- Secure PostgreSQL database with proper relational design
- Reliable notification system using SNS/SQS
- Responsive React frontend with Bootstrap styling
- Secure authentication via AWS Cognito
- Scalable serverless backend architecture