

# AWS RDS 활용 실습

## ◆ AWS RDS란?

RDS(Relational Database Service)는 Amazon Web Services(AWS)에서 제공하는 관리형 관계형 데이터베이스 서비스이다.

## ✅ 핵심 개념 요약

항목	설명
목적	데이터베이스의 설치, 운영, 백업, 복구 등을 자동화하여 사용자가 손쉽게 DB를 사용할 수 있게 함
지원 DB	MySQL, PostgreSQL, MariaDB, Oracle, Microsoft SQL Server, Amazon Aurora 등
주요 기능	자동 백업, 장애 복구, 모니터링, 스케일링, 보안 설정 등 제공
사용 방법	AWS 콘솔이나 CLI로 인스턴스 생성 → 애플리케이션에서 엔드포인트로 접속

Aurora and RDS

대시보드

데이터베이스

Query Editor

성능 개선 도우미

스냅샷

Amazon S3에서 내보내기

자동 백업

예약 인스턴스

프록시

서브넷 그룹

파라미터 그룹

옵션 그룹

사용자 지정 엔진 버전

제로 ETL 통합 [신규](#)

이벤트

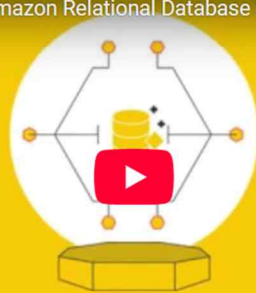
이벤트 구독

작동 방법

aws

Understanding Amazon Relational Database Service (RDS)

Copy link



Aurora









Watch on

YouTube

Amazon RDS는 하드웨어 프로비저닝, 데이터베이스 설정, 패치, 백업과 같은 시간 소모적인 관리 작업을 자동화하는 한편, 비용 효율적이고 확장 가능한 관계형 데이터베이스 용량을 제공합니다. 애플리케이션에 집중하여 애플리케이션에 필요한 빠른 성능,고가용성, 보안 및 호환성을 제공할 수 있도록 지원합니다.

**엔진 옵션**

엔진 유형 [정보](#)

<input type="radio"/> Aurora (MySQL Compatible) 	<input checked="" type="radio"/> Aurora (PostgreSQL Compatible) 
<input type="radio"/> MySQL 	<input type="radio"/> PostgreSQL 
<input type="radio"/> MariaDB 	<input type="radio"/> Oracle 
<input type="radio"/> Microsoft SQL Server 	<input type="radio"/> IBM Db2 

#### ◆ 왜 RDS를 사용할까?

- ☒ DB 설치 및 유지보수 필요 없음 (AWS가 관리)
- ☒ 고가용성(HA) 및 장애 복구 자동화 가능 (Multi-AZ)
- ☒ 자동 백업 및 스냅샷 제공
- ☒ 손쉬운 스케일업/스케일다운
- ☒ IAM 기반 보안 설정 + VPC로 격리된 네트워크 환경

#### ◆ 구성 요소

구성 요소	설명
DB 인스턴스	실제로 돌아가는 DB 서버
DB 엔진	MySQL, PostgreSQL 등 사용자가 선택하는 RDBMS 종류
스토리지	SSD 기반, 자동 확장 가능
백업	자동 또는 수동 백업 설정 가능
보안 그룹	접근 제어를 위한 방화벽 역할 (IP/포트 허용 설정)

## ◆ 사용 방법 요약

1. AWS 콘솔 로그인
2. RDS 서비스 선택
3. DB 엔진 선택 (예: MySQL)
4. DB 인스턴스 사양 설정 (CPU, RAM 등)
5. 스토리지 및 백업 옵션 설정
6. VPC 및 보안 그룹 설정
7. DB 생성 → 엔드포인트 발급
8. 애플리케이션에서 엔드포인트로 접속

## 실습 시작

### 사전 준비

#### Cloud9 IDE 환경 구성

아래 workshop 지시 대로 미리 cloud9을 생성해 놓는다

<https://catalog.us-east-1.prod.workshops.aws/workshops/46236689-b414-4db8-b5fc-8d2954f2d94a/ko-KR/install/10-cloud9>



인증/자격증명 및 환경 구성과 Cloud9 인스턴스에 보안그룹 수정까지 완료한다

## AWS Aurora and RDS 실습

### 1단계: RDS (MySQL) 인스턴스 생성

#### 1.1 AWS 콘솔에서 RDS 서비스 이동

- AWS 콘솔 → **RDS** 검색 후 클릭 (**Aurora and RDS**)
- 왼쪽 메뉴에서 **Databases** → **Create database**
- 


#### 1.2 기본 설정 : 아래 없는 내용은 기본 설정 값으로 두고 진행한다

- 데이터베이스 생성 방식 선택 : 표준 생성
- 엔진 옵션 : MySQL
- 템플릿 : 프리 티어
- 가용성 및 내구성 : 단일 AZ DB 인스턴스 배포(인스턴스 1개)
- DB 인스턴스 식별자: mydb
- 마스터 사용자 이름: admin
- 마스터 암호: password1234
- DB instance class: db.t3.micro (프리 티어 대상)
- Storage: 기본값 (20GiB)

#### 1.3 네트워크 설정

- VPC: Default VPC
- Public access: 예 (Yes)
- VPC 보안 그룹 (방화벽): 기존 항목 선택
- 기존 VPC 보안 그룹 :

default를 지우고 새로 생성된 cloud9의 보안그룹을 선택해준다



The screenshot shows a dropdown menu titled "기존 VPC 보안 그룹" (Existing VPC Security Group). The dropdown is open, showing a list of security groups. The selected option is "aws-cloud9-myscoud9-b7e3a617e092482ea1f3611a892e8d75-InstanceSecurityGroup-nY35gwFMk7jC". There is a blue 'X' icon next to the selected option. Above the dropdown, there is a text input field with the placeholder "하나 이상의 옵션 선택" (Select one or more options).

✦ Cloud9과 같은 보안그룹을 지정하면, IP가 바뀌어도 계속 접속이 가능

[데이터 베이스 생성] 버튼을 클릭한다(몇 분 소요)

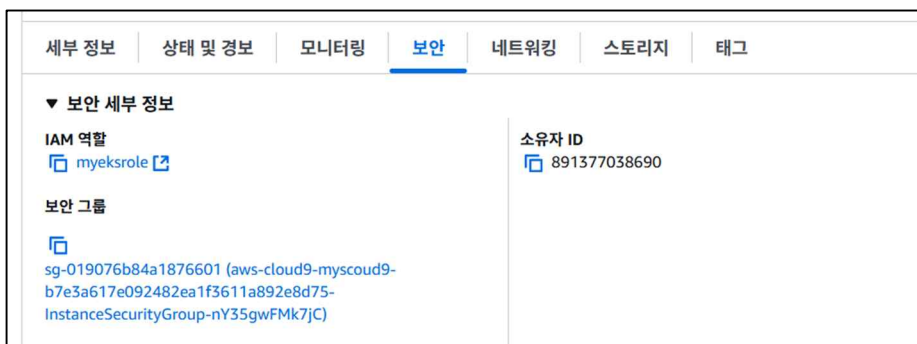
---

## 2단계: IAM 권한 설정

RDS에 직접 접속할 때는 IAM 권한이 필요 없지만, **Cloud9 인스턴스가 접근 가능한 보안 그룹에 속해 있어야** 합니다.

### 2.1 Cloud9 EC2의 보안 그룹 확인

1. EC2 에서 인스턴스 ID 클릭
2. 보안 탭 클릭
3. 보안그룹에 있는 내용 복사해둔다



예시)

aws-cloud9-myscoud9-b7e3a617e092482ea1f3611a892e8d75-InstanceSecurityGroup-nY35gwFMk7jC

### 2.2 RDS 인스턴스의 보안 그룹에서 인바운드 규칙 추가

- RDS 인스턴스 **VPC 보안 그룹** → Inbound rules 편집

- Type: MySQL/Aurora
- Protocol: TCP
- Port Range: 3306
- Source: 위에서 확인한 Cloud9 보안 그룹 선택
- [규칙 저장] 버튼 클릭

**인바운드 규칙 편집**

인바운드 규칙은 인스턴스에 도달하도록 허용된 수신 트래픽을 제어합니다.

보안 그룹 규칙 ID	유형	프로토콜	포트 범위	소스	설명 - 선택 사항	액션
sg-0485ed859ab97fd28	HTTP	TCP	80	사용자 ...	0.0.0.0/0	삭제
sg-0b3766871dd04c7f5	사용자 지정 TCP	TCP	2000	사용자 ...	0.0.0.0/0	삭제
-	MySQL/Aurora	TCP	3306	사용자 ...	sg-019076b84a1876601	삭제

[규칙 추가](#)

### 3단계: Python에서 RDS(MySQL) 접속

#### 3.1 필요한 패키지 설치

#### Cloud9 터미널에서 아래 명령 실행

pip install pymysql

pip3 install flask

pip install boto3

#### 3.2 RDS 접속 Endpoint 정보 확인

- RDS 대시보드 → 생성한 DB → **Endpoint** 복사

예시: mydb.czeysqe6yl7y.us-east-1.rds.amazonaws.com

- 포트: 3306
- 사용자명: admin
- 비밀번호: 생성할 때 입력한 비밀번호(password1234)
- 

---

#### 4단계: Python 코드로 RDS 접근 및 데이터 처리

##### cloud9에서 아래 명령 실행

```
touch rds_app.py
```

에디터에서 rds\_app.py 소스 아래와 같이 수정

"host" 에는 앞에서 복사해 놓은 RDS 접속 Endpoint 주소를 넣는다

```
from flask import Flask, request, jsonify
import boto3
import pymysql

app = Flask(__name__)

# DB 접속 설정
db_settings = {
    "host": "mydb.czeysqe6yl7y.us-east-1.rds.amazonaws.com",
    "port": 3306,
    "user": "admin",
    "password": "password1234",
    "database": "testdb",
    "charset": "utf8mb4"
}

# 초기화: 데이터베이스 및 테이블 생성
def init_db():
```

```

try:
    # DB 없을 경우 생성 (초기 접속은 mysql DB 사용)
    conn = pymysql.connect(
        host=db_settings['host'],
        port=db_settings['port'],
        user=db_settings['user'],
        password=db_settings['password'],
        database='mysql',
        charset='utf8mb4',
        autocommit=True
    )
    with conn.cursor() as cursor:
        cursor.execute("CREATE DATABASE IF NOT EXISTS testdb;")
    conn.close()

    # 테이블 생성
    conn = pymysql.connect(**db_settings)
    with conn.cursor() as cursor:
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS my_table (
                id INT PRIMARY KEY,
                name VARCHAR(100),
                age INT
            );
        """)
    conn.close()
    print("✅ RDS 초기화 완료")
except Exception as e:
    print("❌ RDS 초기화 실패:", e)

# RDS 연결 (매 요청마다 열고 닫는 방식이 안정적)
def get_db_connection():
    return pymysql.connect(**db_settings)

# DynamoDB 리소스

```



```

dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
table = dynamodb.Table('MyDynamoTable')

@app.route('/')
def home():
    return "<h2>Flask 서버가 실행 중입니다. 😊<br>사용 가능한 엔드포인트: /store/dynamodb, /fetch/dynamodb, /store/rds, /fetch/rds</h2>"

# RDS에 저장
@app.route('/store/rds', methods=['POST'])
def store_rds():
    data = request.json
    conn = get_db_connection()
    try:
        with conn.cursor() as cursor:
            sql = "INSERT INTO my_table (id, name, age) VALUES (%s, %s, %s)"
            cursor.execute(sql, (data['id'], data['name'], data['age']))
        conn.commit()
        return jsonify({"message": "Data stored in RDS"}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500
    finally:
        conn.close()

# RDS에서 조회
@app.route('/fetch/rds', methods=['GET'])
def fetch_rds():
    id = request.args.get('id')
    conn = get_db_connection()
    try:
        with conn.cursor() as cursor:
            cursor.execute("SELECT id, name, age FROM my_table WHERE id = %s",
(id,))

            result = cursor.fetchone()
            if result:

```

```

        return jsonify({"id": result[0], "name": result[1], "age": result[2]}), 200
    else:
        return jsonify({"error": "No data found"}), 404
except Exception as e:
    return jsonify({"error": str(e)}), 500
finally:
    conn.close()

# RDS에서 데이터 수정
@app.route('/update/rds', methods=['PUT'])
def update_rds():
    data = request.json
    conn = get_db_connection()
    try:
        with conn.cursor() as cursor:
            sql = "UPDATE my_table SET name = %s, age = %s WHERE id = %s"
            cursor.execute(sql, (data['name'], data['age'], data['id']))
        conn.commit()
        if cursor.rowcount == 0:
            return jsonify({"error": "No record found to update"}), 404
        return jsonify({"message": "Data updated in RDS"}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500
    finally:
        conn.close()

# RDS에서 데이터 삭제
@app.route('/delete/rds', methods=['DELETE'])
def delete_rds():
    id = request.args.get('id')
    conn = get_db_connection()
    try:
        with conn.cursor() as cursor:
            sql = "DELETE FROM my_table WHERE id = %s"
            cursor.execute(sql, (id,))

```

```

        conn.commit()
        if cursor.rowcount == 0:
            return jsonify({"error": "No record found to delete"}), 404
        return jsonify({"message": "Data deleted from RDS"}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500
    finally:
        conn.close()

# DynamoDB에 저장
@app.route('/store/dynamodb', methods=['POST'])
def store_dynamodb():
    data = request.json
    try:
        table.put_item(Item=data)
        return jsonify({"message": "Data stored in DynamoDB"}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500

# DynamoDB에서 조회
@app.route('/fetch/dynamodb', methods=['GET'])
def fetch_dynamodb():
    id = request.args.get('id')
    try:
        response = table.get_item(Key={'ID': id})
        return jsonify(response.get('Item', {})), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500

# DynamoDB에서 데이터 수정
@app.route('/update/dynamodb', methods=['PUT'])
def update_dynamodb():
    data = request.json

```

```

try:
    table.update_item(
        Key={'ID': data['ID']},
        UpdateExpression="SET #n = :name, age = :age",
        ExpressionAttributeNames={'#n': 'name'},
        ExpressionAttributeValues={
            ':name': data['name'],
            ':age': data['age']
        }
    )
    return jsonify({"message": "Data updated in DynamoDB"}), 200
except Exception as e:
    return jsonify({"error": str(e)}), 500

# DynamoDB에서 데이터 삭제
@app.route('/delete/dynamodb', methods=['DELETE'])
def delete_dynamodb():
    id = request.args.get('id')
    try:
        response = table.delete_item(
            Key={'ID': id},
            ReturnValues='ALL_OLD'
        )
        if 'Attributes' in response:
            return jsonify({"message": "Data deleted from DynamoDB"}), 200
        else:
            return jsonify({"error": "No record found to delete"}), 404
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    init_db()
    app.run(host='0.0.0.0', port=5000)

```

소스 저장 후 cloud9에서 아래 명령 수행

```
python3 rds_app.py
```

아래와 같이 서버가 실행된다

```
ec2-user:~/environment $ python3 rds_app.py
✓ RDS 초기화 완료
* Serving Flask app 'rds_app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.31.24.106:5000
Press CTRL+C to quit
```

cloud9에서 새로운 터미널을 열고 아래 명령을 순서 대로 실행한다

[RDS 데이터 저장]

```
curl -X POST http://127.0.0.1:5000/store/rds ₩
-H "Content-Type: application/json" ₩
-d '{"id": 1, "name": "Alice", "age": 30}'
```

[RDS 데이터 읽기]

```
curl http://127.0.0.1:5000/fetch/rds?id=1
```

[RDS 데이터 변경]

```
curl -X PUT http://127.0.0.1:5000/update/rds ₩
-H "Content-Type: application/json" ₩
-d '{"id": 1, "name": "Tom", "age": 40}'
```

[RDS 데이터 읽기]

```
curl http://127.0.0.1:5000/fetch/rds?id=1
```

[RDS 데이터 삭제]

```
curl -X DELETE http://127.0.0.1:5000/delete/rds?id=1
```

[RDS 데이터 읽기]

```
curl http://127.0.0.1:5000/fetch/rds?id=1
```

<실행 출력 결과>

```
ec2-user:~/environment $ curl -X POST http://127.0.0.1:5000/store/rds \
> -H "Content-Type: application/json" \
> -d '{"id": 1, "name": "Alice", "age": 30}'
{"message": "Data stored in RDS"}
ec2-user:~/environment $
ec2-user:~/environment $ curl http://127.0.0.1:5000/fetch/rds?id=1
{"age": 30, "id": 1, "name": "Alice"}
ec2-user:~/environment $
ec2-user:~/environment $ curl -X PUT http://127.0.0.1:5000/update/rds \
> -H "Content-Type: application/json" \
> -d '{"id": 1, "name": "Tom", "age": 40}'
{"message": "Data updated in RDS"}
ec2-user:~/environment $
ec2-user:~/environment $ curl http://127.0.0.1:5000/fetch/rds?id=1
{"age": 40, "id": 1, "name": "Tom"}
ec2-user:~/environment $
ec2-user:~/environment $ curl -X DELETE http://127.0.0.1:5000/delete/rds?id=1
{"message": "Data deleted from RDS"}
ec2-user:~/environment $
ec2-user:~/environment $ curl http://127.0.0.1:5000/fetch/rds?id=1
{"error": "No data found"}
ec2-user:~/environment $
```

<서버측 메시지 출력>

```
ec2-user:~/environment $ python3 rds_app.py
[✓] RDS 초기화 완료
* Serving Flask app 'rds_app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.31.24.106:5000
Press CTRL+C to quit
127.0.0.1 - - [31/Mar/2025 16:34:44] "POST /store/rds HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 16:34:56] "GET /fetch/rds?id=1 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 16:35:05] "PUT /update/rds HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 16:35:10] "GET /fetch/rds?id=1 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 16:35:40] "DELETE /delete/rds?id=1 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 16:35:45] "GET /fetch/rds?id=1 HTTP/1.1" 404 -
```

Cloud9에서 직접 RDS에 연결하고 데이터를 읽고 쓸 수 있게 되었다.

## 5단계: Python 코드로 DynamoDB 접근 및 데이터 처리

### 2) DynamoDB 테이블 생성

DynamoDB 테이블을 생성하려면 다음 명령어를 실행합니다.

```
aws dynamodb create-table --table-name MyDynamoTable --attribute-definitions  
AttributeName=ID,AttributeType=S --key-schema AttributeName=ID,KeyType=HASH  
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

화면에 긴 텍스트가 출력되면 'q'를 입력하고 Enter를 쳐서 빠져나온다

AWS 콘솔에서 DynamoDB에 DB가 생성된 결과를 확인한다



python3 rds\_app.py 실행되어 있는 상태에서

[DynamoDB 데이터 저장]

```
curl -X POST http://127.0.0.1:5000/store/dynamodb ₩  
-H "Content-Type: application/json" ₩  
-d '{"ID": "123", "name": "Alice", "age": 30}'
```

[DynamoDB 데이터 읽기]

```
curl http://127.0.0.1:5000/fetch/dynamodb?id=123
```

[DynamoDB 데이터 변경]

```
curl -X PUT http://127.0.0.1:5000/update/dynamodb ₩
```

```
-H "Content-Type: application/json" ₩
```

```
-d '{"ID": "123", "name": "John", "age": 40}'
```

[DynamoDB 데이터 읽기]

```
curl http://127.0.0.1:5000/fetch/dynamodb?id=123
```

[DynamoDB 데이터 삭제]

```
curl -X DELETE http://127.0.0.1:5000/delete/dynamodb?id=123
```

[DynamoDB 데이터 읽기]

```
curl http://127.0.0.1:5000/fetch/dynamodb?id=123
```

<출력 결과>

```
ec2-user:~/environment $ curl -X POST http://127.0.0.1:5000/store/dynamodb \
> -H "Content-Type: application/json" \
> -d '{"ID": "123", "name": "Alice", "age": 30}'
{"message": "Data stored in DynamoDB"}
ec2-user:~/environment $ curl http://127.0.0.1:5000/fetch/dynamodb?id=123
{"ID": "123", "age": "30", "name": "Alice"}
ec2-user:~/environment $ curl -X PUT http://127.0.0.1:5000/update/dynamodb \
> -H "Content-Type: application/json" \
> -d '{"ID": "123", "name": "John", "age": 40}'
{"message": "Data updated in DynamoDB"}
ec2-user:~/environment $ curl http://127.0.0.1:5000/fetch/dynamodb?id=123
{"ID": "123", "age": "40", "name": "John"}
ec2-user:~/environment $ curl -X DELETE http://127.0.0.1:5000/delete/dynamodb?id=123
{"message": "Data deleted from DynamoDB"}
ec2-user:~/environment $ curl http://127.0.0.1:5000/fetch/dynamodb?id=123
{}

```



## <서버측 메시지>

```
ec2-user:~/environment $ python3 rds_app.py
[✓] RDS 초기화 완료
* Serving Flask app 'rds_app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.31.24.106:5000
Press CTRL+C to quit
127.0.0.1 - - [31/Mar/2025 17:13:32] "POST /store/dynamodb HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 17:13:40] "GET /fetch/dynamodb?id=123 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 17:13:49] "PUT /update/dynamodb HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 17:14:00] "GET /fetch/dynamodb?id=123 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 17:14:11] "DELETE /delete/dynamodb?id=123 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 17:14:20] "GET /fetch/dynamodb?id=123 HTTP/1.1" 200 -
```

## AWS DynamoDB

### 1. 개요

Amazon DynamoDB는 완전관리형 NoSQL 데이터베이스 서비스이다. 밀리초 수준의 일관된 응답 속도와 자동 확장 기능을 제공하며, 키-값(key-value) 및 문서(document) 데이터 모델을 지원한다. 서버 관리 없이고가용성과 내결함성, 보안을 갖춘 데이터 저장소를 제공한다.

---

### 2. 주요 특징

특징	설명
완전관리형 서비스	인프라 운영 없이 데이터베이스 자동 관리
NoSQL	관계형 스키마 없이 유연한 구조의 데이터 저장
빠른 성능	SSD 기반 저장, 메모리 캐싱(DAX), 분산 구조 활용
유연한 확장성	자동 확장 또는 수동 설정 가능
높은 가용성	리전 간 복제, 내결함성, 지속적 백업
보안	IAM 통합, KMS 암호화, VPC 엔드포인트 지원

---

### 3. 데이터 모델

#### 3.1 테이블(Table)

DynamoDB의 기본 단위로, 데이터를 저장하는 논리적인 그룹이다.

#### 3.2 항목(Item)

테이블 내의 하나의 레코드로, JSON 형식의 객체로 표현된다.

#### 3.3 속성(Attribute)

항목의 키-값 쌍이다. 속성 타입은 문자열, 숫자, 리스트, 맵, 부울 등 다양한 형식을 지원한다.

#### 3.4 파티션 키 / 정렬 키

- **파티션 키 (Partition Key):** 항목의 기본 해시 키 역할
  - **정렬 키 (Sort Key):** 복합 기본 키 구성 시 사용, 항목 간 정렬 가능
- 

### 4. 테이블 생성 예시

```
aws dynamodb create-table ₩  
  
  --table-name Users ₩  
  
  --attribute-definitions ₩  
  
    AttributeName=UserId,AttributeType=S ₩  
  
  --key-schema ₩  
  
    AttributeName=UserId,KeyType=HASH ₩  
  
  --provisioned-throughput ₩  
  
    ReadCapacityUnits=5,WriteCapacityUnits=5
```

---

### 5. 주요 API 예시 (AWS CLI)

#### 5.1 항목 추가

```
aws dynamodb put-item ₩
```

```
--table-name Users ₩
```

```
--item '{"UserId": {"S": "user123"}, "Name": {"S": "Alice"}}'
```

## 5.2 항목 조회

```
aws dynamodb get-item ₩
```

```
--table-name Users ₩
```

```
--key '{"UserId": {"S": "user123"}}'
```

## 5.3 항목 삭제

```
aws dynamodb delete-item ₩
```

```
--table-name Users ₩
```

```
--key '{"UserId": {"S": "user123"}}'
```

## 5.4 전체 스캔

```
aws dynamodb scan --table-name Users
```

---

## 6. 액세스 패턴 최적화

- DynamoDB는 쿼리 성능을 보장하기 위해 사용자의 **접근 패턴을 기준으로 테이블 설계**해야 한다.
- 관계형 데이터베이스와 달리 조인 연산이 없으므로, 중복을 허용한 비정규화 구조가 권장된다.

---

## 7. 인덱스

종류	설명
LSI (Local Secondary Index)	기본 파티션 키는 같고, 다른 정렬 키로 조회 가능
GSI (Global Secondary Index)	파티션 키와 정렬 키 모두 다른 조합으로 조회 가능

## GSI 생성 예시 (테이블 생성 시)

```
aws dynamodb create-table ₩
--table-name Orders ₩
--attribute-definitions ₩
  AttributeName=OrderId,AttributeType=S ₩
  AttributeName=CustomerId,AttributeType=S ₩
--key-schema ₩
  AttributeName=OrderId,KeyType=HASH ₩
--global-secondary-indexes ₩
'[{
  "IndexName": "CustomerIndex",
  "KeySchema":[
    {"AttributeName":"CustomerId","KeyType":"HASH"}
  ],
  "Projection":{"ProjectionType":"ALL"},
  "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":5}
}]'
```

---

## 8. 정적 vs 온디맨드 용량 모드

모드	설명
프로비저닝(Provisioned)	읽기/쓰기 용량을 수동 설정, 예측 가능한 워크로드에 적합
온디맨드(On-Demand)	요청 수에 따라 자동 확장, 예측 불가능한 트래픽에 적합

---

## 9. 보안

- IAM 정책을 통해 사용자 접근 제어
- KMS를 이용한 데이터 암호화
- VPC 엔드포인트를 통한 프라이빗 액세스

## IAM 정책 예시

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Users"
    }
  ]
}
```

---

## 10. DynamoDB Streams

- 테이블의 변경 사항(삽입, 갱신, 삭제)을 실시간 스트림으로 제공
  - AWS Lambda 등과 연동하여 실시간 처리 가능
- 

## 11. CloudFormation 예시

Resources:

UsersTable:

Type: AWS::DynamoDB::Table

Properties:

TableName: Users

AttributeDefinitions:

- AttributeName: UserId

AttributeType: S

KeySchema:

- AttributeName: UserId

KeyType: HASH

ProvisionedThroughput:

ReadCapacityUnits: 5

WriteCapacityUnits: 5

---

## 12. 요약

- DynamoDB는 서버 관리가 필요 없는 NoSQL DB 서비스이다.
- 테이블, 항목, 속성, 키 기반의 단순한 데이터 모델을 가진다.
- 온디맨드 또는 프로비저닝 모드로 용량 설정 가능하다.
- IAM, VPC, KMS 등과 통합되어 보안이 강화된다.
- 실시간 스트림과 Lambda를 연동하면 이벤트 기반 처리도 가능하다.

---

## 13. 참고 명령어 정리

# 테이블 생성

```
aws dynamodb create-table --table-name Users --attribute-definitions ...
```

# 항목 추가

```
aws dynamodb put-item --table-name Users --item '{...}'
```

# 항목 조회

```
aws dynamodb get-item --table-name Users --key '{...}'
```

# 항목 삭제

```
aws dynamodb delete-item --table-name Users --key '{...}'
```

# 전체 스캔

```
aws dynamodb scan --table-name Users
```

<The End>