**Day 6 – DynamoDB Integration for Leave Requests**

**What I Accomplished Today**

Today I successfully integrated Amazon DynamoDB into my HRMS project to handle leave request storage and management. This was a significant step forward as I moved from traditional relational database to NoSQL cloud storage for this specific feature.

**DynamoDB Setup and Configuration**

**Local DynamoDB Configuration**

I set up my DynamoDB client to work with a local DynamoDB instance for development:

public DynamoDBService() {

dynamoDb = DynamoDbClient.builder()

.region(Region.US\_EAST\_1)

.credentialsProvider(

StaticCredentialsProvider.create(

AwsBasicCredentials.create("dummyKey", "dummySecret")

)

)

.endpointOverride(URI.create("http://localhost:8000")) // Local DynamoDB

.build();

ensureTableExists();

}

**Key Configuration Decisions:**

* **Local Development**: Used http://localhost:8000 for local DynamoDB testing
* **Dummy Credentials**: Used placeholder credentials for local development
* **Auto-Setup**: Automatically creates table if it doesn't exist
* **Table Name**: LeaveRequests as the main table

**Table Schema Design**

**LeaveRequests Table Structure**

private void ensureTableExists() {

try {

// Check if table exists

dynamoDb.describeTable(DescribeTableRequest.builder()

.tableName(tableName)

.build());

System.out.println("✅ DynamoDB Table already exists: " + tableName);

} catch (ResourceNotFoundException e) {

// Create table if not found

CreateTableRequest request = CreateTableRequest.builder()

.tableName(tableName)

.keySchema(KeySchemaElement.builder()

.attributeName("requestId")

.keyType(KeyType.HASH) // Partition key

.build()

)

.attributeDefinitions(AttributeDefinition.builder()

.attributeName("requestId")

.attributeType(ScalarAttributeType.S)

.build()

)

.provisionedThroughput(ProvisionedThroughput.builder()

.readCapacityUnits(5L)

.writeCapacityUnits(5L)

.build()

)

.build();

dynamoDb.createTable(request);

System.out.println("✅ Table created successfully!");

}

}

**Table Design Decisions:**

* **Primary Key**: requestId (String) as partition key
* **Capacity**: 5 RCU/WCU for development (cost-effective)
* **Attributes**: No predefined attributes (NoSQL flexibility)
* **Auto-Creation**: Smart detection and creation if table missing

**Core CRUD Operations**

**Adding Leave Requests**

public void addLeaveRequest(LeaveRequest lr) {

Map<String, AttributeValue> item = new HashMap<>();

item.put("requestId", AttributeValue.builder().s(lr.getRequestId()).build());

item.put("employeeId", AttributeValue.builder().n(String.valueOf(lr.getEmployeeId())).build());

item.put("startDate", AttributeValue.builder().s(lr.getStartDate().toString()).build());

item.put("endDate", AttributeValue.builder().s(lr.getEndDate().toString()).build());

item.put("reason", AttributeValue.builder().s(lr.getReason()).build());

item.put("status", AttributeValue.builder().s(lr.getStatus()).build());

if (lr.getEmployeeName() != null) {

item.put("employeeName", AttributeValue.builder().s(lr.getEmployeeName()).build());

}

dynamoDb.putItem(PutItemRequest.builder()

.tableName(tableName)

.item(item)

.build());

System.out.println("✅ Leave request added with Request ID " + lr.getRequestId());

}

**What I learned**:

* DynamoDB stores data as key-value AttributeValue objects
* Need to specify data types: .s() for strings, .n() for numbers
* Optional fields (like employeeName) need null checks
* LocalDate converts to string for storage

**Query Operations I Implemented**

**Get All Leave Requests**

public List<LeaveRequest> getAllLeaveRequests() {

List<LeaveRequest> list = new ArrayList<>();

ScanResponse response = dynamoDb.scan(ScanRequest.builder()

.tableName(tableName)

.build());

for (Map<String, AttributeValue> item : response.items()) {

list.add(mapToLeaveRequest(item));

}

return list;

}

**Get Leave Requests by Employee**

public List<LeaveRequest> getLeaveRequestsByEmployee(int employeeId) {

List<LeaveRequest> list = new ArrayList<>();

ScanResponse response = dynamoDb.scan(ScanRequest.builder()

.tableName(tableName)

.build());

for (Map<String, AttributeValue> item : response.items()) {

LeaveRequest lr = mapToLeaveRequest(item);

if (lr.getEmployeeId() == employeeId) {

list.add(lr);

}

}

return list;

}

**Get Pending Leave Requests (Admin Feature)**

public List<LeaveRequest> getAllPendingLeaveRequests() {

List<LeaveRequest> pending = new ArrayList<>();

ScanResponse response = dynamoDb.scan(ScanRequest.builder()

.tableName(tableName)

.build());

for (Map<String, AttributeValue> item : response.items()) {

LeaveRequest lr = mapToLeaveRequest(item);

if ("PENDING".equalsIgnoreCase(lr.getStatus())) {

pending.add(lr);

}

}

return pending;

}

**Current Approach**: Using scan operations for simplicity in development. In production, I would optimize with proper indexes and query operations.

**Update and Delete Operations**

**Update Leave Status**

public void updateLeaveStatus(int employeeId, LocalDate startDate, String status) {

String requestId = getRequestIdForLeave(employeeId, startDate);

if (requestId == null) {

System.out.println("❌ Could not find leave request for Employee ID " + employeeId + " on " + startDate);

return;

}

dynamoDb.updateItem(UpdateItemRequest.builder()

.tableName(tableName)

.key(Map.of("requestId", AttributeValue.builder().s(requestId).build()))

.updateExpression("SET #s = :status")

.expressionAttributeNames(Map.of("#s", "status"))

.expressionAttributeValues(Map.of(":status", AttributeValue.builder().s(status).build()))

.build());

System.out.println("✅ Leave status updated to " + status + " for Employee ID " + employeeId + " (Start: " + startDate + ")");

}

**Delete Leave Request**

public void deleteLeaveRequest(String requestId) {

dynamoDb.deleteItem(DeleteItemRequest.builder()

.tableName(tableName)

.key(Map.of("requestId", AttributeValue.builder().s(requestId).build()))

.build());

System.out.println("✅ Leave request deleted with Request ID " + requestId);

}

**Key Learning**: DynamoDB updates require the primary key (requestId) and use expression-based syntax for modifications.

**Data Mapping Helper Method**

**Converting DynamoDB Items to Java Objects**

private LeaveRequest mapToLeaveRequest(Map<String, AttributeValue> item) {

LeaveRequest lr = new LeaveRequest(

item.get("requestId").s(),

Integer.parseInt(item.get("employeeId").n()),

LocalDate.parse(item.get("startDate").s()),

LocalDate.parse(item.get("endDate").s()),

item.get("reason").s(),

item.get("status").s()

);

if (item.containsKey("employeeName")) {

lr.setEmployeeName(item.get("employeeName").s());

}

return lr;

}

**What I handled**:

* String to LocalDate parsing
* Number string to integer conversion
* Optional field checking to avoid null pointer exceptions
* Clean object creation with proper data types

**Helper Methods for Complex Operations**

**Finding Request ID by Employee and Date**

private String getRequestIdForLeave(int employeeId, LocalDate startDate) {

ScanResponse response = dynamoDb.scan(ScanRequest.builder()

.tableName(tableName)

.build());

for (Map<String, AttributeValue> item : response.items()) {

int empId = Integer.parseInt(item.get("employeeId").n());

LocalDate sDate = LocalDate.parse(item.get("startDate").s());

if (empId == employeeId && sDate.equals(startDate)) {

return item.get("requestId").s();

}

}

return null;

}

This helper method allows me to find the unique requestId when I only have employee information, which is useful for update operations.

**Features I Successfully Implemented**

**What Works:**

1. **Automatic Table Creation** - Checks and creates table if needed
2. **CRUD Operations** - Create, Read, Update, Delete leave requests
3. **Employee-Specific Queries** - Get leaves for specific employees
4. **Status-Based Filtering** - Find pending/approved/rejected leaves
5. **Admin Functions** - View all pending requests for approval
6. **Data Type Handling** - Proper conversion between Java and DynamoDB types
7. **Error Handling** - Console feedback with success/error messages

**📊 Query Methods Available:**

* getAllLeaveRequests() - Complete list
* getLeaveRequestsByEmployee(int employeeId) - Employee-specific
* getAllPendingLeaveRequests() - For admin approval workflow
* getLeaveRequestsByStatus(String status) - Status-based filtering

**What I Learned**

**Technical Skills:**

1. **NoSQL Database Design** - Understanding partition keys and data modeling
2. **AWS SDK Usage** - Working with DynamoDB Java SDK v2
3. **Data Serialization** - Converting between Java objects and DynamoDB format
4. **Local Development** - Setting up local DynamoDB for testing

**Future Improvements**

**Performance Optimizations:**

* **Global Secondary Indexes (GSI)** - For efficient employeeId queries
* **Query vs Scan** - Replace scans with queries for better performance
* **Batch Operations** - Handle multiple requests efficiently

**Production Readiness:**

* **Real AWS Credentials** - Configure proper IAM roles
* **Error Handling** - More robust exception management
* **Logging** - Replace console output with proper logging
* **Connection Pooling** - Optimize client connections

**Feature Enhancements:**

* **Date Range Queries** - Find leaves within specific periods
* **Bulk Status Updates** - Handle multiple approvals at once
* **Audit Trail** - Track who approved/rejected leaves

**Integration with Existing HRMS**

This DynamoDB service integrates with my existing HRMS architecture:

* **Model Layer**: Uses existing LeaveRequest class
* **Service Layer**: Can be called from LeaveDynamoService
* **Flexibility**: Runs alongside traditional SQL database for other features

.