Create Lark data mart checker app using AWS Lambda, AWS SAM

In this article, you will learn to create a data mart checker for country data marts (SG, MY) in Lark chat room (Data Pipeline Monitoring chat group) using AWS Lambda, AWS Serverless Application Model (SAM) CLI, and Lark API.

> Read more about AWS Lambda, AWS Serverless Application Model (SAM) CLI with examples attached.

Instruction

To build AWS Lambda Function, steps needed to cover:

1. Get data marts tables info from PostgreSQL
2. Get Lark Service API, Install AWS SAM CLI interface.
3. Define AWS Lambda function with the parameters we prepared, store the code file to Docker Image with the new build
4. Deploy the app with the newly built image to our server

Repositories:

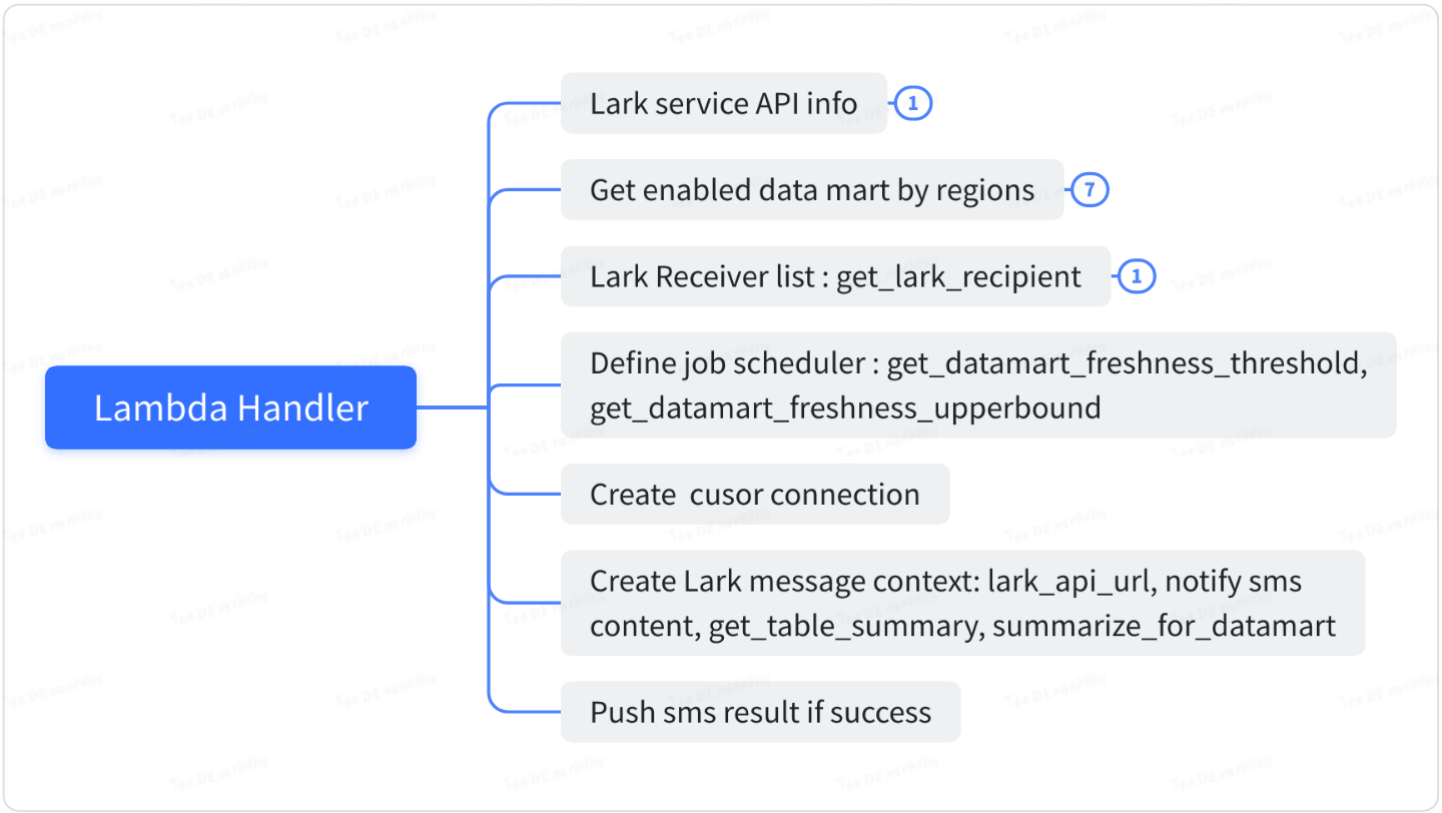
1. https://github.com/TrustyCars/carro-datamart-monitoring

This project contains source code and supporting files for a serverless application that you can deploy with the SAM CLI. It includes the following files and folders.

- src - Code for the application's Lambda function.
- events - Invocation events that you can use to invoke the function.
- tests - Unit tests for the application code.
- template.yaml - A template that defines the application's AWS resources.

The app builder process

The whole app builder pipeline on this project is as follows, which is to get information we need to push into AWS Lambda Handler function. The app code can be found at /src/app.py



The resources we will need:

- carro-datamart-monitoring SAM template in yaml file (template that defines the application's AWS resources)
- gitignore
- ARN set up and events file (Invocation events that you can use to invoke the function)
- Besides, you can setup AWS SAM CLI and clone tests folder to play around with.

Requirements.txt

These packages are not supported in AWS Lambda, Requirement.txt file will help the system know what to install packages you desire to run your code.

If you test in your local IDE, make sure you have requirement packages installed and updated.

```
psycopg2-binary==2.9.3
requests==2.28.1
```

Code explain

@ Class EnvHandler : A class to handle/parse env vars.

Lark Service API info

```python
def get_lark_service_api_info(self) -> dict:
        api_info = dict()
        attr_host = "LARK_SERVICE_HOST"
        attr_port = "LARK_SERVICE_PORT"
        attr_proto = "LARK_SERVICE_PROTO"
        attr_endpoint = "LARK_SERVICE_ENDPOINT"
        attr_api_key = "LARK_SERVICE_API_KEY"

        api_info["host"] = os.environ.get(attr_host)
        api_info["port"] = os.environ.get(attr_port)
        api_info["proto"] = os.environ.get(attr_proto)
        api_info["endpoint"] = os.environ.get(attr_endpoint)
        api_info["api_key"] = os.environ.get(attr_api_key)

        setattr(self, attr_host, api_info.get("host"))
        setattr(self, attr_port, api_info.get("port"))
        setattr(self, attr_proto, api_info.get("proto"))
        setattr(self, attr_endpoint, api_info.get("endpoint"))
        setattr(self, attr_api_key, api_info.get("api_key"))

        return api_info
```

Get Lark Receivers

```python
def get_lark_recipient(self) -> dict:
        notify_info = dict()
        available_recipient_keys = [
            "NOTIFY_TO_LARK_CHAT_ID",
            "NOTIFY_TO_LARK_USER_ID",
            "NOTIFY_TO_LARK_OPEN_ID",
            "NOTIFY_TO_LARK_EMAIL",
        ]

        for each in available_recipient_keys:
            tmp_recipient = os.environ.get(each)
            if tmp_recipient:
                tmp_key_name = each.replace("NOTIFY_TO_LARK_", "").lower()
                notify_info[tmp_key_name] = tmp_recipient
                break

        return notify_info
```

Get enabled data mart regions, data mart connections, table names info（for 5 countries: SG, MY, ID, TW, TH). A data mart would have this info as an example:

```
- DATAMART_SG_HOST
- DATAMART_SG_PORT
- DATAMART_SG_USER
- DATAMART_SG_PASSWORD
- DATAMART_SG_DB
- DATAMART_SG_SCHEMA
```

```python
def get_tables(self) -> list:
        tables = []
        try:
            tables = json.loads(os.environ.get("DATAMART_TABLES"))
        except Exception as exc:
            raise exc
        return tables

    def get_enabled_datamart_regions(self) -> list:
        regions = []
        try:
            regions = json.loads(os.environ.get("ENABLED_DATAMART_REGIONS"))
        except Exception as exc:
            raise exc
        return regions

    def get_datamart_connection_info(self, country_code: str) -> dict:
        conn_info = dict()
        upper_cased = country_code.upper()
        attr_host = f"DATAMART_{upper_cased}_HOST"
        attr_port = f"DATAMART_{upper_cased}_PORT"
        attr_user = f"DATAMART_{upper_cased}_USER"
        attr_password = f"DATAMART_{upper_cased}_PASSWORD"
        attr_dbname = f"DATAMART_{upper_cased}_DB"
        attr_schemaname = f"DATAMART_{upper_cased}_SCHEMA"

        conn_info["host"] = os.environ.get(attr_host)
        conn_info["port"] = int(os.environ.get(attr_port, "5432"))
        conn_info["user"] = os.environ.get(attr_user)
        conn_info["password"] = os.environ.get(attr_password)
        conn_info["dbname"] = os.environ.get(attr_dbname)
        schema_name = os.environ.get(attr_schemaname)
        conn_info["options"] = f"-c search_path={schema_name}"

```

```
        setattr(self, attr_host, conn_info.get("host"))
        setattr(self, attr_port, conn_info.get("port"))
        setattr(self, attr_user, conn_info.get("user"))
        setattr(self, attr_password, conn_info.get("password"))
        setattr(self, attr_dbname, conn_info.get("dbname"))
        setattr(self, attr_schemaname, schema_name)

        return conn_info
```

As our data marts servers get fresh everyday every 3 hours to run through all ETL pipelines from raw data source, we should get info of their refresh, put into a range of [threshold, upperbound] = [0,24]

```
def get_datamart_freshness_threshold(self) -> int:
        return int(os.environ.get("DATAMART_FRESHNESS_THRESHOLD") or 0)

    def get_datamart_freshness_upperbound(self) -> int:
        return int(os.environ.get("DATAMART_FRESHNESS_UPPERBOUND") or 24)
```

Create connection to server using **psycopg2**

```
import psycopg2
import requests
from psycopg2 import DatabaseError
from psycopg2.extras import RealDictCursor

def create_cursor(db_conn: dict) -> Tuple:
    cursor_options = dict(cursor_factory=RealDictCursor)
    conn = psycopg2.connect(**db_conn)
    cursor = conn.cursor(**cursor_options)
    return (conn, cursor)
```

@Class Lark Extension : Compile message output context

- Create lark API url, where we parse message result into
- Get Lark API and url
- Define notify message : message, title, chat_id, open_id, user_id, email, message_type, title_template (optional), lines (optional) to provide to Lark, in json format, or else raise error

```
def __init__(self, api_info: dict):
```

```python
        self.api_info = api_info
        self.api_url = self.create_lark_api_url()
        self.headers = {
            "Authorization": "Bearer {}".format(self.api_info.get("api_key"))
        }

    def create_lark_api_url(self) -> str:
        return "{proto}://{host}:{port}/{endpoint}".format_map(self.api_info)

    def notify(
        self,
        message: str,
        title: str,
        chat_id: str = None,
        open_id: str = None,
        user_id: str = None,
        email: str = None,
        message_type: str = "interactive",
        title_template: str = "orange",
        lines: int = 100,
    ) -> Any:
        if not (chat_id or open_id or user_id or email):
            raise ValueError(
                "Please provide either chat_id or open_id or"
                " user_id or email."
            )

        data = {
            "message_type": message_type,
            "message": message,
            "title": title,
            "title_template": title_template,
            "lines": lines,
        }
        if chat_id:
            data["chat_id"] = chat_id
        elif open_id:
            data["open_id"] = open_id
        elif user_id:
            data["user_id"] = user_id
        elif email:
            data["email"] = email

        resp = requests.post(self.api_url, json=data, headers=self.headers)
        if resp.ok:
            return resp.json()
```

Get table data summary

We want to check data freshness every day by 2 columns : [table_name]_created_at and [table_name]_updated_at.

We use functions that we already defined:

- Class EnvHandler
- get_datamart_connection_info
- create_cursor

The logic query applied here, for example:

```
SELECT 'activities' AS table_name,
        COUNT(*) AS total,
        MAX(activity_created_at) AS max_created_at,
        MAX(activity_updated_at) AS max_updated_at,
        now() - MAX(activity_created_at) as freshness_interval
        FROM activities
  # table_name = {schema_name}.{table_name}
```

```
def get_table_summary(
    table_name: str, region_name: str, pipe_conn: Any
) -> dict:
    row = dict()
    env_handler = EnvHandler()
    db_conn = env_handler.get_datamart_connection_info(region_name)
    conn, cursor = create_cursor(db_conn)
    info_query = (
        "SELECT column_name FROM information_schema.columns"
        " WHERE table_catalog = %s"
        " AND table_schema = %s"
        " AND table_name = %s"
        " AND (column_name LIKE '%%_created_at'"
        " OR column_name LIKE '%%_updated_at')"
    )
    try:
        db_name = getattr(env_handler, f"DATAMART_{region_name.upper()}_DB")
        schema_name = getattr(
            env_handler, f"DATAMART_{region_name.upper()}_SCHEMA"
        )
        cursor.execute(info_query, (db_name, schema_name, table_name))
        info_rows = cursor.fetchall()
```

```
        # SELECT 'activities' AS table_name,
        #     COUNT(*) AS total,
        #     MAX(activity_created_at) AS max_created_at,
        #     MAX(activity_updated_at) AS max_updated_at,
        #     now() - MAX(activity_created_at) as freshness_interval
        # FROM activities
        col_mapped = {}
        query = f"SELECT '{table_name}' AS table_name" ", COUNT(*) AS total"
        for each_info in info_rows:
            col_name = each_info.get("column_name")
            if col_name.endswith("created_at"):
                query += f", MAX({col_name}) AS max_created_at"
                query += f", FLOOR(EXTRACT(EPOCH FROM now() - MAX({col_name}))/3
            elif col_name.endswith("updated_at"):
                query += f", MAX({col_name}) AS max_updated_at"
        query += f" FROM {schema_name}.{table_name};"
        cursor.execute(query)
        row = cursor.fetchone()
    except DatabaseError as de:
        print(f"DatabaseError: {de}")
    except Exception as ge:
        print(f"GeneralError: {ge}")
        import traceback

        traceback.print_exc()
    finally:
        conn.close()

    pipe_conn.send(row)
    pipe_conn.close()
```

Get data mart conditions summary

Functions to be used:

- Class EnvHandler
- get_tables
- get_table_summary

```
def summarize_for_datamart(region_name: str) -> list:
    result = []
    env_handler = EnvHandler()
    tables = env_handler.get_tables()

    result = []
```

```
    processes = []
    parent_connections = []

    for table_name in tables:
        parent_conn, child_conn = Pipe()
        parent_connections.append(parent_conn)

        process = Process(
            target=get_table_summary,
            args=(
                table_name,
                region_name,
                child_conn,
            ),
        )
        processes.append(process)

    for each_process in processes:
        each_process.start()

    for each_process in processes:
        each_process.join()

    for parent_connection in parent_connections:
        result.append(parent_connection.recv())

    return
```

Environment set up

The Serverless Application Model Command Line Interface (SAM CLI) is an extension of the AWS CLI that adds functionality for building and testing Lambda applications. It uses Docker to run your functions in an Amazon Linux environment that matches Lambda. It can also emulate your application's build environment and API.

To use the SAM CLI, you need the following tools.

- SAM CLI - Install the SAM CLI
- Python 3 installed
- Docker - Install Docker community edition

To build and deploy your application for the first time, run the following in your shell:

```
sam build --use-container
sam deploy --guided
```

Moving forward, to build our application with the `sam build --use-container` command.

```
carro-db-monitoring$ sam build --use-container
```

The SAM CLI installs dependencies defined in `hello_world/requirements.txt`, creates a deployment package, and saves it in the `.aws-sam/build` folder.

Test a single function by invoking it directly with a test event. An event is a JSON document that represents the input that the function receives from the event source. Test events are included in the `events` folder in this project.

Run functions locally and invoke them with the `sam local invoke` command.

```
carro-db-monitoring$ sam local invoke CarroDatamartMonitoring --event events/ev
```

The SAM CLI can also emulate your application's API. Use the `sam local start-api` to run the API locally on port 3000.

```
carro-db-monitoring$ sam local start-api
carro-db-monitoring$ curl http://localhost:3000/
```

The SAM CLI reads the application template to determine the API's routes and the functions that they invoke. The `Events` property on each function's definition includes the route and method for each path.

```
Events:
  CarroDatamartMonitoring:
    Type: Api
    Properties:
      Path: /hello
      Method: get
```

Fetch, tail, and filter Lambda function logs

To simplify troubleshooting, SAM CLI has a command called `sam logs`. `sam logs` lets you fetch logs generated by your deployed Lambda function from the command line. In addition to

printing the logs on the terminal, this command has several nifty features to help you quickly find the bug.

`NOTE` : This command works for all AWS Lambda functions; not just the ones you deploy using SAM.

```
carro-db-monitoring$ sam logs -n CarroDatamartMonitoring --stack-name carro-db-m
```

You can find more information and examples about filtering Lambda function logs in the SAM CLI Documentation.

Tests

To understand more about AWS SAM, you can run tests folder. Tests are defined in the `tests` folder in this project. Use PIP to install the test dependencies and run tests.

```
carro-db-monitoring$ pip install -r tests/requirements.txt --user
# unit test
carro-db-monitoring$ python -m pytest tests/unit -v
# integration test, requiring deploying the stack first.
# Create the env variable AWS_SAM_STACK_NAME with the name of the stack we are t
carro-db-monitoring$ AWS_SAM_STACK_NAME=<stack-name> python -m pytest tests/inte
```

Cleanup

To delete the sample application that you created, use the AWS CLI. Assuming you used your project name for the stack name, you can run the following:

```
aws cloudformation delete-stack --stack-name carro-db-monitoring
```

Resources

See the AWS SAM developer guide for an introduction to SAM specification, the SAM CLI, and serverless application concepts.

Next, you can use AWS Serverless Application Repository to deploy ready to use Apps that go beyond hello world samples and learn how authors developed their applications: AWS Serverless Application Repository main page