Brainvoy- CSD (Code Specification Document)

Overview:

The **Brainvoy Flask App** is a web application that integrates with HubSpot using Airbyte, allowing users to interact with the HubSpot API. The project includes a web interface for users, a backend service, and scripts for managing data flows.

1- Technologies Used

- **Backend:** Python (Flask)
- **Frontend:** HTML, CSS, Bootstrap
- **Data Integration:** Airbyte for HubSpot

Below Project Structure layout is mentioned:

```
Brainvoy/
│
├── static/
│   └── style.css              # CSS file for custom styles
│
├── templates/
│   ├── home.html              # Homepage template
│   └── success.html           # Success page template
│
├── __pycache__/               # Compiled Python bytecode files
│
├── Airbyte_HubSpot.py         # Handles data integration with Airbyte and HubSpot
├── app.py                     # Main Flask application
└── requirements.txt           # Project dependencies
```

2- Dependencies and Installation

Dependencies:

- Flask
- Airbyte API Client
- Any other Python libraries specified in `requirements.txt`

3- Application Logic

Frontend (HTML Templates):

- **home.html**: This file contains the layout for the homepage. It includes the form which contains user data and Connect with HubSpot link to interact with new users.
- **success.html**: Displays a confirmation or success message after the user completes a task

4- Backend:

1- app.py script

Routes:

1. **`/` (Home Route)**

2. **`/process_selection` (Process Selections Route)**

Functions working:

When the Flask app is started (`app.py`), it first imports all the required modules and sets up OAuth 2.0 credentials for HubSpot. Below see required credentials are mentioned for **OAuth 2.0 Configuration.**

- **`CLIENT_ID` and `CLIENT_SECRET`**: OAuth credentials for HubSpot.
- **`INSTALL_URL`**: Authorization URL for users to grant access to the HubSpot API.

Home Page (`/`): `home()` function

1. This function is the main route of the Flask application and handles two scenarios of HubSpot with an authorization code:

    a. If the authorization `code` is absent, the function skips loading stored user data (from the YAML file) via function **`load_yaml_file()`** and renders the home page.

Processing Selections (`/process_selection`): If the user selects specific HubSpot accounts to fetch data for, the app refreshes the tokens and runs the `airbyte_app()` function for those accounts, which triggers data integration.

b. When the user returns from HubSpot with an authorization code. If the `code` parameter exists in the URL, the app continues to get HubSpot access and refresh tokens via `get_tokens()` function. Then `get_access_token_info()` function, which retrieves the details associated with the access token (such as the `hub_id` and the user's email).

2. Then the user_data dictionary contains the retrieved user information: `hub_id`, `access_token`, `refresh_token`, `user`, and the custom `naming_convention`.

    1. Concatenate the `user_name` and `hub_id_str` to create a unique `naming_convention`

3. After storing the user's tokens, the app calls the `airbyte_app()` function, which triggers a data synchronization process between HubSpot and Airbyte using the refreshed tokens.

2- Airbyte_HubSpot.py script:

The following script shows the integration of Airbyte with HubSpot and S3. It handles the creation of sources, destinations, and connections.

Airbyte_app():

1. This function first will go through the Access Token Management process via `get_access_token()` function to handle token refresh properly and return a valid token.

2. The `process_yaml_to_json` function reads a JSON file (`user_data.json,` it contains account users data with source, destination and connection info).

    Structure of `user_data.json:` For instance, take the user samiullah_47293483, where 'samiullah' represents the username of the authenticated user and '47293483' is the hub_id. This approach ensures unique identification across multiple accounts with the same username. In this case, the user 'samiullah' is registered under two different accounts, distinguished by the hub_id assigned after the underscore.

    1. **Sources:**
        a. `refresh_token`: `"797a6153-4f06-438b-beaf-e452b98bd1f1"`
        b. `name`: `"samiullah_47293483_HubSpot"`

2. **Destination:**
    a. `name: "samiullah_47293483_S3"`
    b. `s3_bucket_path: "samiullah_47293483/${YEAR}/${MONTH}/${DAY}"`
3. **Connections:**
    a. `name: "samiullah_47293483_HubSpot_S3"`

1. **Load YAML file** → 2. **Convert data to JSON** → 3. **Save JSON data to a file**

Inside the process_yaml_to_json function after loading yaml file data, there's `yaml_to_json` function that is designed to convert YAML data into a JSON format.

3. For further source, destination , connection creation we'll access `workspace_id` for a workspace within Airbyte.

    a. **Source and Destination Creation:** The `create_source` , `create_destination` function aims to create a new source & destination if it does not already exist with the specified name.

        - The function checks if the `refresh_token` matches before creating sources and destinations. Ensure that `refresh_token` matches with the tokens in `user_data.json` to create sources and destinations.
        - Constructs the URL for the API endpoint to create a new source & destination.
        - Sets up the headers for the HTTP request, including the authorization header with the access token.

    b. **Creating Connections:** The `create_connection` function is designed to create a new connection between a source and a destination if it doesn't already exist.

        - The function checks if the `refresh_token` matches before creating a connection . Ensure that `refresh_token` matches with the tokens in `user_data.json` to create connection.
        - Constructs the URL for the API endpoint to create a new connection.
        - Sets up the headers for the HTTP request, including the authorization header with the access token.
        - Constructs the payload for the API request which also includes specified streams to synchronized data with `"full_refresh_append"` sync mode, meaning each dataset will be completely refreshed and appended to the destination.

5- Code Demo:

app.py file code:

```
import requests, os, yaml, signal
import pandas as pd
from flask import (
    Flask,
    request,
    render_template,
    render_template_string,
    redirect,
    url_for,
)
from hubspot import HubSpot

# Import the airbyte_app function from airbyte_hub.py
from Airbyte_HubSpot import airbyte_app


yml_path = os.getcwd() + "\\tokens.yaml"


app = Flask(__name__)

# OAuth 2.0 credentials of hubspot
CLIENT_ID = "46a9699b-6de9-4e5a-aef4-6e38aaeffb22"
CLIENT_SECRET = "742140b0-f917-4a75-830b-e7c178a43c63"
INSTALL_URL = f"https://app.hubspot.com/oauth/authorize?client_id=46a9699b-6de9-4e5a-
aef4-
6e38aaeffb22&redirect_uri=http://localhost:5000&scope=crm.objects.line_items.read%20cont
ent%20crm.schemas.deals.read%20crm.objects.carts.read%20media_bridge.read%20market
ing-
email%20automation%20crm.pipelines.orders.read%20crm.objects.subscriptions.read%20tim
eline%20oauth%20crm.objects.owners.read%20forms%20transactional-
email%20crm.objects.users.read%20tickets%20crm.objects.users.write%20e-
commerce%20crm.objects.marketing_events.read%20crm.schemas.custom.read%20crm.obj
ects.custom.read%20crm.objects.feedback_submissions.read%20sales-email-
read%20crm.objects.goals.read%20crm.objects.companies.read%20crm.lists.read%20crm.ob
jects.deals.read%20crm.schemas.contacts.read%20crm.objects.contacts.read%20crm.schem
as.companies.read"

# Function to load data from a YAML file
def load_yaml_file(file_path):
    try:
        with open(file_path, "r") as file:
            return yaml.safe_load(file) or []
    except FileNotFoundError:
        return []
```

```
    except yaml.YAMLError as e:
        print(f"Error reading YAML file: {e}")
        return []


# Function to save data to a YAML file
def save_to_yaml_file(data, filename):
    """Helper function to save data to a YAML file."""
    if not data:
        return

    existing_data = load_yaml_file(filename)
    existing_hub_ids = {row["hub_id"] for row in existing_data}

    # Update existing data
    for row in data:
        for existing_row in existing_data:
            if existing_row["hub_id"] == row["hub_id"]:
                existing_row.update(row)
                break
        else:
            existing_data.append(row)

    with open(filename, "w") as file:
        yaml.safe_dump(existing_data, file)


def is_yaml_empty():
    data = load_yaml_file(yml_path)
    return len(data) == 0


# Function to get new access token using refresh token
def get_new_access_token(refresh_token):
    url = "https://api.hubapi.com/oauth/v1/token"

    data = {
        "grant_type": "refresh_token",
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET,
        "refresh_token": refresh_token,
    }

    response = requests.post(url, data=data)

    if response.status_code == 200:
        updated_data = response.json()
        return updated_data
    else:
        print(f"Error: {response.status_code} - {response.text}")
```

```
        return None


# Function to get access token info
def get_access_token_info(token):
    url = f"https://api.hubapi.com/oauth/v1/access-tokens/{token}"

    headers = {"accept": "application/json", "authorization": f"Bearer {token}"}

    response = requests.get(url, headers=headers)

    if response.status_code == 200:
        data = response.json()
        return data
    else:
        print(f"Failed to fetch access token info. Status Code: {response.status_code}")
        print("Response:", response.text)
        return None


# Function to get tokens using authorization code
def get_tokens(authorization_code):
    url = "https://api.hubspot.com/oauth/v1/token"
    data = {
        "grant_type": "authorization_code",
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET,
        "redirect_uri": "http://localhost:5000",
        "code": authorization_code,
    }

    response = requests.post(url, data=data)
    if response.status_code == 200:
        print(response.json())
        return response.json()
    else:
        print(f"Error: {response.status_code} - {response.text}")
        return None


# Route for the home page
@app.route("/")
def home():

    authorization_code = request.args.get("code")
    if authorization_code:
        tokens = get_tokens(authorization_code)
        if tokens:
            access_token = tokens["access_token"]
            refresh_token = tokens["refresh_token"]
```

```
            # hubspot_client = HubSpot(access_token=access_token)
            access_token_info = get_access_token_info(access_token)
            if access_token_info:
                user = access_token_info["user"]
                hub_id = access_token_info["hub_id"]
                # Convert hub_id to string
                hub_id_str = str(hub_id)
                user_name = user.split("@")[0]

                # Concatenate user and hub_id_str
                name = user_name + "_" + hub_id_str
                user_data = [
                    {
                        "hub_id": hub_id,
                        "access_token": access_token,
                        "refresh_token": refresh_token,
                        "user": user,
                        "naming_convention": name,
                    }
                ]
                save_to_yaml_file(user_data, yml_path)
                print("inside")
                airbyte_app(refresh_token)

                return render_template("success.html", user_data=user_data)

        return render_template("success.html", user_data=[])
    else:
        # Load stored data from YAML
        user_data = load_yaml_file(yml_path)
        return render_template(
            "home.html", authorization_url=INSTALL_URL, user_data=user_data
        )


# Route to process user selections
@app.route("/process_selection", methods=["POST"])
def process_selection():
    print("airbyte_app")
    selected_hub_ids = request.form.getlist("fetch_now")
    print("selected_hub_ids", selected_hub_ids)
    all_user_data = load_yaml_file(yml_path)
    # Track if any changes were made
    data_updated = False

    for row in all_user_data:
        if str(row["hub_id"]) in selected_hub_ids:
            refresh_token = row.get("refresh_token")
            print("refresh_token", refresh_token)
            if refresh_token:
```

```
            airbyte_app(refresh_token)

    if data_updated:
        # Re-save the updated tokens to the YAML file only if updates occurred
        save_to_yaml_file(all_user_data, yml_path)

    return render_template(
        "success.html",
        user_data=[
            row for row in all_user_data if str(row["hub_id"]) in selected_hub_ids
        ],
    )


if __name__ == "__main__":
    print("starting")
    app.run(debug=True)
```

Airbyte_HubSpot.py file code:

```
import requests
import json, os
import yaml


yaml_file_path = os.getcwd() + "\\tokens.yaml"
json_file_path = os.getcwd() + "\\user_data.json"

# Replace with your Airbyte web app URL
WEBAPP_URL = "http://ec2-13-59-215-90.us-east-2.compute.amazonaws.com:8000"  # Self-hosted URL or Airbyte's URL

# Your client_id and client_secret of airbyte
CLIENT_ID = "45925526-3951-4012-b274-f0f84a05d963"
CLIENT_SECRET = "KyN3u9YXidIIgPATNxnfi4e0u60hMEEl"

HUBSPOT_CLIENT_ID = "46a9699b-6de9-4e5a-aef4-6e38aaeffb22"
HUBSPOT_CLIENT_SECRET = "742140b0-f917-4a75-830b-e7c178a43c63"

S3_BUCKET_NAME = "hubspot-users-data-uncleaned"
S3_BUCKET_REGION = "us-east-2"
S3_BUCKET_ACCESS_KEY = "AKIA2UC26WFHR233QZWT"
S3_BUCKET_SECRET_KEY = "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"

# Source and Destination IDs
```

```
SOURCE_DEF_ID = "3e4730d5-92f2-463c-8bb9-e1f5a763ebe3"
DESTINATION_DEF_ID = "4816b78f-1489-44c1-9060-4b19d5fa9362"
WORKSPACE_ID = "d82eaa66-e92e-49fb-b0f4-dd0245126392"


def get_access_token():
    token_url = f"{WEBAPP_URL}/api/v1/applications/token"
    payload = {"client_id": CLIENT_ID, "client_secret": CLIENT_SECRET}
    headers = {"Content-Type": "application/json"}
    response = requests.post(token_url, json=payload, headers=headers)

    if response.status_code == 200:
        access_token = response.json().get("access_token")
        return access_token
    else:
        print(f"Failed to get access token: {response.status_code}")
        return None


def get_sources(access_token, workspace_id):
    source_url = f"{WEBAPP_URL}/api/v1/sources/list"
    headers = {
        "Authorization": f"Bearer {access_token}",
        "Content-Type": "application/json",
    }
    payload = {"workspaceId": workspace_id}
    response = requests.post(source_url, json=payload, headers=headers)

    if response.status_code == 200:
        sources = response.json().get("sources", [])
        return sources
    else:
        print(f"Failed to retrieve sources: {response.status_code}")
        return []


def get_destinations(access_token, workspace_id):
    destination_url = f"{WEBAPP_URL}/api/v1/destinations/list"
    headers = {
        "Authorization": f"Bearer {access_token}",
        "Content-Type": "application/json",
    }
    payload = {"workspaceId": workspace_id}
    response = requests.post(destination_url, json=payload, headers=headers)

    if response.status_code == 200:
        destinations = response.json().get("destinations", [])
        return destinations
    else:
        print(f"Failed to retrieve destinations: {response.status_code}")
```

```
        return []


def get_connections(access_token, workspace_id):
    connection_url = f"{WEBAPP_URL}/api/v1/connections/list"
    headers = {
        "Authorization": f"Bearer {access_token}",
        "Content-Type": "application/json",
    }
    payload = {"workspaceId": workspace_id}
    response = requests.post(connection_url, json=payload, headers=headers)

    if response.status_code == 200:
        connections = response.json().get("connections", [])
        # Save connections to file
        with open("connections.json", "w") as json_file:
            json.dump(connections, json_file, indent=4)
        return connections
    else:
        print(f"Failed to retrieve connections: {response.status_code}")
        return []


def create_source(access_token, refresh_token, name):
    existing_sources = get_sources(access_token, WORKSPACE_ID)
    if any(src["name"] == name for src in existing_sources):
        print(f"Source with name '{name}' already exists.")
        return

    source_url = f"{WEBAPP_URL}/api/public/v1/sources"
    headers = {
        "accept": "application/json",
        "content-type": "application/json",
        "authorization": f"Bearer {access_token}",
    }

    payload = {
        "sourceDefinitionId": SOURCE_DEF_ID,
        "name": name,
        "workspaceId": WORKSPACE_ID,
        "configuration": {
            "sourceType": "hubspot",
            "credentials": {
                "client_id": HUBSPOT_CLIENT_ID,
                "client_secret": HUBSPOT_CLIENT_SECRET,
                "refresh_token": refresh_token,
                "credentials_title": "OAuth Credentials",
            },
            "enable_experimental_streams": True,
        },
```

```
    }

    response = requests.post(source_url, json=payload, headers=headers)
    if response.status_code == 200:
        print(f"Source '{name}' created successfully.")
    else:
        print(
            f"Failed to create source '{name}': {response.status_code} - {response.text}"
        )


def create_destination(access_token, name, s3_bucket_path):
    existing_destinations = get_destinations(access_token, WORKSPACE_ID)
    if any(dest["name"] == name for dest in existing_destinations):
        print(f"Destination with name '{name}' already exists.")
        return

    destination_url = f"{WEBAPP_URL}/api/public/v1/destinations"
    headers = {
        "accept": "application/json",
        "content-type": "application/json",
        "authorization": f"Bearer {access_token}",
    }

    payload = {
        "definitionId": DESTINATION_DEF_ID,
        "workspaceId": WORKSPACE_ID,
        "name": name,
        "configuration": {
            "destinationType": "s3",
            "s3_bucket_region": S3_BUCKET_REGION,
            "access_key_id": S3_BUCKET_ACCESS_KEY,
            "secret_access_key": S3_BUCKET_SECRET_KEY,
            "s3_bucket_name": S3_BUCKET_NAME,
            "s3_bucket_path": s3_bucket_path,
            "format": {"format_type": "Parquet"},
        },
    }

    response = requests.post(destination_url, json=payload, headers=headers)
    if response.status_code == 200:
        print(f"Destination '{name}' created successfully.")
    else:
        print(
            f"Failed to create destination '{name}': {response.status_code} - {response.text}"
        )


def run_connection(access_token, connection_id):
```

```
    connection_url = f"{WEBAPP_URL}/api/public/v1/connections/{connection_id}"
    print(connection_url)
    headers = {
        "accept": "application/json",
        "content-type": "application/json",
    }

    payload = {"schedule": {"scheduleType": "cron"}, "namespaceFormat": None}
    response = requests.patch(connection_url, json=payload, headers=headers)
    if response.status_code == 200:
        print(f"Connection '{connection_id}' scheduled successfully.")
    else:
        print(
            f"Failed to scheduled connection '{connection_id}': {response.status_code} -
{response.text}"
        )


def create_connection(access_token, source_id, destination_id, name):
    # Retrieve all existing connections
    existing_connections = get_connections(access_token, WORKSPACE_ID)
    if any(conn["name"] == name for conn in existing_connections):
        print(f"Connection with name '{name}' already exists.")
        return

    connection_url = f"{WEBAPP_URL}/api/public/v1/connections"
    headers = {
        "accept": "application/json",
        "content-type": "application/json",
        "authorization": f"Bearer {access_token}",
    }
    payload = {
        "sourceId": source_id,
        "destinationId": destination_id,
        "name": name,
        "workspaceId": WORKSPACE_ID,
        "status": "active",
        "configurations": {
            "streams": [
                {"syncMode": "full_refresh_append", "name": "campaigns"},
                {
                    "syncMode": "full_refresh_append",
                    "name": "companies",
                },
                {"syncMode": "full_refresh_append", "name": "contact_lists"},
                {
                    "syncMode": "full_refresh_append",
                    "name": "contacts",
                },
                {
```

```
                "syncMode": "full_refresh_append",
                "name": "contacts_form_submissions",
            },
            {
                "syncMode": "full_refresh_append",
                "name": "contacts_list_memberships",
            },
            {
                "syncMode": "full_refresh_append",
                "name": "deal_pipelines",
            },
            {"syncMode": "full_refresh_append", "name": "deals_archived"},
            {
                "syncMode": "full_refresh_append",
                "name": "deals",
            },
            {"syncMode": "full_refresh_append", "name": "form_submissions"},
            {
                "syncMode": "full_refresh_append",
                "name": "engagements",
            },
            {"syncMode": "full_refresh_append", "name": "engagements_emails"},
            {"syncMode": "full_refresh_append", "name": "forms"},
            {"syncMode": "full_refresh_append", "name": "goals"},
            {"syncMode": "full_refresh_append", "name": "line_items"},
            {"syncMode": "full_refresh_append", "name": "owners"},
            {"syncMode": "full_refresh_append", "name": "products"},
            {"syncMode": "full_refresh_append", "name": "tickets"},
            {"syncMode": "full_refresh_append", "name": "workflows"},
        ]
    },
    "schedule": {"scheduleType": "cron", "cronExpression": "0 00 7 * * ?"},
    "dataResidency": "auto",
    "namespaceFormat": None,
    "nonBreakingSchemaUpdatesBehavior": "ignore",
}

response = requests.post(connection_url, json=payload, headers=headers)
if response.status_code == 200:
    print(f"Connection '{name}' created successfully.")
else:
    print(
        f"Failed to create connection '{name}': {response.status_code} - {response.text}"
    )


def yaml_to_json(yaml_data):
    if yaml_data is None:
        raise ValueError("yaml_data is None")
    json_data = {}
```

```
    # Process each item in the list
    for item in yaml_data:
        # Assuming each item in the list is a dictionary with 'naming_convention' key
        if isinstance(item, dict):
            user = item.get("naming_convention", "")

            if user:
                # Define names based on the user
                source_name = f"{user}_HubSpot"
                des_name = f"{user}_S3"
                con_name = f"{user}_HubSpot_S3"

                # Add the JSON structure to the json_data dictionary
                json_data[user] = {
                    "sources": {
                        "refresh_token": item.get("refresh_token"),
                        "name": source_name,
                    },
                    "destination": {
                        "name": des_name,
                        "s3_bucket_path": f"{user}/${{YEAR}}/${{MONTH}}/${{DAY}}",
                    },
                    "connections": {"name": con_name},
                }

    return json_data


def load_yaml_file(file_path):
    try:
        with open(file_path, "r") as file:
            data = yaml.safe_load(file)
            if data is None:
                print(
                    f"Warning: The file {file_path} is empty or contains invalid YAML."
                )
            return data
    except FileNotFoundError:
        print(f"File not found: {file_path}")
        return []
    except yaml.YAMLError as e:
        print(f"Error reading YAML file: {e}")
        return []


# Function to read YAML file and convert to JSON format
def process_yaml_to_json(yaml_file_path, json_file_path):
    # Load YAML data
    yaml_data = load_yaml_file(yaml_file_path)
```

```
    if yaml_data is None:
        print(f"No data found in {yaml_file_path}")
        return

    # Convert YAML data to JSON format
    json_data = yaml_to_json(yaml_data)

    # Write JSON data to a file
    with open(json_file_path, "w") as json_file:
        json.dump(json_data, json_file, indent=4)


# Test load_yaml_file
yaml_data = load_yaml_file(yaml_file_path)

# Test yaml_to_json
if yaml_data is not None:
    json_data = yaml_to_json(yaml_data)

    # Save JSON data
    with open(json_file_path, "w") as json_file:
        json.dump(json_data, json_file)


def airbyte_app(refresh_token):
    print("start airbyte new_refresh_token")

    access_token = get_access_token()
    if access_token:
        workspace_id = WORKSPACE_ID
        print("read file")
        process_yaml_to_json(yaml_file_path, json_file_path)
        with open("user_data.json", "r") as file:
            user_data = json.load(file)

        # Create sources and destinations
        for user, data in user_data.items():
            exist_user = refresh_token

            source_params = data.get("sources", {})
            if source_params:
                if source_params.get("refresh_token") == refresh_token:
                    name = data["sources"].get("name", "")
                    print("target name", name)
                    create_source(access_token, exist_user, name)

            destination_params = data.get("destination", {})
            if destination_params:
                if source_params.get("refresh_token") == refresh_token:
                    create_destination(
```

```
                access_token,
                destination_params.get("name", ""),
                destination_params.get("s3_bucket_path", ""),
            )

    # Get sources and destinations after creation
    source_list = get_sources(access_token, workspace_id)
    destination_list = get_destinations(access_token, workspace_id)

    # Create connections if needed
    for user, data in user_data.items():
        # Move extraction of parameters inside the loop
        source_params = data.get("sources", {})
        destination_params = data.get("destination", {})
        connection_params = data.get("connections", {})

        if source_params.get("refresh_token") == refresh_token:
            source_token = source_params.get("refresh_token")
            print("source_token", source_token)
            print("refresh_token", refresh_token)

            if connection_params:
                source_name = source_params.get("name", "")
                destination_name = destination_params.get("name", "")
                connection_name = connection_params.get("name", "")

                result = {
                    "connection_name": connection_params.get("name", ""),
                    "source_name": source_params.get("name", ""),
                    "destination_name": destination_params.get("name", ""),
                }

                print("result", result)

                # Find source and destination IDs
                source_id = next(
                    (
                        src["sourceId"]
                        for src in source_list
                        if src["name"] == source_name
                    ),
                    None,
                )
                destination_id = next(
                    (
                        dest["destinationId"]
                        for dest in destination_list
                        if dest["name"] == destination_name
                    ),
                    None,
```

```
                )

                if source_id and destination_id:
                    create_connection(
                        access_token, source_id, destination_id, connection_name
                    )
                else:
                    print(
                        f"Source ID or Destination ID not found for '{source_name}' or
'{destination_name}'"
                    )
```