Technical Documentation on the process of Aurora to Quickbase Integration

Date:		**08/07/2025**	
Version:		**0.1**	
Date:	**Version**	**Elaboration responsible:**	**Reviewer responsible:**
06/05/2024	0.1	Aranza Armas	

Content Table

Introduction

This document describes the design and implementation of an automated data integration process between Aurora Solar and Quickbase CRM. The solution was developed using Python and API-based communication to retrieve the latest solar design data and register it directly into a Quickbase sandbox table. The project replaces a previously manual data transfer process, improving speed, accuracy, and operational efficiency.

Objectives

The primary objective of this project is to automate the retrieval of solar design information from Aurora Solar, transform and map the extracted data to align with specific fields in a Quickbase table, and create new records in Quickbase without any manual intervention. The solution is designed to be reusable, scalable, and easy to maintain, allowing for continuous use across different projects and scenarios with minimal modification.

Scope

This solution covers the end-to-end process of integrating data between Aurora and Quickbase. It includes connecting to the Aurora Solar API using authenticated requests, fetching project-level metadata and the latest design details, extracting and transforming system attributes such as system size, module model, azimuth, annual energy production, and customer address. The solution also includes authenticating with the Quickbase API, formatting the data according to expected schema, submitting the data to the designated Proposal Sandbox table, and logging the results while handling errors gracefully. Multiple project IDs can be supported in a single execution loop.

However, this project does not include the ability to write data back into Aurora, nor does it support manual overrides of records in Quickbase. Additionally, this implementation does not currently include a full production deployment setup such as Docker containerization or CI/CD pipelines, though the codebase is structured to allow for such future enhancements.

Business Need

At present, project managers are responsible for manually transferring solar design data from Aurora into Quickbase. This manual process is slow, repetitive, and highly susceptible to human error. As the number of solar projects grows, the lack of automation in this workflow becomes a bottleneck, reducing team efficiency and limiting scalability. By automating this integration, we can significantly reduce time spent on repetitive tasks, eliminate data entry mistakes, and enable the business to handle a larger volume of projects without increasing operational overhead.

Prerequisites

To run and test this integration, the following items are required:

1) Aurora Solar API access:

- Tenant ID
- Project ID(s)
- Bearer token

2) Quickbase developer account with:

- User token (QB-USER-TOKEN)
- Target Table ID (e.g. bu8e7bjy4)
- Field mapping details

3) Python 3.8+ installed locally or in a virtual environment.

4) Required Python packages installed (via pip install -r requirements.txt)

5) A config.json file containing API credentials and a list of projects to process.

Project Description

This project implements a robust and modular Python-based integration pipeline that automates the process of transferring solar system design data from the Aurora Solar API to a Quickbase CRM table. It was developed in response to a scenario where project managers were manually entering system design information, creating a time-consuming and error-prone workflow.

The script identifies the most recent design associated with each Aurora project, extracts a defined set of fields, and creates a new record in a designated Quickbase sandbox table. It is structured to support configuration-driven execution and can be extended for broader use cases or other data sources.

Technical Assessment Description Link:
(https://docs.google.com/document/d/1hq9GJSoLZ4yYcideHYmaeuEP1wSk4oyPbC-JcFRkFRc/edit?usp=sharing)

Repository Link :
(https://github.com/Aranz44rmas/aurora-quickbase-integration)

1. Repository Structure

```
aurora-quickbase-integration/
├── src/
│ ├── extractor.py
├── config/
│ └── auth.json
├── utils/
│ └── logger
├── main.py
├── README.md
├── requirements.txt
└── .gitignore
```

Setup Instructions

1. Clone the repository:

```
git clone https://github.com/Aranz44rmas/aurora-quickbase-integration.git
cd aurora-quickbase-integration
```

2. Install dependencies:

```
pip install -r requirements.txt
```

3. Configure your API tokens and project list in `config/config.json`:

```
{
  "tenant_id": "YOUR_TENANT_ID",
  "bearer_token": "AURORA_BEARER_TOKEN",
  "auth": "QUICKBASE_USER_TOKEN",
  "table_id": "QUICKBASE_TABLE_ID",
  "project_ids": [
    { "id": "PROJECT_UUID_1", "label": "Demo Project A" },
    { "id": "PROJECT_UUID_2", "label": "Demo Project B" }
  ]
}
```

4. Run the integration:

```
python main.py
```

Logs will be saved under `logs/aurora_quickbase.log`.

Core Components

`AuroraProjectDataExtractor` (in `extractor.py`)

This class encapsulates the full ETL process, including API communication, data transformation, and Quickbase record creation

Key methods:

1) `__init__(self, tenant_id, project_id, bearer_token)`

Initializes the class instance with the required identifiers and credentials for API authentication and scoping.

Attributes initialized:

- self.tenant_id → Tenant ID from Aurora
- self.project_id → Specific project UUID to extract data from
- self.bearer_token → Bearer token string used for authenticating Aurora API requests

These values are passed from an external config JSON and used across all API calls.

2) `get_json_with_bearer(self, url)`

Performs an HTTP GET request with proper bearer token authentication and returns the parsed JSON response.

Inputs:

- url → A complete API endpoint string

Output:

- dict → JSON-parsed response content

Headers:

```
{
  "accept": "application/json",
  "authorization": "Bearer <token>"
}
```

Raises requests.HTTPError if status code is not 2xx. Logs error messages if requests fail.

3) get_designs(self)

Retrieves all available designs linked to the project_id, focusing only on metadata needed for later steps.

Output:

- A pandas DataFrame with the following fields:
    - 'design_id': Aurora design UUID
    - '6': Project name (Quickbase field 6)
    - '7': System size in kW (Quickbase field 7)

Raw API Endpoint Example:

```
GET
https://api-sandbox.aurorasolar.com/tenants/{tenant_id}/projects/{project_id}
/designs
```

Uses pandas.json_normalize() to flatten nested JSON structures Selects only relevant columns and renames them to match Quickbase field mapping

4) get_latest_proposal(self, design_id)

Finds and returns the latest proposal document linked to a specific solar design.

Inputs:

- design_id: UUID of the design to search for

Output:

- A single-row DataFrame with:
    - 'updated_at': Timestamp of proposal update (used for sorting)
    - '19': Proposal URL (Quickbase field 19)

Calls /proposals/default endpoint for the design. Normalizes and sorts by updated_at descending. Takes only the most recent proposal

5) get_summary(self, design_id)

Pulls core technical specifications from a solar design's summary.

Inputs:

- design_id: UUID of the solar design

Output:

- A single-row DataFrame with:
  - '8': Number of modules
  - '9': Module model name
  - '10': Annual energy production in kWh
  - '18': Azimuth angle of the first array

Data source:

`GET /tenants/{tenant_id}/designs/{design_id}/summary`

Structure:

- arrays: List of roof array objects (only the first is used)
- module.count and module.name are extracted from nested JSON
- 'energy_production.annual' is taken from the root level

6) get_address(self)

Purpose:
Retrieves and formats the customer's address information associated with the project.

Output:

- A single-row DataFrame with:
  - '12': Street address
  - '13': Street address 2 ("")
  - '14': City
  - '15': Region/state (abbreviated)
  - '16': Postal code
  - '17': Country

Applies a dictionary (us_states) to **map full state names to abbreviations** (e.g., {'California': 'CA', 'Texas': 'TX'}

7) extract_all_data(self)

Executes the full extraction pipeline:

- Retrieves project metadata
- Iterates through all designs
- Combines all data components into a normalized structure

Output:

- A DataFrame with final columns matching the Quickbase field IDs:
  - '6': Project Name
  - '7': System Size
  - '8': Number of Modules
  - '9': Module Model
  - '10': Annual Energy
  - '12'–'17': Address fields
  - '18': Azimuth
  - '19': Proposal Link

Filters to use the design with the most recent updated_at. Handles missing components with logging. Returns a list of merged records (can handle multiple designs per project, if needed)

8) format_for_quickbase(self, df, table_id, return_fields=None)

Formats the DataFrame row(s) into the exact JSON structure required by Quickbase's POST /v1/records endpoint.

Inputs:

- df: The final, merged DataFrame from extract_all_data
- table_id: Target Quickbase table ID
- return_fields: Optional list of field IDs to return in the response.

Output:

```
{ "to": "table_id",
  "data": [
    {
      "6": {"value": "Joanna Test"},
      "7": {"value": 10000},
      "9": {"value": "Q.PEAK DUO BLK ML-G10+ 400"},
      ...
    }
  ],
  "fieldsToReturn": [6, 7, 8, ...]}
```

Field '9' is **hardcoded** to "Q.PEAK DUO BLK ML-G10+ 400" (as there is in the predefined options).

9) `post_to_quickbase(self, payload, user_token)`

Sends the formatted data payload to Quickbase to create one or more records in the specified table.

Inputs:

- payload: JSON-formatted body with to, data, and optional fieldsToReturn
- user_token: Quickbase token (QB-USER-TOKEN)

Output:

- The parsed JSON response from Quickbase (e.g., record IDs, metadata)

Endpoint:

```
POST https://api.quickbase.com/v1/records
```

Headers:

```
{
  "Content-Type": "application/json",
  "QB-Realm-Hostname": "betterearthsolar.quickbase.com",
  "Authorization": "QB-USER-TOKEN <token>"
}
```

Prints and logs detailed error messages if the request fails. Returns None if the request is unsuccessful

`main.py' - Execution Driver

The main.py script serves as the **orchestrator** for the entire Aurora → Quickbase data pipeline. While the heavy lifting (API interaction, transformation, formatting) is handled in the AuroraProjectDataExtractor class, main.py coordinates the workflow across multiple projects by:

1. Load Configuration

```
config = load_config("config/config.json")
```

This line loads the external configuration file, which contains:

- The tenant_id and bearer_token for Aurora authentication
- The auth token for Quickbase
- A list of project_ids (each with an id and an optional label)
- The table_id in Quickbase to which records will be posted

This abstraction separates secrets and project-specific values from the main logic.

2. Iterate Over Project IDs

```
for project in project_ids:
    project_id = project['id']
    label = project.get('label', project_id)
```

This loop allows you to process multiple Aurora projects in a single run. Each project is logged with either a human-readable label or the raw UUID. This setup supports batch ETL and is easily expandable.

3. Instantiate the Extractor Class

```
extractor = AuroraProjectDataExtractor(tenant_id, project_id,
bearer_token)
```

For each project, the script creates an instance of AuroraProjectDataExtractor, passing in the relevant credentials. This instantiation ensures that each project is scoped correctly and allows for clean API calls under the hood.

4. Extract, Transform & Combine Data

```
final_df = extractor.extract_all_data()
```

This is the core ETL step:

- extract_all_data() internally calls:
  - get_designs() → design metadata
  - get_latest_proposal() → fetch latest proposal
  - get_summary() → extract solar specs
  - get_address() → standardize location
- The outputs are merged into a single DataFrame per project.
- If data is incomplete or missing, it logs a warning and skips that project.

5. Format Payload for Quickbase

```
payload = extractor.format_for_quickbase(final_df, table_id,
return_fields=fields_to_return)
```

This method:

- Converts the DataFrame into a JSON payload structure compatible with Quickbase's POST /v1/records endpoint.
- Hardcodes field 9 (Module Model) to "Q.PEAK DUO BLK ML-G10+ 400" as per spec.
- Supports optional fieldsToReturn list to get confirmation on specific fields (like record ID).

6. Post to Quickbase

```
result = extractor.post_to_quickbase(payload, user_token=auth)
```

This step sends the payload via a secure HTTP request to Quickbase. The method handles:

- Authentication with the Quickbase user token
- Structured error responses and API exceptions
- Printing and logging of status and result

If successful, Quickbase responds with:

```
{
  "data": [
    {
      "3": { "value": 197 },
      "6": { "value": "Joanna Test" }
    }
  ]
}
```

7. Log Status and Outcomes

```
if result:
    print(" Record updated")
else:
    print("Quickbase Post Failed")
```

The result of each execution is printed directly to the console for real-time visibility, allowing the user to track progress as records are processed. Simultaneously, all events—successful or failed—are written to a dedicated log file named aurora_quickbase.log, which serves as an auditable trail for future debugging or historical review. In the event of an exception, the script leverages a try/except block that logs the error with full contextual information, such as the project label and ID, and gracefully continues processing the remaining projects without halting the entire pipeline.

Considerations

The solution currently implements a wide range of features that align with best practices for data integration. These include secure API authentication with both Aurora and Quickbase, a configuration-driven loop that supports multiple project IDs, and intelligent detection of the most recent solar design based on the updated_at timestamp. It also includes normalization of deeply nested JSON structures from Aurora—such as roof array, module details, and address components—and accurate mapping of fields to the corresponding Quickbase field IDs. The payload is formatted according to Quickbase's schema requirements, and the entire process is supported by structured logging and resilient error handling.

Looking forward, there are several opportunities for optimization and extension. One key improvement would be implementing retry logic with exponential backoff to handle transient API timeouts or temporary failures. Another is replacing the static token approach with .env support or integrating with AWS Secret Manager using an ARN, improving credential security and flexibility. The project could also benefit from a CI/CD infrastructure for automated deployment and version control. Additionally, the solution could be extended to dynamically retrieve new project_ids and corresponding designs by leveraging Aurora's design request endpoint. This enhancement would allow the script to query design requests, filter out entries marked as "designer rejected," and sort or select designs based on key timestamps such as submission_time, completed_at, reviewed_at, and designer_rejected_at. This logic would enable the system to prioritize only valid, recently completed designs and improve overall automation intelligence.