Code Flow Guide : AWS Migration App

This document provides a detailed reference for the API endpoints of the AWS Migration App, along with an overview of the code flow for both the backend and frontend. It also includes instructions for adding new API endpoints and frontend features, enabling the team to extend the application efficiently.

Tech Stack Overview

- **Frontend**: React (TypeScript), Vite, Tailwind CSS, Axios, React Router, Framer Motion
- **Backend**: Python, FastAPI, SQLAlchemy, boto3, PostgreSQL (managed via pgAdmin)
- **Database**: PostgreSQL (database name: `aws_migration`)

API Endpoints (steps.py)

The backend exposes API endpoints for managing migration steps, defined in `server/Backend/app/api/routes/steps.py`. These endpoints handle AWS-related checks and actions, storing results in the `aws_migration` database. All routes are under a base path corresponding to the migration phase (e.g., `/assess-existing/`).

1. Assess Existing Phase Endpoints

These endpoints perform checks on AWS accounts during the "Assess Existing" phase of the migration process.

Endpoint	Method	Description	Parameters
`/assess-existing/check_ram`	GET	Checks for resources shared via AWS RAM.	`account_id` (query parameter, required)
`/assess-existing/check_admin_services`	GET	Checks for delegated admin services (e.g., AWS Backup, GuardDuty, Inspector).	`account_id` (query parameter, required)

`/assess-existing/cost_explorer_data`	GET	Checks Cost Explorer data and CUR reports for the account.	`account_id` (query parameter, required)
`/assess-existing/check_savings`	GET	Checks for active Reserved Instances and Savings Plans, including utilization.	`account_id` (query parameter, required)
`/assess-existing/check_policies`	GET	Scans IAM, S3, KMS, SQS, SNS, Lambda, and Secrets Manager policies for AWS Organizations/OU references.	`account_id` (query parameter, required)
`/assess-existing/check_stacksets`	GET	Checks if CloudFormation StackSets use AWS Organizations integration.	`account_id` (query parameter, required)
`/assess-existing/create_iam_admin`	GET	Creates a fallback IAM admin user for the account in case of SSO failure.	`account_id` (query parameter, required)

2. Step Execution History and Status Endpoints

These endpoints retrieve execution details for migration steps.

Endpoint	Method	Description	Parameters
`/{phase_type}/{step_slug}/latest`	GET	Retrieves the latest execution result for a specific step.	`phase_type` (e.g., `assess-existing`), `step_slug` (e.g., `check_ram`), `account_id` (query parameter, required)
`/{phase_type}/{step_slug}/history`	GET	Retrieves the execution history for a specific step.	`phase_type` (e.g., `assess-existing`), `step_slug` (e.g., `check_ram`), `account_id` (query parameter, required)

Step ID and Phase Mapping

- **Step IDs**:
  - `check_ram`: 1
  - `check_admin_services`: 2
  - `cost_explorer_data`: 3
  - `check_savings`: 4
  - `check_policies`: 5
  - `check_stacksets`: 6
  - `create_iam_admin`: 8
- **Phase to Step Mapping**:
  - `assess-existing`: Steps [1, 2, 3, 4, 5, 6, 8]
  - `prepare-new`, `migrate`, `verify`, `post-migration`: Currently empty (no steps assigned)

Example API Usage

Check RAM Shared Resources:

GET http://localhost:8000/assess-existing/check_ram?account_id=<account_id>

1. *Response*: JSON with step ID, title, status, result, logs, and execution time.

Get Latest Execution for check_ram:

GET http://localhost:8000/assess-existing/check_ram/latest?account_id=<account_id>

2. *Response*: JSON with the latest execution details.

Get Execution History for check_ram:

GET http://localhost:8000/assess-existing/check_ram/history?account_id=<account_id>

3. *Response*: JSON array of historical execution details.

Notes

- All endpoints require a valid `account_id` to identify the AWS account.
- Responses are structured using Pydantic models (`StepResponse`) defined in `app/db/schemas.py`.
- Results and logs are stored in the `aws_migration` database for persistence.

Backend Code Flow

The backend is structured to separate API routes, business logic, and database operations, ensuring modularity and maintainability.

1. **App Initialization** (`main.py`):
    - The FastAPI app is initialized in `server/Backend/main.py`.
    - It loads configurations from `app/core/config.py` and environment variables from `.env`.
    - API routes from `app/api/routes/` (e.g., `steps.py`, `account_management.py`) are included.
2. **API Request Handling** (`app/api/routes/steps.py`):
    - When a frontend request hits an endpoint (e.g., `/assess-existing/check_ram`), the corresponding route handler:
        - Validates input (e.g., `account_id`) using FastAPI's query parameters and Pydantic models.
        - Ensures the step is registered in the database using helper functions from `app/db/PG_queries.py`.
        - Calls the relevant AWS function from `app/services/aws_services.py`.
        - Collects results, logs, and status, saves the execution to the database, and returns a structured response.
3. **AWS Business Logic** (`app/services/aws_services.py`):
    - Functions like `check_ram_shared_resources`, `check_delegated_admins`, etc., use boto3 to interact with AWS services.
    - Each function:
        - Obtains a boto3 session for the specified `account_id` using `get_aws_session` (from `app/services/aws_client_helper.py`).
        - Performs the AWS operation (e.g., listing RAM resources, checking policies).
        - Handles errors and formats results for the API response.
    - Example: `check_ram_shared_resources` lists shared resources, categorizes them, and returns a summary.
4. **Database Layer** (`app/db/`):
    - `PG.py`: Manages the PostgreSQL connection to the `aws_migration` database.
    - `session.py`: Handles database sessions and transactions.

- `PG_queries.py`: Contains helper functions for querying and saving step executions/results.
- `schemas.py`: Defines Pydantic models for request/response validation and SQLAlchemy ORM models for database tables.
- `migrations.py`: Sets up the database schema (run during setup).

5. **Data Flow**:
   - **Request**: Frontend sends a request (e.g., `GET /assess-existing/check_ram?account_id=123`).
   - **Route**: `steps.py` validates the request and calls the corresponding `aws_services.py` function.
   - **AWS Logic**: st`:` aws_services.py` performs the AWS operation and returns results.
   - **Database**: Results/logs are saved via `PG_queries.py`.
   - **Response**: Structured JSON response is sent back to the frontend.

Frontend Code Flow

The frontend is a React application that provides a user interface for managing AWS migrations, built with Vite and TypeScript.

1. **App Initialization**:
   - The app starts in `src/main.tsx`, which renders `src/App.tsx`.
   - `App.tsx` sets up:
     - React Router for routing to pages (e.g., `Dashboard.tsx`, `MigrationJourney.tsx`).
     - Context providers (`ThemeContext`, `AccountContext`, `MigrationContext`, `SidebarContext`) for global state.
     - Layout components (`Header.tsx`, `Sidebar.tsx`) for consistent navigation.
2. **Routing**:
   - Each file in `src/pages/` (e.g., `Dashboard.tsx`, `Login.tsx`, `MigrationJourney.tsx`) maps to a route defined in `App.tsx`.
   - Navigating to a route (e.g., `/dashboard`) renders the corresponding page component.
3. **State Management**:
   - **Global State**: Managed via React Contexts in `src/context/` (e.g., `AccountContext` for selected AWS account, `MigrationContext` for migration progress).

        - Components use custom hooks (e.g., `useAccount`, `useMigration`) to access/update global state.
    - **Local State**: Managed with React's `useState` or `useReducer` for UI-specific needs (e.g., modal visibility, form inputs).
4. **API Calls**:
    - API interactions are abstracted in `src/services/` (e.g., `accountApi.ts`, `migrationApi.ts`).
    - Pages/components call these services (using Axios) to fetch or update data (e.g., `accountApi.getAccounts()` in `Dashboard.tsx`).
    - Example: Triggering a migration step calls an endpoint like `/assess-existing/check_ram`, and the response is displayed using components like `StepCard.tsx` or `StepLogs.tsx`.
5. **UI Components**:
    - Reusable UI elements (e.g., `Button.tsx`, `Card.tsx`, `Tabs.tsx`) are in `src/components/ui/`.
    - Feature-specific components (e.g., `AWSAuthForm.tsx`, `MigrationProgress.tsx`) are in folders like `src/components/migration/`.
    - Components receive data via props or context and manage local state as needed.
6. **Data Flow**:
    - **Request**: User triggers an action (e.g., clicking a button to check RAM resources).
    - **API Call**: The component calls a service function (e.g., `migrationApi.checkRamResources`).
    - **Response**: The backend returns data (status, logs, results), which is rendered in components like `StepDashboard.tsx` or `StepLogs.tsx`.

Adding a New API Endpoint

To add a new migration step API (e.g., `check_new_feature` in the `assess-existing` phase):

Implement AWS Logic (`app/services/aws_services.py`):
Add a new function to perform the AWS operation:

```
def check_new_aws_feature(db: Session = None, account_id: str = None):
    session = get_aws_session(account_id)  # From aws_client_helper.py
    # Example: Check a new AWS service
    client = session.client('some-service')
```

```
    result = client.some_api_call()
    return {"success": True, "data": result}
```

1. **Expose the Function as an API Route** (`app/api/routes/steps.py`):
   Import the new function:

   ```
   from app.services.aws_services import check_new_aws_feature
   ```

Add a new route:

```
@router.get("/assess-existing/check_new_feature", response_model=StepResponse)
async def check_new_feature(account_id: str = Query(None), db: Session =
Depends(get_db)):
    step_id = <new_unique_step_id>  # e.g., 9
    # Register step in DB if needed (using PG_queries.py)
    result = check_new_aws_feature(db, account_id)
    # Save execution to DB (using PG_queries.py)
    execution = save_execution(db, step_id, account_id, result)
    return {"step_id": step_id, "status": result["success"], "result": result["data"], "logs": [],
"execution_time": ...}
```

2. **Update Step and Phase Mappings**:
   In `steps.py`, update:
   - `STEP_IDS`: Add `check_new_feature: <new_unique_step_id>`.
   - `PHASE_STEPS`: Append the new `step_id` to the `assess-existing` list.
3. **Update Frontend** (Optional):
   - Add a button or action in a relevant page (e.g., `MigrationJourney.tsx`) to call the new endpoint via `migrationApi.ts`.
   - Update components (e.g., `StepCard.tsx`) to display the new step's results.

Adding a New Frontend Feature

To extend the frontend with a new page, context, or component:

Adding a New Page (e.g., Reports Page)

Create the Page Component (`src/pages/Reports.tsx`):

```
 import React from 'react';

const Reports: React.FC = () => {
  return (
    <div className="p-4">
      <h1 className="text-2xl font-bold">Reports</h1>
      <p>This is the reports page.</p>
    </div>
  );
};

export default Reports;
```

1. **Add the Route** (`src/App.tsx`):
   Import and add the route:

   ```
    import Reports from './pages/Reports';

   // Inside <Routes> block:
   <Route path="/reports" element={<Reports />} />
   ```

2. **Update Navigation**:
   - Add a link to `/reports` in `Sidebar.tsx` or `Header.tsx` for navigation.

Adding a New Context (e.g., Notification Context)

Create the Context (`src/context/NotificationContext.tsx`):

```
import React, { createContext, useContext, useState } from 'react';
const NotificationContext = createContext(null);
export const NotificationProvider = ({ children }) => {
  const [message, setMessage] = useState('');
  return (
    <NotificationContext.Provider value={{ message, setMessage }}>
      {children}
    </NotificationContext.Provider>
  );};
```

```
export const useNotification = () => useContext(NotificationContext);
```

1. **Wrap the App** (`src/App.tsx` or `src/main.tsx`):

   ```
   import { NotificationProvider } from './context/NotificationContext';

   // Wrap routes or app:
   <NotificationProvider>
     <App />
   </NotificationProvider>
   ```

2. **Use the Context**:
   In any component:

   ```
   import { useNotification } from '../context/NotificationContext';

   const { message, setMessage } = useNotification();
   // Example: setMessage('New notification!');
   ```

3. **Adding a New Component**

- Create a new component in `src/components/ui/` or a feature-specific folder (e.g., `src/components/migration/NewComponent.tsx`).
- Import and use it in relevant pages or components.

Notes for Developers

- **Modularity**: The backend separates API routes (`steps.py`), AWS logic (`aws_services.py`), and database operations (`PG_queries.py`) for clean, testable code.
- **Frontend Structure**: Pages, components, and services are organized for reusability and scalability.
- **Database**: All step executions are stored in the `aws_migration` database, accessible via pgAdmin for inspection.
- **Extensibility**: New APIs or features follow the same pattern, leveraging existing services, contexts, and components.
- **Testing**: Use `http://localhost:8000/docs` for API testing and `http://localhost:5173` for frontend testing.