**Technical Documentation: Alpha/Beta Command Center (v2 - Auth Update)**

**Overview**

The Product Operations Alpha/Beta Command Center is a web-based application designed to streamline various tasks related to client recruitment and communication for Alpha/Beta programs. It provides distinct tools accessed via a central landing page:

1. **List Generation:** Generates new client recruitment lists using natural language prompts processed by Snowflake Cortex AI.
2. **List Scrubbing/Filtering:** Filters existing client lists based on specified criteria, also leveraging Cortex AI.
3. **Marketing Content Generation:** Creates templated DOCX invite documents (e.g., Opt-In/Opt-Out invites) populated with data fetched from Snowflake based on a selected list ID.
4. **Qualtrics Survey Creation:** Automates the creation of standardized Opt-In/Opt-Out surveys in Qualtrics based on feature details, either manually entered or derived from generated lists.

The system integrates a Node.js backend server, HTML/JavaScript frontend, PowerShell and Python scripts for orchestration and specific tasks, and leverages Snowflake for data storage, processing, and AI capabilities. Background processes (Python jobs and Snowflake tasks) ensure underlying data tables are kept up to date. User authentication is managed via **username/password verification against a dedicated Snowflake user table (CR\_APP\_USERS)**, using secure password hashing (bcrypt), and maintained across requests using server-side sessions (express-session).

**Main Components**

1. **Web Application (HTML + JavaScript)**
   * **Purpose:** Provides user interfaces for the login/password setup process and the different operational tools.
   * **Files:**
     + public/home.html: Central landing page displaying available tools. It **does not handle login forms**. On load, its embedded JavaScript checks for an active session via /auth/check-session. If no valid session exists, it immediately **redirects the browser to /login**. If a session is active, it displays the tool links and a logout button.
     + public/login.html: **New dedicated login page.** Contains the HTML structure for a multi-step authentication flow: an initial form to enter the username, a subsequent form (initially hidden) for first-time password setup (new password + confirmation), and another form (initially hidden) for returning users to enter their existing password. Also includes areas for displaying status or error messages.
     + public/login.js: **New JavaScript file** linked by login.html. Manages the entire client-side authentication workflow: handles submissions for username check, password setup, and login forms; communicates with the backend /auth/\* endpoints; shows/hides the appropriate forms based on server responses; displays feedback/errors; redirects to / upon successful login or setup.
     + public/list\_tools.html: UI for List Generation and List Scrubbing. Features interactive prompt building, context input, file naming, filter selection, and a modal for scrubbing options. **Contains a session check script** in its <head> which calls /auth/check-session on page load. If the session is invalid or missing, the script **redirects the browser to /login**. Relies on the active server-side session for authenticated backend API calls.
     + public/marketing\_content.html: UI for generating DOCX invites. **Contains a session check script** in <head> redirecting to /login if needed. If authenticated, it dynamically loads and displays lists associated with the user via an initial call to the /get-lists endpoint. Includes list search functionality and a modal for template selection. Relies on the active server-side session.
     + public/qualtrics\_survey.html: UI for Qualtrics survey creation. **Contains a session check script** in <head> redirecting to /login if needed. If authenticated, it dynamically loads and displays lists associated with the user via /get-lists. Includes a manual form for entering feature details directly. Relies on the active server-side session.
   * **Functionality:**
     + Handles the complete user login and initial password setup flow via login.html and login.js.
     + Collects user inputs for various tools (prompts, filenames, filters, feature details).
     + Enforces authentication by redirecting unauthenticated users from tool pages to the login page.
     + Dynamically constructs prompts based on selected filters (List Tools).
     + Fetches necessary data via Node.js backend API calls (e.g., user lists, feature definitions).
     + Submits data via POST requests to specific Node.js API endpoints for processing tasks.
     + Displays results, confirmation messages, or error messages returned from the server.
     + Handles user logout via a POST request to /auth/logout.
2. **Node.js Server (server.js + modules)**
   * **Purpose:** Acts as the central backend API, orchestrator, and session manager. It routes requests, enforces authentication/authorization, interacts with the database, and spawns external scripts.
   * **Framework:** Express.js
   * **Key Files:**
     + server.js: Initializes the Express app, configures middleware (static files, body parsing, session management), sets up logging, defines application routes (serving HTML pages, tool APIs), mounts the authentication router, starts the HTTP server.
     + authRoutes.js: Defines routes specifically for authentication, prefixed with /auth (e.g., /auth/login). Delegates request handling to functions in authController.js.
     + authController.js: Contains the core logic for each authentication route. Handles request validation, interacts with dbAuth.js for user data, performs password hashing and comparison using bcrypt, manages req.session properties (setting req.session.username on success), and sends JSON responses to the client.
     + dbAuth.js: Contains functions dedicated to querying and updating the CR\_APP\_USERS table in Snowflake (e.g., finding users, updating password hashes, updating last login timestamp). Uses database utilities from dbUtils.js.
     + dbUtils.js: Provides shared database functionality, primarily the executeSnowflakeQuery function which handles establishing Snowflake connections (using credentials from environment variables), executing parameterized queries via odbc, and closing connections. Also includes buildConnectionString.
   * **Middleware:**
     + express.static: Serves static files (HTML, JS, CSS, images) from the public directory.
     + body-parser: Parses incoming request bodies (JSON and URL-encoded).
     + express-session: Manages user sessions using signed cookies. Configured with a secret (from SESSION\_SECRET env var), cookie settings (max age, httpOnly), and storage options.
     + Custom Request Logger: Logs incoming requests, including method, URL, and the username from the session (or 'anonymous'). Redacts passwords from logged request bodies.
     + requireVerifiedUser: Custom middleware applied to routes that require authentication (e.g., tool pages, tool API endpoints). Checks for the presence of req.session.username. If missing, it redirects browser requests for HTML pages to /login, and responds with a 401 Unauthorized status for API requests.
   * **Configuration:**
     + Hardcoded Paths: File paths for PowerShell/Python scripts, DOCX templates, and output directories are defined in server.js. (Consider externalizing).
     + Snowflake Credentials: API\_USERNAME, API\_PASSWORD are read from environment variables (used by dbUtils.js).
     + Session Secret: SESSION\_SECRET environment variable is used to sign session ID cookies. CRITICAL for security.
     + bcrypt salt rounds are defined in authController.js.
   * **Key API Endpoints:**
     + **(Auth - Defined in authRoutes.js)**
       - /auth/check-user-status (POST): Input: { username }. Checks CR\_APP\_USERS. Output: { status: 'INVALID' | 'INACTIVE' | 'NEEDS\_SETUP' | 'NEEDS\_LOGIN', username?: string }.
       - /auth/setup-password (POST): Input: { username, newPassword, confirmPassword }. Verifies user status, validates password, hashes using bcrypt.hash, updates PASSWORD\_HASH in CR\_APP\_USERS, creates req.session.username. Output: { success: true/false, message?: string, username?: string }.
       - /auth/login (POST): Input: { username, password }. Verifies user status, fetches hash from CR\_APP\_USERS, compares using bcrypt.compare, creates req.session.username. Output: { success: true/false, message?: string, username?: string }.
       - /auth/logout (POST): Destroys the current session (req.session.destroy) and clears the client-side cookie. Output: { success: true/false, message?: string }.
       - /auth/check-session (GET): Checks if req.session.username exists. Output: { loggedIn: true/false, username?: string }.
     + **(Tools - Defined in server.js, Protected by requireVerifiedUser)**
       - /run-powershell (POST): Retrieves req.session.username. Spawns list\_generation\_power\_shell.ps1 passing username, prompt, filename. Returns PS script output.
       - /run-list-filter (POST): Retrieves req.session.username. Spawns list\_filter\_power\_shell.ps1 passing username, prompt, filename, saveToDownloads flag. Returns PS script output.
       - /generate-content (POST): Retrieves req.session.username. Parses list ID, fetches feature data (getFeatureContentData), generates DOCX using docxtemplater, saves file, logs event (logContentCreation using session username). Returns path or error.
       - /get-lists (GET): Retrieves req.session.username. Fetches list metadata for the user from cr\_user\_requests (getAllListMetadata). Returns JSON list or error.
       - /create-qualtrics-survey (POST): Retrieves req.session.username. Spawns create\_qualtrics\_survey.py passing survey type, feature details, and formatted session username. Returns JSON result from Python script.
     + **(Public - Defined in server.js)**
       - /search-features-by-criteria (GET): Searches cr\_alpha\_content. Unprotected.
       - /get-feature-name (GET): Fetches client-facing name for a feature key from temp\_feature\_master\_6\_13\_2022. Unprotected.
   * **Direct Snowflake Interaction:** Handled via helper functions in dbUtils.js, dbAuth.js, and within server.js (for non-auth tables like cr\_alpha\_content, cr\_user\_requests etc.). These functions use the imported executeSnowflakeQuery. Username formatting (e.g., adding @domain.com) is handled within specific helpers as needed for different tables/scripts.
3. **PowerShell Scripts (list\_generation\_power\_shell.ps1, list\_filter\_power\_shell.ps1)**
   * **Purpose:** Orchestrate list generation and filtering using Snowflake Stored Procedures and Cortex AI.
   * **Parameters:** $var1 (username - now supplied directly from Node.js session), $var2 (prompt/filters), $var3 (base filename), $SaveToDownloads (switch, only for filter script).
   * **Functionality:** Retrieve machine Snowflake credentials (from env vars); connect via ODBC; call respective Snowflake Stored Procedure (alpha\_beta\_list\_generation or alpha\_beta\_list\_generation\_filter); parse the JSON response containing the generated SQL; wrap or modify the SQL as needed; execute the final SQL query against alpha\_beta\_list\_generation table; log the results data rows into alpha\_beta\_list\_generation\_results (tagging with $var1 username and timestamp); export the results to an Excel file (to OneDrive or Downloads based on $SaveToDownloads); log the initial request details to cr\_user\_requests (tagging with $var1 username, prompt, final SQL, filename).
4. **Python Script (create\_qualtrics\_survey.py)**
   * **Purpose:** Create a Qualtrics survey via API based on provided details and log the event.
   * **Execution:** Invoked by Node.js server (spawn).
   * **Parameters (Command Line):** Survey Type ('Opt-In' or 'Opt-Out'), Feature Number, Alpha/Beta Stage, Client-Facing Feature Name, Requesting Username (base username without domain, provided by Node.js session).
   * **Functionality:** Load base QSF template file; retrieve Qualtrics API token/datacenter and Snowflake credentials from environment variables; substitute feature details into the QSF structure; make API call to Qualtrics to create the survey; connect to Snowflake (using env var credentials); query UX\_PROD.SURVEY.qualtrics\_user table to find the Qualtrics User ID corresponding to the requesting username (for sharing); attempt to share the newly created survey via Qualtrics API; log survey details (including Qualtrics ID, URL, feature info, requesting username, and sharing attempt status) to the cr\_opt\_in\_out\_surveys table in Snowflake; print final JSON result (including success flag, surveyId, surveyUrl, sharingStatus) to standard output for Node.js to capture.
5. **Snowflake Stored Procedures (alpha\_beta\_list\_generation, alpha\_beta\_list\_generation\_filter)**
   * **Purpose:** Leverage Snowflake Cortex AI (SNOWFLAKE.CORTEX.COMPLETE) to generate SQL queries or filtering clauses based on natural language prompts.
   * **Functionality:** Receive prompt and user identifier; construct a detailed prompt for the AI including schema information, required output format, filtering logic instructions, context ID handling, feature exclusion logic, and sampling instructions; execute the Cortex AI function; parse and clean the AI's SQL output; return the generated SQL in a JSON structure.
6. **Snowflake Database**
   * **Purpose:** Central data repository for source data, generated lists, historical information, feature content, user credentials, and operational logs.
   * **Key Tables:**
     + CR\_APP\_USERS (New): Stores USERNAME (PK, lowercase), nullable PASSWORD\_HASH (bcrypt), IS\_ACTIVE flag, LAST\_LOGIN\_TS, CREATED\_TS. Source of truth for application login.
     + alpha\_beta\_list\_generation: Source client/context data used for list generation and scrubbing.
     + cr\_full\_invite\_list: Historical record of previously invited clients/contexts for specific features, used for exclusion logic.
     + alpha\_beta\_list\_generation\_results: Stores the actual data rows output by the list generation/scrubbing PowerShell scripts. Includes RUN\_BY (session username) and RUN\_AT timestamp. [Metadata Table]
     + cr\_user\_requests: Logs the metadata of each list generation/scrubbing request (session user, prompt, generated SQL, output filename). Written by PowerShell scripts. [Metadata Table]
     + cr\_alpha\_content: Contains detailed content definitions for features (client-facing name, descriptions, dates, links, etc.). Read by Node.js for DOCX generation.
     + cr\_alpha\_beta\_content\_creation: Logs each DOCX marketing content generation event (session user formatted as user@domain, timestamp, content type, list ID used). Written by Node.js. [Metadata Table]
     + cr\_opt\_in\_out\_surveys: Logs details of each Qualtrics survey creation attempt (feature details, survey type, generated Qualtrics ID/URL, requesting session user, sharing status, timestamp). Written by Python script. [Metadata Table]
     + Staging Tables (e.g., CR\_ALPHA\_OPT\_INS, CR\_CLIENT\_LIST): Intermediate tables populated by background Python jobs, used by Snowflake tasks.
     + UX\_PROD.SURVEY.qualtrics\_user: Contains mappings between usernames and Qualtrics internal User IDs, used by the Python script for survey sharing.
     + EMERALD\_CITY\_WFA\_DATA: Source table used by the daily Snowflake task to find new eligible users for the application.
   * **Key Tasks:**
     + ADD\_NEW\_APP\_USERS\_TASK (New): Runs daily (e.g., midnight UTC). Executes an INSERT...SELECT statement to add new users from EMERALD\_CITY\_WFA\_DATA into CR\_APP\_USERS if they meet specific criteria (active, correct manager, not already present).
     + ALPHA\_BETA\_LIST\_GENERATION\_TASK: Runs daily. Calls a stored procedure to refresh the main alpha\_beta\_list\_generation data table.
     + CR\_FULL\_INVITE\_LIST\_TASK: Runs daily. Calls a stored procedure to truncate and reload the CR\_FULL\_INVITE\_LIST table from staging data.
7. **Excel File Output (.xlsx)**
   * **Purpose:** Provide user-friendly, downloadable results of list generation and scrubbing operations.
   * **Location:** Saved either to a shared OneDrive location (C:\...\Client List Pulls) or the user's local Downloads folder, depending on the operation (generation vs. scrubbing) and the "Save to Downloads" flag selection during scrubbing.
   * **Generation:** Created by the PowerShell scripts using the ImportExcel module.
   * **Common Sheets:** Typically includes "Metadata" (prompt, SQL, parameters), "ALLCONTACTS" (the main list data), and potentially "ExcludedContexts" or "ExcludedContextsDetails" (listing contexts filtered out).
8. **DOCX File Output (.docx)**
   * **Purpose:** Generate standardized, ready-to-use marketing invite documents (Opt-In/Opt-Out).
   * **Location:** Saved to a fixed shared OneDrive path (C:\...\Invite Drafts) defined by OUTPUT\_DIR\_PATH in server.js.
   * **Generation:** Created by the Node.js server (/generate-content endpoint) using the docxtemplater and pizzip libraries. DOCX templates (.docx files in the templates directory) are populated with data fetched primarily from the cr\_alpha\_content table based on the selected list ID.

**Authentication & Authorization (Updated)**

* **Authentication:** User identity is verified through a username and password mechanism against the CR\_APP\_USERS Snowflake table. Secure password hashing is implemented using bcrypt.
  + **Flow:**
    1. Unauthenticated users are directed to /login.
    2. User submits their username via login.html.
    3. login.js sends the username to /auth/check-user-status.
    4. The backend authController queries dbAuth.getUserByUsername (which checks CR\_APP\_USERS).
    5. Based on the user's existence, IS\_ACTIVE status, and whether PASSWORD\_HASH is NULL, the backend responds with a status (NEEDS\_SETUP, NEEDS\_LOGIN, INVALID, INACTIVE).
    6. login.js displays the appropriate next step: either the password setup form (if NEEDS\_SETUP), the password entry form (if NEEDS\_LOGIN), or an error message.
    7. **Password Setup:** User submits the new password form. login.js sends username and password to /auth/setup-password. The backend authController re-validates user status, hashes the password using bcrypt.hash, calls dbAuth.updateUserPasswordHash to store the hash in CR\_APP\_USERS, and upon success, creates the user session by setting req.session.username.
    8. **Password Login:** User submits the login form. login.js sends username and password to /auth/login. The backend authController retrieves the stored hash via dbAuth.getUserByUsername, compares the submitted password against the hash using bcrypt.compare, and upon successful comparison, creates the user session by setting req.session.username.
    9. Upon successful session creation (from either setup or login), the backend responds with success, and login.js redirects the browser to the home page (/).
  + **Session Management:** express-session middleware handles session state. A cryptographically signed session ID is stored in a browser cookie (httpOnly, secure in production). The server uses this ID to look up session data (containing the verified username) on subsequent requests. Session timeout is configured (e.g., 8 hours).
* **Authorization:** Access control for application pages and API endpoints is enforced by the requireVerifiedUser middleware in server.js.
  + This middleware is applied to routes like /, /list-tools, /run-powershell, /generate-content, etc.
  + It checks if req.session.username has been set (which only happens after successful login or setup).
  + If the session or username is missing:
    1. For browser requests expecting HTML content (e.g., accessing /list-tools directly), the user is redirected to the /login page.
    2. For API requests (e.g., POST to /run-powershell), a 401 Unauthorized status code and JSON error message are returned.

**Metadata Logging Summary**

Metadata and operational logs are captured across various components and stored in dedicated Snowflake tables:

* cr\_user\_requests: Logs list generation/scrubbing request metadata. Includes the **verified session username**, the user's prompt, the final generated/executed SQL, and the output filename. Written by the PowerShell scripts (list\_generation\_power\_shell.ps1, list\_filter\_power\_shell.ps1).
* alpha\_beta\_list\_generation\_results: Stores the actual client/context data rows resulting from list generation/scrubbing queries run by PowerShell. Includes RUN\_BY (verified session username) and RUN\_AT timestamp for each row. Written by the PowerShell scripts.
* cr\_alpha\_beta\_content\_creation: Logs events related to DOCX marketing content generation. Includes the **verified session username (formatted as** [**user@domain.com**](https://www.google.com/url?sa=E&q=mailto%3Auser%40domain.com) **by Node.js helper)**, timestamp, type of content generated (e.g., 'Alpha Opt-In'), and the List ID used as input. Written by the Node.js server (/generate-content endpoint).
* cr\_opt\_in\_out\_surveys: Logs details of Qualtrics survey creation attempts. Includes feature details, survey type, client-facing name, the generated Qualtrics Survey ID and URL, the **requesting session username**, the status of the survey sharing attempt, and a timestamp. Written by the Python script (create\_qualtrics\_survey.py).
* CR\_APP\_USERS: Implicitly logs user activity via CREATED\_TS (when user added) and optionally LAST\_LOGIN\_TS (updated on successful login/setup).

**Main Flows (Updated with Authentication)**

1. **Authentication Flow:**
   * User attempts to access / or a tool page (e.g., /list-tools).
   * If no valid session exists, requireVerifiedUser middleware (for /) or the page's session check script redirects to /login.
   * login.html loads. User enters username, submits. login.js calls POST /auth/check-user-status.
   * Backend checks CR\_APP\_USERS, responds with status (e.g., NEEDS\_SETUP).
   * login.js displays the password setup form. User enters password, confirms, submits. login.js calls POST /auth/setup-password.
   * Backend validates, hashes password, updates CR\_APP\_USERS, creates session (req.session.username = 'verified\_user'), responds success.
   * login.js receives success, redirects to /.
   * home.html loads, /auth/check-session confirms session, tools are displayed. (Similar flow for NEEDS\_LOGIN status, using /auth/login endpoint).
2. **List Generation Flow:**
   * User (authenticated, session active) navigates to /list-tools. Access granted by requireVerifiedUser / page script passes session check.
   * User interacts with UI, builds prompt, specifies filename details, clicks "Generate List".
   * Frontend JS sends POST request to /run-powershell with prompt (var2) and filename (var3).
   * Node.js requireVerifiedUser middleware verifies req.session.username exists.
   * /run-powershell handler retrieves the verified username from req.session.
   * Node.js uses spawn to execute list\_generation\_power\_shell.ps1, passing the **session username as $var1**, prompt as $var2, filename as $var3.
   * PowerShell script uses $var1 (session username) when connecting to Snowflake SPs and logging to alpha\_beta\_list\_generation\_results and cr\_user\_requests. It executes the SP, processes SQL, queries data, saves Excel to OneDrive.
   * Node.js captures PowerShell output and sends the result/status message back to the frontend JS.
   * Frontend JS displays the message.
3. **List Scrubbing/Filtering Flow:**
   * User (authenticated) is on /list-tools. Access granted/session confirmed.
   * User opens "Scrub Existing List" modal, enters details (filename parts, context IDs, filters, save location preference). Clicks "Submit Scrub Request".
   * Frontend JS constructs the prompt (var2), filename (var3), save flag, sends POST to /run-list-filter.
   * Node.js requireVerifiedUser middleware verifies session.
   * /run-list-filter handler retrieves verified username from req.session.
   * Node.js uses spawn to execute list\_filter\_power\_shell.ps1, passing **session username as $var1**, prompt as $var2, filename as $var3, and -SaveToDownloads switch if applicable.
   * PowerShell script uses $var1 (session username) for SP calls and logging. It executes the filter SP, processes SQL, queries data, saves Excel (OneDrive or Downloads), logs results and request.
   * Node.js captures PowerShell output, sends result/status message back to frontend.
   * Frontend JS displays the message.
4. **Marketing Content Generation Flow:**
   * User (authenticated) navigates to /marketing-content. Access granted/session confirmed.
   * Frontend JS calls /get-lists on load.
   * Node.js requireVerifiedUser middleware verifies session for /get-lists. /get-lists handler retrieves req.session.username, queries DB (getAllListMetadata formats username) for lists associated with that user.
   * Node.js responds with the list data. Frontend JS populates the "Your Generated Lists" table.
   * User selects a list (from their list or search results), clicks "Generate Doc", and selects a template in the modal.
   * Frontend JS sends POST to /generate-content with listId and templateName.
   * Node.js requireVerifiedUser middleware verifies session.
   * /generate-content handler retrieves req.session.username. It parses the listId to get feature details, calls getFeatureContentData to query cr\_alpha\_content, uses docxtemplater to populate the selected DOCX template.
   * Node.js saves the generated DOCX file to the configured OneDrive path. It then calls logContentCreation, passing the **session username** (which the helper formats to [user@domain.com](https://www.google.com/url?sa=E&q=mailto%3Auser%40domain.com) for DB consistency) and other details.
   * Node.js responds to the frontend with the path of the saved file or an error.
   * Frontend JS displays the confirmation message with the path.
5. **Qualtrics Survey Creation Flow:**
   * User (authenticated) navigates to /qualtrics-survey. Access granted/session confirmed.
   * Frontend JS calls /get-lists on load. Lists are fetched and displayed as in the Marketing Content flow.
   * Option 1 (List): User clicks a "Create Opt-In/Opt-Out" button on a list row. JS extracts feature details, potentially calls /get-feature-name if needed.
   * Option 2 (Manual): User fills the manual form, JS potentially calls /get-feature-name based on Feature Number input.
   * User submits the request (via button click or form submit). Frontend JS sends POST to /create-qualtrics-survey with survey type, feature number, stage, and client-facing name.
   * Node.js requireVerifiedUser middleware verifies session.
   * /create-qualtrics-survey handler retrieves req.session.username.
   * Node.js uses spawn to execute create\_qualtrics\_survey.py, passing the survey details along with the **session username formatted without the domain** (as expected by the Python script for Qualtrics user lookup).
   * Python script connects to Qualtrics API, creates the survey, connects to Snowflake, looks up the Qualtrics User ID for the provided username, attempts sharing, logs the outcome to cr\_opt\_in\_out\_surveys, and prints a JSON result (including sharingStatus) to stdout.
   * Node.js captures the JSON output from Python and sends it back to the frontend JS.
   * Frontend JS displays the success message (with survey link) and any sharing status warnings.

**Background Processes**

* **Python Job (Opt\_Ins\_and\_Opt\_Outs.py):** Runs periodically. Reads raw data files (e.g., opt-in/out lists) from OneDrive, cleans them, and uploads to Snowflake staging tables. (Functionality assumed unchanged).
* **Snowflake Tasks:** Run daily.
  + ADD\_NEW\_APP\_USERS\_TASK (New): Inserts new eligible users from EMERALD\_CITY\_WFA\_DATA into CR\_APP\_USERS.
  + ALPHA\_BETA\_LIST\_GENERATION\_TASK: Refreshes the alpha\_beta\_list\_generation table (source data for list tools).
  + CR\_FULL\_INVITE\_LIST\_TASK: Reloads the CR\_FULL\_INVITE\_LIST table (historical invites for exclusions).

**Reporting Hierarchy & Data Usage Guide**

*\*\*use: corpanalytics\_business\_prod.scratchpad\_prdpf for all below tables*

**1. User Authentication & Activity**

* **Table:** **CR\_APP\_USERS**
* **Purpose:** Tracks authorized application users, their status, and basic login activity.
* **Key Columns for Reporting:**
  + USERNAME: The unique identifier of the user (e.g., cmollica).
  + IS\_ACTIVE: Flag indicating if the account is enabled (TRUE) or disabled (FALSE).
  + LAST\_LOGIN\_TS: Timestamp of the user's most recent successful login or initial password setup.
  + CREATED\_TS: Timestamp when the user record was initially added.

**2. List Generation & Scrubbing Tool Usage**

* **Table:** **cr\_user\_requests**
* **Purpose:** Logs each request submitted via the List Generation or List Scrubbing tool (invokes PowerShell scripts).
* **Key Columns for Reporting:**
  + "USER": The username (session user, typically user@domain.com format) who initiated the request.
  + PROMPT: The full natural language prompt or filter criteria submitted.
  + GENERATED\_SQL: The final SQL query executed by the PowerShell script.
  + FILE\_NAME: The base name of the output Excel file (e.g., FEATURE-123\_Beta\_1). May include annotations like (Saved to Downloads).
  + INSERT\_TIMESTAMP: Timestamp when the request was logged.
* **Table: alpha\_beta\_list\_generation\_results**
* **Purpose**: Stores the actual data rows (client/context details) included in the output of successful list generation/scrubbing operations.
* **Key Columns for Reporting**:
  + RUN\_BY: The username (session user, typically user@domain.com format) who generated this specific data row.
  + RUN\_AT: Timestamp when this specific data row was processed/logged by PowerShell.
  + (Other columns): Includes all client/context details like CONTEXTID, CLIENT\_NAME, CSM\_TIER, SPECIALTY, etc., representing the content of the generated list.

**3. Marketing Content Generation Tool Usage**

* **Table:** **cr\_alpha\_beta\_content\_creation**
* **Purpose:** Logs each successful generation of a DOCX marketing invite document.
* **Key Columns for Reporting:**
  + "USER": The username (session user, formatted as user@domain.com) who generated the document.
  + "TIMESTAMP": Timestamp of the generation event.
  + "CONTENT\_TYPE": The type of template selected (e.g., Alpha Opt-In, Beta Opt-Out).
  + "LIST\_NAME": The identifier (filename) of the list used as input.

**4. Qualtrics Survey Creation Tool Usage**

* **Table:** **cr\_opt\_in\_out\_surveys**
* **Purpose:** Logs attempts to create Qualtrics surveys via the application.
* **Key Columns for Reporting:**
  + REQUESTING\_USER: The username (session user, base format like cmollica) who initiated the request.
  + FEATURE\_NUM: The feature key associated with the survey (e.g., FEATURE-123).
  + ALPHA\_BETA: The stage associated with the survey (e.g., Alpha, Beta).
  + CLIENT\_FACING\_NAME: The feature name used within the survey content.
  + SURVEY\_TYPE: The type of survey created ('Opt-In' or 'Opt-Out').
  + QUALTRICS\_SURVEY\_ID: The unique ID assigned by Qualtrics upon successful creation.
  + QUALTRICS\_SURVEY\_URL: The direct URL link to the created survey in Qualtrics.
  + SHARING\_STATUS: Outcome of the automatic survey sharing attempt (success, failed, user\_not\_found, etc.).
  + INSERT\_TIMESTAMP: Timestamp when the log entry was created.

**Summary Hierarchy for Usage Reporting:**

1. **User Base & Login Activity:** **CR\_APP\_USERS** (Who can use it, when they last logged in)
2. **List Tool Activity (Requests):** **cr\_user\_requests** (Who requested what list/scrub, when, with what prompt)
3. **List Tool Activity (Data Output):** **alpha\_beta\_list\_generation\_results** (What specific clients/contexts were in the output lists generated by whom and when)
4. **Marketing Tool Activity:** **cr\_alpha\_beta\_content\_creation** (Who generated which document type for which list, when)
5. **Qualtrics Tool Activity:** **cr\_opt\_in\_out\_surveys** (Who created which survey for which feature, when, Qualtrics links, sharing status)

**Configuration & Dependencies**

* **Node.js:**
  + Runtime required.
  + Dependencies: express, body-parser, express-session, odbc, pizzip, docxtemplater, cheerio, **bcrypt**.
  + Environment Variables: API\_USERNAME, API\_PASSWORD (Snowflake), SESSION\_SECRET (Critical for session security).
  + File Structure: Includes new files authRoutes.js, authController.js, dbAuth.js, dbUtils.js, public/login.html, public/login.js.
  + Hardcoded Paths: Paths to scripts, templates, output dirs in server.js need careful management.
* **PowerShell:**
  + Runtime required.
  + Dependency: ImportExcel module.
  + Environment Variables: Reads API\_USERNAME, API\_PASSWORD from Machine scope.
* **Python:**
  + Runtime (Python 3) required.
  + Dependencies: requests, snowflake-connector-python.
  + Environment Variables: QUALTRICS\_API\_TOKEN, QUALTRICS\_DATACENTER\_ID, API\_USERNAME, API\_PASSWORD.
  + Hardcoded Paths: Base QSF template path in create\_qualtrics\_survey.py.
* **Snowflake:**
  + Configured account, database (CORPANALYTICS\_BUSINESS\_PROD), schema (SCRATCHPAD\_PRDPF), warehouse, role (CORPANALYTICS\_BDB\_PRDPF\_PROD\_RW), user (API\_USERNAME).
  + New table CR\_APP\_USERS and task ADD\_NEW\_APP\_USERS\_TASK required.
  + ODBC Driver: SnowflakeDSIIDriver installed on machines running Node.js and PowerShell scripts.

**Summary**

The Product Operations Alpha/Beta Command Center now employs a robust username/password authentication system managed via Node.js, bcrypt hashing, and a dedicated Snowflake table (CR\_APP\_USERS). Approved users set their own passwords on first use. Server-side sessions (express-session) maintain the authenticated state. A new dedicated login page (/login) handles the auth flow. Access to application pages and APIs is protected by middleware (requireVerifiedUser), redirecting unauthenticated users appropriately. Core tool functionalities securely utilize the verified username from the user's session for executing tasks and logging metadata. Background processes continue to support data freshness and user list management. Secure configuration, particularly of the SESSION\_SECRET environment variable, is essential.