Student Portal Documentation

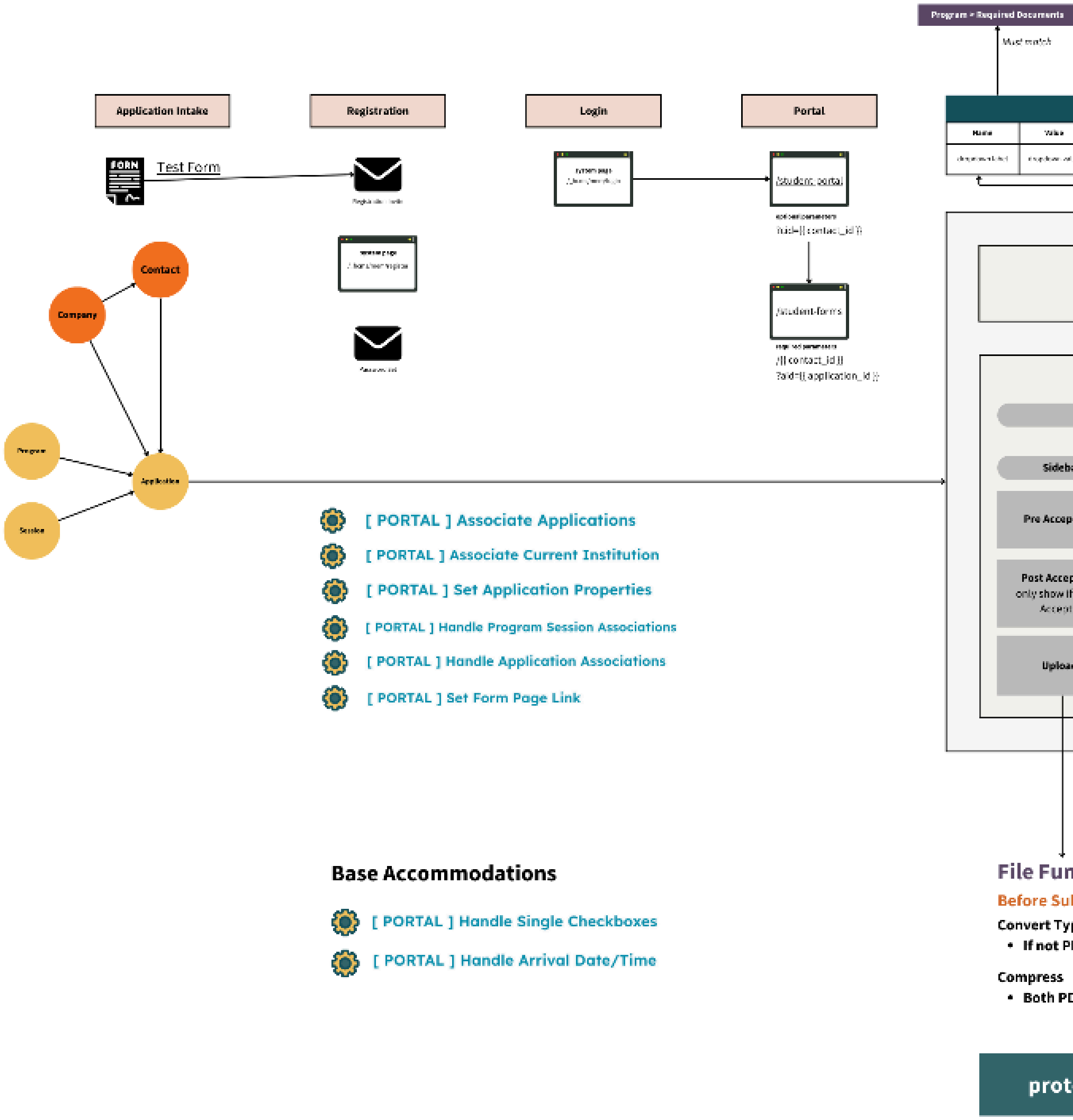
Project Big Picture:

- Move existing website to HubSpot CMS
- Build and configure Portal for business specific needs (applications, sessions, etc.)
- Migrate from Sales Force CRM to HubSpot

Overview

See Canva

The Student Portal is designed to facilitate the application and form submission process for students who wish to participate in study abroad programs. The portal offers functionality for managing personal and contact information, application properties, required forms, payment processing, and file uploads. It leverages HubSpot's Custom Objects, HubDB tables, serverless functions, and GraphQL queries to create a seamless experience for students.


Program + Required Documents
Must match
Name
Value
Application Intake
Registration
Login
Portal
FORM
Test Form
/student-portal
?uid={{ contact_id }}
/student-forms
/{{ contact_id }}
?aid={{ application_id }}
Contact
Company
Program
Application
Session
[PORTAL] Associate Applications
[PORTAL] Associate Current Institution
[PORTAL] Set Application Properties
[PORTAL] Handle Program Session Associations
[PORTAL] Handle Application Associations
[PORTAL] Set Form Page Link
Sideba
Pre Accep
Post Accep
only show if
Accept
Uploa
Base Accommodations
[PORTAL] Handle Single Checkboxes
[PORTAL] Handle Arrival Date/Time
File Fun
Before Sul
Convert Ty
If not P
Compress
Both PD
prot

Overview
Data Structure

Data Structure

Custom Objects

Programs

The **Programs custom object** acts as a foundational data structure within the student application portal, providing a comprehensive and organized way to manage all program-related details, requirements, sessions, and associations with student applications. It ensures that students have a tailored experience based on the specific program they are applying to or have enrolled in, supports the dynamic display of relevant forms and data, and aids administrators in maintaining accurate and up-to-date program information throughout the application process.

Sessions

The **Sessions custom object** serves as a detailed repository of all the various sessions associated with study abroad or internship programs. It captures and manages session-specific details such as start/end dates, session requirements, and application deadlines, ensuring that applicants are guided through the process based on the exact session they have selected. By linking applications to specific sessions, this object provides the flexibility to handle multiple offerings within a single program, allowing the portal to display tailored information and requirements for each applicant.

In essence, the Sessions custom object is crucial for ensuring that each program can offer multiple timeframes or cohorts, providing students with options for when they can participate. It works in tandem with the Programs custom object to create a comprehensive and flexible structure for managing all aspects of program offerings, from application intake to final session details. This object enables a more dynamic, organized, and session-specific application process, ensuring that both students and administrators have accurate and relevant data throughout the application journey.

Applications

The **Applications custom object** serves as the central data entity that manages every aspect of a student's application process for study abroad or internship programs. It captures individual application details, tracks progress through various stages, and ensures that all relevant information is presented based on the associated program and session. By linking with both the Programs and Sessions custom objects, it provides a comprehensive and contextually accurate view of each applicant's journey.

This object is essential for providing a personalized, organized, and data-driven application experience. It facilitates transparency, efficient management, and automation throughout the application process, ensuring that both students and administrators have the tools they need to navigate and manage applications effectively. The Applications custom object is the heart of the system, tying together all elements of the program offerings to create a seamless and user-friendly application experience.

HubDB Tables

The portal utilizes HubDB tables to manage data related to students, programs, and their associated requirements:

Document Types

Table: https://app.hubspot.com/hubdb/21167250/table/7547819

This table stores the general names of all required form types. This table is synchronized with the **Required Forms**, **Submitted Forms** & **Pending Forms** properties.

Protected Properties

Table: https://app.hubspot.com/hubdb/21167250/table/7582635

This table contains data about how upload properties should be handled including whether they are required and how and where the files should be saved.

Student Portal - Forms

Table: https://app.hubspot.com/hubdb/21167250/table/7548486

This table contains the form name which should match a **Document Type** and the **form ID** to show for that type.

Assets

Template

Student-portal.html

Page: https://www.barcelonasae.com/student-portal (Template)

GraphQL: https://app.hubspot.com/design-manager/21167250/file/171286129155

Document-details.html

Page: https://www.barcelonasae.com/student-forms/ (Template)

GraphQL: https://app.hubspot.com/design-manager/21167250/file/142982282757

Modules

Applications Listing

Module: https://app.hubspot.com/design-manager/21167250/modules/142989135519

Relies on data from the template's GraphQL file.

The `application-listing` module displays a list of all applications associated with the logged-in contact in a CRM. It fetches application data, processes status information, and renders application cards within a styled grid layout. This module provides an organized and user-friendly way for students or applicants to track the status of their applications for various programs or internships.

Data Retrieval

- The module starts by importing several utility functions (`slugify`, `icons`, `build_button`, `render_card`, etc.) from an external macro file.
- It identifies the current contact using `Query.cid` or falls back to `request.contact.contact_vid`.
- It gathers all associations related to the contact by combining data from two different association collections: `p_applications_collection__applicant.items` and `p_applications_collection__application_to_contact.items`. This creates a unified list of all applications associated with the contact.

Processing Application Data

- The module extracts the `hs_object_id` from each item in the association list, creating a list of all application IDs (`all_application_ids`).
- It defines the application properties that need to be retrieved, such as `hs_object_id`, `applicant`, `internship_program__c`, `application_name_formula__c`, `advisor_approval_status__c`, `application_completion_date__c`, `hs_pipeline_stage`, and several others.
- Using the `crm_objects` function, the module fetches all application details based on `all_application_ids` and the defined properties.

Rendering the User Interface

- The module generates a container `<div>` that includes a personalized greeting message for the logged-in contact (`Welcome back, [Name]`) followed by a title "My Applications."
- It loops through each `application_id` and matches it with the fetched application details. For each application:

    - It extracts details such as `session__c` and program names.
    - It checks if the application has a valid `associated_program_name` or `program_name`.
    - Based on the status value of the application, it maps the status using definitions stored in `module.definitions`.
    - The module then renders an application card using `render_card`, passing necessary data such as the application, session name, contact information, and the status value.

Application Properties

Module: https://app.hubspot.com/design-manager/21167250/modules/147694622871

The `application-properties` module serves as a dynamic interface for viewing and managing application-related data. It retrieves properties associated with the current application from a CRM, renders them in an organized and interactive layout, and allows users to edit and save changes. The module offers an expandable view for different sections, making it easy to navigate and update application data, thus providing a powerful tool for administrators or students to manage their application details efficiently.

Data Retrieval and Processing

- **Initialization**:
    - The module begins by setting up `module_name` as `application-properties` and importing various utility functions and configuration data from external macro files.
- **Properties and Data Definitions**:
    - It initializes two main dictionaries, `all_props_to_get` and `all_data`, to store properties and values related to the `contact` and `p_applications` objects.
    - It also sets up `associated_ids` to fetch the `contact` ID from the URL path (`request.path`) and the `application` ID (`Query.aid`) to identify the specific application for which data is retrieved.
- **Looping Through Sections**:
    - For each section defined in `module.sections`, the module extracts the following properties: `contact_properties`, `doc_properties`, `program_properties`, and `application_properties`.
    - The `application_properties` are collected and appended to `all_props_to_get`, grouping properties based on their `object_type`.
- **Fetching Data from CRM**:

    - For each object type (`contact` and `p_applications`), the module constructs a comma-separated string of property names and fetches data using `crm_objects`. The results are then stored in `all_data` for later use.

Rendering Properties

- The `render_prop` macro is responsible for generating the HTML structure for each property:
    - It retrieves the property name, type, label, and value from `all_data`.
    - If the property is of type `date`, it formats the value accordingly.
    - The resulting HTML includes:
        - A `label` for the property
        - An `input` field pre-filled with the property's value (disabled by default)
        - A `select` dropdown if the property is of type `enumeration`
        - An "edit" and "save" icon to allow interaction with the property.

User Interface Structure

- The module generates the HTML structure for the interface:
    - Each section in `module.sections` is rendered with its title and properties using a `<div>` container.
    - The properties within a section are rendered using the `render_prop` macro.
    - The module produces an expandable/collapsible interface where each section's properties are initially hidden and can be revealed by clicking the section's title.

JavaScript Functionality

- The module uses jQuery to add interactivity
- **Click Handling**:
    - Clicking on the section title (`.section__title`) toggles the visibility of properties within that section.
    - Clicking the "edit" icon enables the input field for editing.
    - Clicking the "save" icon triggers saving the modified property value.
- **Data Submission**:
    - The `build_this_payload` function constructs the payload required to update the application properties.
    - The `submit_this_to_serverless` function sends the updated data to a serverless API endpoint (`https://www.barcelonasae.com/_hcms/api/handleActions`) using a POST request.

    - When a `select` dropdown value is changed, it updates the corresponding `input` field to reflect the new selection.

Required Form Flow

Module: https://app.hubspot.com/design-manager/21167250/modules/150812618517

The **"form-flow"** module serves as a comprehensive tool for managing the multi-step application process for users. It integrates data retrieval, user interface rendering, form handling, and backend updates to provide a seamless experience.

Key functionalities include:

- **Dynamic Form Determination:** The module determines which forms are required for the user based on their application status and associated programs.
- **Data Integration and Pre-Filling:** It retrieves existing data from the CRM to pre-fill forms, reducing user effort and ensuring data consistency.
- **Progress Tracking and Updates:** The module tracks the user's progress through the forms and updates the CRM upon form submissions, including advancing the application stage when appropriate.
- **User Interface Enhancements:** Provides a user-friendly interface with navigation, form statuses, completion messages, and error handling through popups.
- **Conditional Rendering and Logic:** Utilizes conditional logic to handle various scenarios, such as whether only post-acceptance forms are required, handling different application stages, and adapting to different versions or debug modes.
- **Asynchronous Data Handling:** Employs asynchronous JavaScript functions to fetch and process data without blocking the user interface, enhancing performance and responsiveness.
- **Validation and Error Handling:** Includes client-side validation, such as checking passport expiration dates and handling date/time inputs, to prevent invalid data submission and guide the user appropriately.

By orchestrating these functionalities, the module ensures that users can complete their application process efficiently, while the organization maintains accurate and up-to-date records within the CRM system. The module's design emphasizes scalability and adaptability, allowing it to accommodate various program requirements and user scenarios.

Module Initialization and Macros Import

- **Module Naming:** The module is initialized with the name "form-flow" to identify its purpose within the application.
- **Macros Importing:** A set of utility macros are imported from a shared macros file. These macros include functions for string manipulation (`slugify`), icon rendering (`icons`), UI

components (`build_button`, `render_card`), and CSS styling helpers (`transition`, `transform`, `borderRadius`, `boxShadow`, `textShadow`). These macros facilitate code reuse and maintainability.

Icons Macro Definition

- **Icon Rendering Macro:** A custom macro named `icons` is defined to render SVG icons based on the provided icon name. It includes definitions for several icons such as "world", "angle-right", "circle-check", and "spinner". This macro allows for consistent icon usage throughout the module without repeating SVG code.

Variable Initialization

- **Tracker Property Name:** The module retrieves the name of the property used to track application stages (`tracker_prop`) from the module's configuration. This property is essential for determining the user's progress in the application process.
- **Properties to Retrieve:** The module defines lists of properties to fetch from the database or CRM for documents (`document_props_to_get`), applications (`application_props_to_get`), and programs (`program_props_to_get`). These properties include identifiers, names, types, and other relevant data.
- **Application ID and Query Parameters:** The application ID is retrieved from the URL query parameters (`application_id`), which is used to identify the specific application the user is working on.

Data Retrieval from Databases and CRM

- **Fetching Data from HubDB:** The module fetches all documents and forms from HubDB tables using their respective table IDs. This data includes information about the forms and documents that may be required in the application process.
- **Fetching CRM Data:** The module retrieves the user's applications and contact information from the CRM. It accesses the contact's applications (`contact_applications`), the contact's details (`associated_contact`), the contact's email (`associated_contact_email`), and the forms the contact has submitted (`associated_contact_forms_submitted`).

Processing Submitted Forms and Documents

- **Tracking Submitted Documents:** The module initializes an empty list to store the names of documents that the user has submitted (`associated_contact_docs_submitted`). It then iterates over the submitted forms associated with the application to populate this list.

- **Initializing Data Arrays:** Several arrays are initialized to store data related to documents (`these_documents`), document types (`associated_document_types`), program IDs (`associated_program_ids`), application IDs (`associated_application_ids`), required document types (`required_document_types`), contact data (`all_contact_data`), and application data (`all_application_data`).

Collecting Contact and Application Data

- **Collecting Contact Data:** The module iterates over the contact's properties, excluding any associations, and collects all relevant data into a list of dictionaries. Each dictionary contains a property name and its corresponding value. This data is used to pre-fill form fields and for processing logic.
- **Collecting Application Data:** For each application associated with the contact, the module:
    - Retrieves the associated program ID and adds it to the list of program IDs.
    - Retrieves the submitted documents for the application and processes their types, adding them to the list of associated document types.
    - Updates the selected session information based on the application's session name.
    - Collects all properties of the application, excluding associations, into the application data list.

Determining Required and Pending Documents

- **Identifying Required Documents:**
    - The module checks each associated program to determine the required documents. It also checks if the program is configured to require only post-acceptance forms (`post_acceptance_only`).
    - It appends the labels of the required documents to the `required_document_types` list.
- **Filtering Required Documents:**
    - Based on whether a deposit is required, the module selects the appropriate required documents from all documents. If the deposit is required, all documents matching the required types are selected; otherwise, any documents containing "500" in their name are excluded.
- **Determining Submitted and Remaining Documents:**
    - The module identifies which required documents have already been submitted by the user by comparing the associated document types with the required document types.
    - It calculates the remaining documents that the user needs to submit by rejecting the submitted document names from the list of required documents.
- **Tracking Pending Forms:**
    - The names of the remaining documents are added to the `pending_forms` list, which represents the forms that the user still needs to complete.

Assessing Completion Status

- **Checking Pre-Acceptance Completion:**
    - The module selects all pre-acceptance documents from the required documents.
    - It iterates over these documents, checking if each one has been submitted by the user.
    - An array (`is_completed_array`) tracks whether each document is complete.
    - The module determines whether all pre-acceptance forms are completed by verifying that there are no "false" values in the `is_completed_array`.
- **Determining Acceptance Status:**
    - The module checks the application stage based on the tracker property value. If the application stage is not in the pre-acceptance stages, the user is considered accepted (`is_accepted`).

Frontend Rendering of the Form Flow Interface

- **Main Container and Header:**
    - The module sets up the main container for the form flow, including a header that displays the user's programs and application stages.
    - The header shows a list of the user's programs with links to program pages and a nested list of application stages, indicating the user's progress.
- **Sidebar Navigation:**
    - A navigation sidebar displays the required student forms, categorized by stages such as Pre-Acceptance and Post-Acceptance.
    - The module determines which sections and stages to display based on the user's acceptance status and whether only post-acceptance forms are required.
    - For each stage, it lists the required documents, indicating which ones have been submitted or are pending.
- **Main Content Area with Forms:**
    - The module renders each required form in the main content area using the form tag.
    - It includes logic to:
        - Show or hide forms based on their submission status and whether the user has editing access.
        - Display appropriate messages if the form is already submitted or if the user does not have edit access.
        - Handle special cases like the confirmation deposit form, where submission may be tied to payment status.
- **Completion Messages:**
    - The module displays a message when all forms are completed.
    - If a payment is required, it shows a message indicating that payment is awaiting or has been processed, based on the payment status.

Error Messages and Popups

- The module creates popup modals for various error messages using the lity library. These include:
    - **Passport Error:** Shown when there is an issue with the user's passport information.
    - **Submit Message:** Displayed upon successful submission of a form.
    - **Error Message:** A general error message for unexpected issues.
    - **File Too Large Error:** Shown when the user attempts to upload a file that exceeds the allowed size.
    - **Missing Required Fields:** Displayed when required fields are not completed in a form.
- Each popup has a unique ID and contains the corresponding message from the module's configuration.

Including Styles and Scripts

- **CSS Styles:**
    - The module includes custom CSS styles specific to the form-flow module. These styles define the layout and appearance of the module's components, such as positioning, z-index, maximum width, padding, margins, and font styles.
    - Styles are applied to elements like the main container, background overlays, headers, navigation menus, and form components.
- **JavaScript Includes:**
    - The module includes several JavaScript files necessary for its functionality:
        - **lity.js:** A library for handling lightbox popups.
        - **protectData.js/protectData-v2.js:** Custom scripts for data protection, included conditionally based on query parameters.
        - **Date and Time Picker Scripts:** Libraries for enhancing date and time input fields.
    - Conditional logic is used to include different scripts or styles based on query parameters, such as debug mode or versioning.

JavaScript Functionality

- **Initialization Functions:**
    - **Event Handlers for Navigation:**
        - The module initializes click events on navigation triggers, allowing users to switch between different forms by clicking on the navigation items.
    - **Populating Hidden Fields:**
        - It populates hidden form fields with user data such as email and application ID to ensure that submissions are correctly associated with the user's account.

    - **Session Options Initialization:**
        - The module initializes session options for select inputs based on available sessions for the program.
        - If only one session is available, it automatically selects it and, if necessary, submits the form.
- **Data Retrieval and Pre-Filling Forms:**
    - **Asynchronous Data Fetching:**
        - The module defines functions to asynchronously fetch application data from the server. It constructs payloads with the necessary properties to retrieve.
        - Data is fetched from a serverless function endpoint and processed upon receipt.
    - **Pre-Filling Form Fields:**
        - Once the data is retrieved, the module pre-fills form fields with existing data from the CRM.
        - It handles various input types, including text inputs, textareas, selects, and checkboxes.
        - For completed forms, it populates the fields and may disable editing to prevent changes to submitted data.
- **Form Submission Handling:**
    - **Event Listener for Form Submission:**
        - The module listens for a custom "force-submit" event on forms, which is triggered upon form submission.
    - **Updating Backend CRM:**
        - Upon submission, it updates the CRM with new data, such as marking forms as submitted and updating the list of pending forms.
        - If the last pre-acceptance form is submitted, it updates the application stage accordingly.
    - **Building Payloads for Server Requests:**
        - The module constructs payloads containing object types, IDs, and properties to be sent to serverless functions for processing.
- **Utility Functions:**
    - **findByPropValue:**
        - A helper function to find an object within an array by matching a specific property value.
    - **waitForVariable and checkVariable:**
        - Functions to wait asynchronously until a certain variable is ready before proceeding with data initialization.
    - **combineObjects:**
        - Merges two objects into one, used when combining data from multiple server responses.
- **Handling Dynamic Form Elements:**

    - **Date and Time Pickers:**
        - Initializes date and time pickers on relevant input fields to enhance user input and ensure correct data formats.
    - **Conditional Form Logic:**
        - Handles dynamic form logic, such as showing or hiding dependent fields based on user selections.
        - For example, if a user indicates they are using financial aid, additional fields are displayed.
- **Error Handling and User Feedback:**
    - **Passport Expiration Handling:**
        - Checks the passport expiration date input by the user. If the date is before a minimum required date, it displays an error popup.
    - **Handling Arrival Date and Time:**
        - Combines separate date and time inputs into a single datetime value for processing.
    - **Preventing Multiple Submissions:**
        - Implements logic to prevent the form submission handler from executing multiple times accidentally.

Conditional Logic and Handling Different Scenarios

- **Handling Query Parameters:**
    - The module includes logic to handle different query parameters, such as "debug" or "v2", to adjust its behavior accordingly.
    - In debug mode, it may display additional information to assist with troubleshooting.
    - The version parameter determines which scripts are included, allowing for version control and testing of new features.
- **Handling Application Stages:**
    - If all pre-acceptance forms are completed and the application is still in the pre-acceptance stage, the module automatically updates the application stage to move the user forward in the process.
- **Exception Handling:**
    - The module includes flags and checks to handle exceptions, such as when no required documents are associated with a program.
    - It ensures that the user interface remains functional and informative even when certain expected data is missing.

JavaScript

protectData.js

This JavaScript file enhances the form submission process by adding custom logic for handling file uploads, form validations, and processing protected data securely. It integrates with external services like ConvertAPI for file conversions and uses serverless functions to handle sensitive data. The script ensures that users receive immediate feedback through modals, improving the user experience. By validating required fields and managing responses effectively, it maintains data integrity and ensures that the backend system receives complete and accurate information. The exclusion of specific forms and dynamic handling based on configurations allows for flexibility and adherence to varying requirements across different forms.

Key Features and Processes:

- **File Handling:**
  - Supports file uploads with automatic conversion and compression.
  - Ensures files meet size and format requirements before submission.
  - Processes sensitive files through secure endpoints.
- **Form Validation:**
  - Validates that all required fields are filled.
  - Provides immediate feedback to users on missing information.
- **Protected Data Management:**
  - Identifies protected properties requiring special handling.
  - Uses configurations from HubDB to determine processing rules.
  - Handles passports and other sensitive documents securely.
- **Asynchronous Operations:**
  - Utilizes asynchronous functions to fetch data and process files without blocking the user interface.
  - Ensures smooth user experience even during long-running operations.
- **Event-Driven Architecture:**
  - Employs event listeners to respond to user actions promptly.
  - Handles form submissions and file input changes dynamically.
- **User Interface Enhancements:**
  - Uses modals to inform users of the progress and outcomes of their actions.
  - Provides clear messaging for errors and successes.
- **Server Communication:**
  - Sends data to serverless functions for processing and storage.
  - Parses and handles responses to guide subsequent actions.
- **Flexibility and Configurability:**
  - Retrieves configurations from HubDB, allowing for easy updates without changing the code.
  - Adapts to different form requirements and properties dynamically.

By integrating these functionalities, the script ensures a robust and secure form submission process, enhancing both the user experience and the reliability of data handling within the application.

Initialization

- **ConvertAPI Initialization:**
    - The script initializes the **ConvertAPI** library using a provided secret key. This library is used for converting and compressing files before they are uploaded to the server.
    - By authenticating with the secret key, the application gains access to the ConvertAPI's services, which include converting images to PDFs, compressing files, and changing file formats.
- **Lity Modal Variable:**
    - A variable is declared to hold an instance of a **Lity** modal. Lity is a lightweight, responsive lightbox library used to display modals and popups.
    - This variable (`activeLity`) is used throughout the script to manage modal dialogs that provide user feedback, such as loaders, error messages, or success notifications.

Utility Functions

These functions serve as helpers to perform common tasks throughout the script.

- **`extractObjectVals(array, prop)`**:
    - **Purpose:** Extracts unique values of a specified property from an array of objects.
    - **Usage:** When fetching data from the HubDB table, this function is used to get a list of unique protected property names that require special handling.
    - **Process:**
        - Iterates over an array of objects.
        - Extracts the value of the specified property from each object.
        - Uses a `Set` to eliminate duplicate values.
        - Returns an array of unique values.
- **`fetchHubDBTableRows()`**:
    - **Purpose:** Fetches rows from a specific HubDB table.
    - **Usage:** Retrieves configuration data that dictates how certain form fields should be handled, especially protected data that requires secure processing.
    - **Process:**
        - Constructs the URL for the HubDB API endpoint, including the table ID and portal ID.
        - Sends a GET request to the endpoint.
        - Parses the JSON response and returns it.

  - **Error Handling:** If the network response is not successful, it throws an error and logs it to the console.
- `sendPayload(url, payload)`:
  - **Purpose:** Sends a POST request with a JSON payload to a specified endpoint.
  - **Usage:** Used to send data to serverless functions or API endpoints for processing form submissions, file uploads, or data handling.
  - **Process:**
    - Sends a POST request to the provided URL with the JSON payload.
    - Includes appropriate headers to indicate the content type.
    - Parses and returns the JSON response.
  - **Error Handling:** If the response is not successful, it throws an error and logs it.
- `handlePassport(passportUrl, applicationId)`:
  - **Purpose:** Handles the processing of passport files by sending them to a specific endpoint designed for passport handling.
  - **Usage:** When a user uploads a passport file, this function ensures that it is sent securely to the backend for further processing, such as extracting data or storing securely.
  - **Process:**
    - Constructs a payload containing the passport file URL and the application ID.
    - Sends the payload to the designated endpoint using `sendPayload`.
    - Returns the response from the server.
- `uploadFiles(prop, fileName, file, applicationId)`:
  - **Purpose:** Uploads files to a specific endpoint for general file handling.
  - **Usage:** Used for uploading files that are not passports but still require secure handling, such as transcripts or other sensitive documents.
  - **Process:**
    - Constructs a payload with the property name, file name, file data (or URL), and application ID.
    - Sends the payload to the upload endpoint using `sendPayload`.
    - Returns the response from the server.
- `getQueryParam(paramName)`:
  - **Purpose:** Retrieves the value of a query parameter from the current URL.
  - **Usage:** Extracts parameters like the application ID (`aid`) from the URL, which are necessary for associating form submissions and file uploads with the correct application.
  - **Process:**
    - Uses the `URLSearchParams` interface to parse the query string.
    - Returns the value of the specified parameter.
- `fileToBase64(file)`:
  - **Purpose:** Converts a `File` object to a Base64-encoded string.

    - **Usage:** Although not directly used in the script, this function can be utilized to convert files to a Base64 string when necessary for data transmission.
    - **Process:**
        - Uses a `FileReader` to read the file as a data URL.
        - Resolves the promise with the result upon successful reading.
        - Rejects the promise if an error occurs.
- **`populateVariable(variable)`:**
    - **Purpose:** Simulates an asynchronous operation, returning the input variable after a delay.
    - **Usage:** Used in testing or when simulating the delay of an asynchronous operation, such as waiting for a server response.
    - **Process:**
        - Returns a promise that resolves to the input variable after a 2-second timeout.

Handling Form Fields

- **`handleFields(e)`:**
    - **Purpose:** This is the core function that handles form field processing during submission. It validates required fields, processes protected data, and manages file uploads.
    - **Usage:** Invoked during form submission to ensure all data is correctly processed before the form is finally submitted to the backend.
    - **Process:**
        - **Form and Inputs Retrieval:**
            - Identifies the form being submitted and retrieves all input fields within it.
            - Extracts the application ID from the URL to associate data correctly.
        - **Fetching Protected Properties:**
            - Calls `fetchHubDBTableRows()` to get data about protected properties that require special handling.
            - Uses `extractObjectVals()` to get a list of these protected properties.
        - **Validating Required Fields:**
            - Iterates over all required fields in the form.
            - Checks if each required field has a value.
            - If a required field is empty, it marks the validation as failed and displays an error message near the field.
        - **Processing Inputs:**
            - If all required fields are filled, the function proceeds to process each input.
            - **For File Inputs:**

    - Retrieves the file from the `selectedFiles` array, which stores files selected by the user.
    - If the property is protected, it handles the file upload securely.
  - **For Protected Properties:**
    - Checks if the property is in the list of protected properties.
    - Retrieves additional configuration from the HubDB data, such as whether the file should be stored in both systems, the file naming convention, and the required file type.
    - Constructs the file name based on the student's name and the property.
    - **Handling Passports:**
      - If the property relates to the passport, calls `handlePassport()` to process it.
    - **Handling Other Files:**
      - For other files, calls `uploadFiles()` to upload them securely.
- **Finalizing Response:**
  - Uses `populateVariable()` to simulate the final response.
  - Returns an object containing the validation status and any response data.

Event Listeners

These listeners handle user interactions with the form and file inputs.

- **File Input Change Event:**
  - **Purpose:** Listens for changes on file input fields to process file uploads immediately after a user selects a file.
  - **Usage:** Ensures that files are converted, compressed, and uploaded as soon as they are selected, enhancing the user experience and reducing the load during form submission.
  - **Process:**
    - **Opening Loader Modal:**
      - Displays a loader modal using Lity to indicate that a process is underway.
    - **Fetching Configuration Data:**
      - Calls `fetchHubDBTableRows()` to get the latest configuration for protected properties.
    - **Retrieving File and Property Details:**
      - Identifies the property associated with the file input.
      - Gets the file object selected by the user.

            - Determines the file type (e.g., image, PDF) and checks the file size.
        - **File Size Validation:**
            - Checks if the file exceeds the maximum allowed size (e.g., 299 MB).
            - If too large, closes the loader modal and displays an error modal.
        - **Processing the File:**
            - Retrieves additional configuration for the property from the HubDB data.
            - Determines the required file type and whether the file needs to be converted.
            - **File Conversion:**
                - If conversion is needed (e.g., converting an image to PDF), uses the ConvertAPI to convert the file.
            - **File Compression:**
                - After conversion (if applicable), compresses the file using the ConvertAPI to reduce its size.
        - **Handling the Result:**
            - If the file processing is successful, adds the file to the `selectedFiles` array.
            - Marks the file input as completed.
            - Closes the loader modal.
            - If there's an error, alerts the user and closes the modal.
- **Form Submission Event:**
    - **Purpose:** Handles the custom processing and submission logic when a user submits a form.
    - **Usage:** Ensures that all form data, including files, are processed correctly and sent to the server in the expected format.
    - **Process:**
        - **Excluding Specific Forms:**
            - Checks if the form being submitted is the deposit form (identified by `depositFormId`). If so, it bypasses custom handling to allow standard form submission.
        - **Preventing Default Submission:**
            - Prevents the default form submission action to implement custom logic.
        - **Opening Loader Modal:**
            - Displays a loader modal to indicate that processing is occurring.
        - **Collecting Form Fields:**
            - Iterates over all inputs in the form to collect field data.
            - **Handling Files:**
                - Sets a flag if any file inputs are present to handle them appropriately.
            - **Preparing Field Data:**

        - Collects all field values into an array, including object type IDs and property names.
        - Handles special cases, such as checkboxes and multiple selections.
    - **Constructing the Submission Payload:**
      - Prepares a payload that includes all fields, context information (like the page URL and title), and legal consent options.
    - **Submitting the Form:**
      - Calls `handleFields()` to process the fields and handle any files or protected data.
      - If all required fields are present, proceeds to submit the form using `handleForm()`.
    - **Handling Form Submission Response:**
      - **Success:**
        - If the form submission is successful and a redirect URI is provided, the user is redirected to that URI.
        - If an inline message is provided, a success modal is displayed, and the page may reload after a delay.
      - **Error:**
        - If there is an error in the response, an error modal is displayed with the error message.
    - **Closing Modals and Cleanup:**
      - Closes the loader modal after processing.
      - If required fields are missing, displays an error modal and closes it after a short delay.

Specific Implementations

- **File Conversion and Compression:**
  - The script uses the ConvertAPI to convert files to the required formats and compress them to reduce file size.
  - It handles different scenarios:
    - **Image to PDF Conversion:** If a user uploads an image when a PDF is required.
    - **Image Compression:** Compresses image files to reduce size without significant loss of quality.
  - Parameters for the conversion and compression are set based on the required file type and naming conventions.
- **Protected Data Handling:**
  - Some form fields are marked as protected and require special handling to ensure data security.

    - The script retrieves configurations from a HubDB table to determine which properties are protected.
    - Protected data is processed through secure endpoints (`handlePassport` and `uploadFiles`) rather than being handled in the standard form submission.
    - This ensures compliance with data protection policies and regulations.
- **Validation of Required Fields:**
    - Before submitting the form, the script checks that all required fields are filled.
    - If any required field is missing, it prevents submission and informs the user by displaying an error message next to the field.
    - This validation enhances data integrity and reduces the likelihood of incomplete submissions.
- **Response Handling:**
    - The script handles various responses from the server after form submission:
        - **Redirects:** If a redirect URL is provided, the user is navigated to that page.
        - **Inline Messages:** If a success message is included, it displays a modal with the message.
        - **Errors:** If an error occurs, it displays an error modal with the details.
- **Excluding Certain Forms:**
    - The deposit form is excluded from the custom handling implemented by the script.
    - This is likely because the deposit form may have specific requirements or uses standard submission methods that should not be altered.
- **User Feedback with Lity Modals:**
    - The script uses Lity modals to provide real-time feedback to the user.
    - Modals are displayed during:
        - **Processing:** A loader modal shows that an operation is in progress.
        - **Errors:** Error modals inform the user of issues that need attention.
        - **Success:** Success messages confirm that actions have been completed.

Serverless Functions

protectData

This JavaScript file serves as a serverless function designed to securely handle the uploading and storage of sensitive files—specifically passport PDFs—to an Amazon S3 bucket. It integrates with the AWS SDK for S3, uses Axios for HTTP requests, and employs cryptographic functions to verify file integrity. The function is intended to be part of a larger application process where applicants submit sensitive documents. By securely uploading these documents to AWS S3 with server-side encryption, the script ensures data protection and compliance with security standards.

Module Imports and Dependencies

- **AWS SDK for S3:**
    - The script imports specific commands (`DeleteObjectsCommand`, `GetObjectCommand`, `ListObjectsCommand`, `PutObjectCommand`) and the `S3Client` from the AWS SDK for JavaScript v3 (`@aws-sdk/client-s3`).
    - These commands facilitate interactions with Amazon S3, such as uploading, retrieving, and managing objects within S3 buckets.
- **File System (`fs`):**
    - Although imported, the `fs` module (Node.js built-in file system module) is not actively used in the provided code.
- **Axios:**
    - A promise-based HTTP client for Node.js and the browser.
    - Used to perform HTTP requests, specifically to download files from provided URLs.
- **Crypto:**
    - Node.js built-in module that provides cryptographic functionalities.
    - Used for generating MD5 hashes to verify file integrity during upload.
- **Other Imports:**
    - **`crypto`:** For hashing and verifying data integrity.
    - **Note:** The code comments indicate some adjustments for compatibility or conversion from different module systems.

Configuration and Environment Setup

- **AWS Configuration:**
    - **Region:** The AWS region is specified (e.g., `'eu-west-1'`).
    - **Credentials:** Access Key ID and Secret Access Key are retrieved securely from environment variables (`process.env.ACCESS_KEY_ID`, `process.env.SECRET_ACCESS_KEY`).
        - **Security Note:** Storing credentials in environment variables is a best practice to avoid hardcoding sensitive information.
- **Environment Variable (`ENV`):**
    - Determines whether the script operates in `'PRODUCTION'` or `'SANDBOX'` mode.
    - Defaults to `'PRODUCTION'` if not specified.
    - **Bucket Selection:**
        - Based on the environment, the script selects the appropriate S3 bucket:
            - **Production Bucket:** For live data.
            - **Sandbox Bucket:** For testing and development purposes.

AWS S3 Client Initialization

- **S3 Client Creation:**

    - An instance of `S3Client` is initialized with the specified AWS region and credentials.
    - This client is used to execute commands that interact with S3, such as uploading and retrieving files.

Main Function

- **Purpose:**
    1. Acts as the entry point for the serverless function.
    2. Handles incoming requests, processes file uploads, and sends responses back to the client.
- **Parameters:**
    1. **`context`:** Contains the request data, including the body with parameters sent by the client.
    2. **`sendResponse`:** A function used to send HTTP responses back to the client.
- **Process Flow:**
    1. **Extracting Request Data:**
        - Retrieves `applicationId` and `file_url` from `context.body`.
        - **`applicationId`:** Unique identifier for the application or user.
        - **`file_url`:** URL pointing to the file (passport PDF) to be uploaded.
    2. **Downloading the File:**
        - Uses Axios to perform a GET request to `file_url`.
        - Sets `responseType` to `'arraybuffer'` to receive the file as binary data.
    3. **Uploading the File to S3:**
        - Calls the `uploadPassport` function with `applicationId` and the downloaded file data.
    4. **Error Handling:**
        - Encloses operations within a `try-catch` block to handle exceptions.
        - Logs errors to the console for debugging.
        - Sends an error response with a 500 status code if an exception occurs.
    5. **Sending Response:**
        - On success, sends a response with a 200 status code and a success message.
        - On failure, sends a response with a 500 status code and an error message.

Helper Functions

uploadPassport(applicationId, base64)

- **Purpose:**
    1. Prepares and uploads the passport file to the specified S3 bucket.

- **Parameters:**
    1. **`applicationId`:** Identifier used to structure the file path within the S3 bucket.
    2. **`base64`:** The binary data of the downloaded file.
- **Process:**
    1. **Defining File Parameters:**
        - Constructs a file object with:
            - **Key:** Combines `applicationId` and `'passport.pdf'` to create a unique file path (e.g., `'12345/passport.pdf'`).
            - **Body:** The binary data (`base64`) of the passport PDF.
            - **Content Type:** Set to `'application/pdf'` to indicate the file type.
    2. **Uploading the File:**
        - Calls the `uploadFile` function with the bucket name and the file object.

uploadFile(bucket, file)

- **Purpose:**
    1. Handles the upload of a file to the specified S3 bucket using the AWS SDK.
- **Parameters:**
    1. **`bucket`:** The name of the S3 bucket where the file will be stored.
    2. **`file`:** An object containing the file's key (path), body (data), and content type.
- **Process:**
    1. **Setting Up Parameters:**
        - Creates parameters required for the `PutObjectCommand`:
            - **Bucket:** The target S3 bucket.
            - **Key:** The file path/key within the bucket.
            - **Body:** The file's binary data.
            - **ContentType:** The MIME type of the file.
            - **ServerSideEncryption:** Sets encryption to `'AES256'` to ensure the file is encrypted at rest.
    2. **Executing the Upload Command:**
        - Creates an instance of `PutObjectCommand` with the specified parameters.
        - Sends the command using the S3 client.
    3. **Logging Success:**
        - Upon successful upload, logs a message indicating the file has been uploaded.

checkFileExists(applicationId, originalBuffer) (Commented Out)

- **Purpose:**

1. Intended to verify that the uploaded file exists in the S3 bucket and matches the original file by comparing file sizes and MD5 hashes.
2. **Note:** This function is commented out and not actively used in the current code.

- **Process Overview:**
    1. **Retrieving the File from S3:**
        - Attempts to get the file from S3 using `GetObjectCommand`.
        - Reads the file data into a buffer.
    2. **Comparing File Sizes:**
        - Checks if the size of the original file matches the size of the downloaded file.
    3. **Verifying File Integrity:**
        - Generates MD5 hashes for both the original and downloaded files.
        - Compares the hashes to confirm the files are identical.
    4. **Returning Verification Result:**
        - Returns `true` if the files match.
        - Returns `false` if there is a mismatch or an error occurs.

streamToBuffer(stream)

- **Purpose:**
    1. Converts a readable stream into a buffer.
    2. Used when reading file data from S3, which provides data as a stream.
- **Process:**
    1. **Collecting Data Chunks:**
        - Listens to the `'data'` event to collect chunks of data into an array.
    2. **Concatenating Chunks:**
        - On the `'end'` event, concatenates all chunks into a single buffer.
    3. **Error Handling:**
        - Listens for the `'error'` event to handle any stream errors.

Error Handling and Logging

- **Try-Catch Blocks:**
    - Encloses asynchronous operations in `try-catch` blocks to catch exceptions.
- **Logging:**
    - Logs detailed error messages to the console for troubleshooting.
    - Logs success messages upon successful operations.
- **Response to Client:**
    - Sends appropriate HTTP responses based on the outcome:
        - **Success:** 200 status code with a success message.
        - **Failure:** 500 status code with an error message.

Security Considerations

- **Environment Variables:**
  - Credentials and sensitive configurations are stored in environment variables, enhancing security by avoiding hardcoding them in the codebase.
- **Server-Side Encryption:**
  - Files are uploaded to S3 with server-side encryption enabled (`'AES256'`), ensuring data is encrypted at rest in AWS.
- **File Path Structuring:**
  - Uses `applicationId` to create unique file paths, organizing files per application and preventing naming collisions.
- **Data Integrity Verification:**
  - The commented-out `checkFileExists` function indicates an intention to verify data integrity post-upload.

Commented-Out Code and Legacy Code

- **Alternative AWS Configuration:**
  - The script contains commented-out sections with alternative AWS configurations and bucket names, possibly for different environments or legacy purposes.
- **Unused Functions:**
  - **`blobToBase64`:** A function intended to convert a Blob to a Base64 string is present but not utilized.
- **Possible Enhancements:**
  - Reactivating and refining the `checkFileExists` function could enhance data integrity checks.

Process Workflow Summary

1. **Initialization:**
   - Set up AWS configurations and initialize the S3 client.
2. **Request Handling:**
   - The serverless function receives a request with `applicationId` and `file_url`.
3. **File Download:**
   - Downloads the file from `file_url` using Axios as a binary array buffer.
4. **File Upload:**
   - Calls `uploadPassport` to prepare and upload the file to S3.
   - Utilizes `uploadFile` to perform the actual upload with server-side encryption.
5. **Response:**
   - Sends a success response if the upload is successful.
   - Sends an error response if any step fails.
6. **Logging:**
   - Logs operations and errors for monitoring and debugging purposes.

uploadFile

This serverless script is a crucial component in automating the process of uploading files to HubSpot and updating CRM records to reflect these uploads. It streamlines the workflow by handling file downloads, conversions, uploads, and CRM updates in a seamless manner. The use of a task queue ensures that resource utilization is controlled, and tasks are processed efficiently.

Key Functionalities:

- **File Download and Upload:**
  - Efficiently downloads files from provided URLs and uploads them to HubSpot's file storage system.
- **CRM Record Update:**
  - Updates CRM objects with new information, linking uploaded files to the appropriate records.
- **Concurrency Management:**
  - Employs a task queue to manage the execution of tasks, preventing resource contention and ensuring orderly processing.
- **Error Handling and Logging:**
  - Implements logging for errors and operational information, aiding in monitoring and troubleshooting.
- **Security Measures:**
  - Utilizes secure methods for handling authentication tokens and sensitive data.
  - Ensures that temporary files are properly managed and that uploaded files are secured.

Module Imports and Dependencies

- **File System (`fs`):**
  - Utilizes Node.js's built-in `fs` module to interact with the file system.
  - Responsible for writing temporary files and deleting them after the upload process is complete.
- **Axios:**
  - A promise-based HTTP client used for making HTTP requests.
  - Employed for fetching binary data from file URLs and making POST and PATCH requests to HubSpot's APIs.
- **Path:**
  - Node.js's built-in `path` module used for handling and manipulating file paths.
  - Assists in constructing file paths in a platform-independent manner.
- **Authentication Token (`authToken`):**
  - Retrieves the HubSpot API authentication token from an environment variable (`process.env.uploadApi`).

  - This token is critical for authorizing API requests to HubSpot's services.

Task Queue Class

- **Purpose:**
  - Implements a concurrency-controlled task queue to manage the execution of asynchronous tasks.
  - Ensures that the number of tasks running simultaneously does not exceed the specified concurrency limit.
- **Constructor:**
  - Initializes the queue as an empty array.
  - Sets the concurrency limit based on the provided parameter (in this case, 1).
  - Initializes a counter to track the number of currently running tasks.
- **Methods:**
  - **`runTask(task)`:**
    - Adds a new task (function) to the queue.
    - Invokes the `next()` method to attempt executing the task.
  - **`async next()`:**
    - Checks if the number of running tasks is below the concurrency limit and if there are tasks in the queue.
    - While these conditions are true, it dequeues a task from the queue.
    - Increments the running task counter.
    - Executes the task and attaches `then` and `catch` handlers:
      - On successful completion, decrements the running counter and calls `next()` to proceed with the next task.
      - On failure, logs the error, decrements the running counter, and calls `next()`.
- **Instance Creation:**
  - An instance of the `TaskQueue` class is created with a concurrency limit of one, ensuring tasks are processed sequentially.

Main Exported Function (`exports.main`)

- **Purpose:**
  - Serves as the entry point for the serverless function.
  - Handles incoming requests, orchestrates the file upload process, and updates the CRM object accordingly.
- **Request Data Extraction:**
  - Extracts necessary data from the incoming request (`context.body`):
    - **`application_id`:** The identifier of the CRM object (application) to be updated.

        - **`file_data`**: The URL or data of the file to be uploaded.
        - **`file_name`**: The desired name for the uploaded file.
        - **`property`**: The CRM object property that will be updated with the file URL.
- **HTTP Headers Configuration:**
    - Sets up HTTP headers for API requests to HubSpot, including:
        - **Authorization:** Uses the `authToken` for bearer token authentication.
        - **Content-Type:** Specifies the content type as `application/json` for JSON payloads.
- **Variable Declarations:**
    - **`application_object_type_id`**: The object type ID for the application CRM object in HubSpot.
    - **`send_response`**: Initialized to hold the response data, though not explicitly used later.

Helper Functions

fetchBinaryData(url)

- **Purpose:**
    - Fetches binary data from a given URL.
    - Used to download the file that needs to be uploaded to HubSpot.
- **Process:**
    - Makes a GET request to the specified URL using Axios.
    - Sets `responseType` to `'arraybuffer'` to receive the data in binary format.
    - Returns the binary data upon successful retrieval.
    - Implements error handling to throw an informative error message if the request fails.

getContentTypeFromBase64(base64String)

- **Purpose:**
    - Extracts content type information from a base64-encoded string.
    - Not effectively utilized in the current implementation.
- **Process:**
    - Attempts to parse the base64 string to extract the content type.
    - Splits the string based on known patterns.
    - Returns an array of parts, potentially containing the content type information.

uploadBase64AsFile(base64Image, file_name)

- **Purpose:**

    1. Converts a base64-encoded image into a file and uploads it to HubSpot's file storage using the HubSpot Files API.
- **Process:**
    1. **Base64 Conversion:**
        - Removes the Data URI scheme from the base64 string if present.
        - Decodes the base64 string into binary data and creates a Buffer.
    2. **Writing to Temporary File:**
        - Defines a temporary file path, typically in the `/tmp` directory, suitable for serverless environments.
        - Writes the binary data to a temporary file using `fs.writeFileSync`.
    3. **Preparing Multipart Form Data:**
        - Generates a unique boundary string for the multipart/form-data request.
        - Constructs the multipart form data as a string, including:
            - **`folderPath`**: Specifies the target folder in HubSpot's file system (`/uploads`).
            - **`options`**: Includes options like setting the file access to private.
            - **File Content:** Adds the file content with appropriate content disposition and content type headers.
    4. **Creating the Payload:**
        - Combines the form data headers and the file content into a single Buffer (`payload`).
        - Ensures the correct boundary is set in the `Content-Type` header.
    5. **Uploading the File:**
        - Makes a POST request to `https://api.hubapi.com/files/v3/files` using Axios.
        - Sends the `payload` with the necessary headers.
        - Sets `maxContentLength` and `maxBodyLength` to `Infinity` to handle large files.
    6. **Handling the Response:**
        - On success, extracts and returns the URL of the uploaded file from the response data.
    7. **Cleanup:**
        - Deletes the temporary file using `fs.unlinkSync` in the `finally` block to prevent accumulation of temporary files.
    8. **Error Handling:**
        - Catches and logs any errors that occur during the upload process.

handleUpload(file_name, file_url)

- **Purpose:**
    1. Orchestrates the downloading of the file and uploading it to HubSpot.
- **Process:**

1. **Downloading the File:**
    - Calls `fetchBinaryData(file_url)` to retrieve the file's binary data.
2. **Converting to Base64:**
    - Creates a `Uint8Array` from the binary data.
    - Converts the binary data into a binary string in chunks to handle potential large file sizes.
    - Encodes the binary string into a base64 string using `btoa`.
3. **Uploading the File:**
    - Calls `uploadBase64AsFile(base64String, file_name)` to upload the base64-encoded file to HubSpot.
4. **Returning the File URL:**
    - Returns the URL of the uploaded file obtained from the upload function.

getFileUrl()

- **Purpose:**
    - Acts as a helper function to initiate the file handling process and retrieve the uploaded file URL.
- **Process:**
    - Calls `handleUpload(file_name, file_data)` to execute the download and upload sequence.
    - Returns the file URL resulting from the upload.

updateObject(object_type_id, object_id, properties)

- **Purpose:**
    1. Updates a CRM object in HubSpot with new property values, linking the uploaded file.
- **Process:**
    1. **Constructing the Request:**
        - Forms the endpoint URL using the object type ID and object ID.
    2. **Making the PATCH Request:**
        - Sends a PATCH request to the HubSpot CRM Objects API.
        - Includes the updated properties in the request body.
        - Uses the predefined headers for authorization.
    3. **Handling the Response:**
        - Returns the response data upon a successful update.
    4. **Error Handling:**
        - Catches any errors during the request and logs the error message.

runFunction()

- **Purpose:**

1. The main function that orchestrates the overall file upload and CRM update process.

- **Process:**
  1. **Obtaining the Uploaded File URL:**
     - Calls `getFileUrl()` to execute the file download and upload process, receiving the uploaded file's URL.
  2. **Preparing Properties for Update:**
     - Constructs a properties object where the key is the property name extracted from the request (`property`), and the value is the uploaded file URL.
  3. **Updating the CRM Object:**
     - Calls `updateObject(application_object_type_id, application_id, properties)` to update the specified CRM object with the new file URL.
  4. **Sending the Response:**
     - Uses `sendResponse` to return the response data back to the client, indicating the result of the update operation.

Adding the Task to the Queue

- **Queue Execution:**
  - The `runFunction` is added to the task queue by calling `queue.runTask(runFunction)`.
  - Given the concurrency limit, tasks are processed sequentially, ensuring that only one task runs at a time.

Workflow Summary

- **Incoming Request Handling:**
  - The serverless function receives a request containing necessary data for processing:
    - **Application ID:** Identifies which CRM object to update.
    - **File Data:** The URL or base64 data of the file to be uploaded.
    - **File Name:** The desired name for the file in the HubSpot file system.
    - **Property:** The CRM object property to be updated with the file URL.
- **File Processing and Upload:**
  - **Downloading the File:**
    - Fetches the file from the provided URL as binary data.
  - **Converting and Uploading:**
    - Converts binary data to a base64-encoded string.
    - Writes the data to a temporary file.
    - Uploads the file to HubSpot using the Files API.

        - Receives the URL of the uploaded file upon success.
- **CRM Object Update:**
    - Updates the specified CRM object's property with the new file URL, effectively linking the uploaded file to the CRM record.
- **Response to Client:**
    - Sends a response back to the client containing the result of the CRM object update, typically including the updated object data.

Error Handling and Logging

- **Error Logging:**
    - Errors encountered during HTTP requests, file operations, or API interactions are logged to the console with detailed messages for troubleshooting.
- **Task Queue Error Handling:**
    - If a task fails, the task queue logs the error and ensures that the queue continues processing subsequent tasks.
- **Exception Management:**
    - Exceptions are propagated appropriately to enable higher-level error handling mechanisms to respond accordingly.

Security Considerations

- **API Token Management:**
    - The HubSpot API authentication token is securely retrieved from an environment variable, preventing hardcoding sensitive credentials in the codebase.
- **Data Security:**
    - Temporary files containing sensitive data are stored in a secure location (`/tmp`) and are deleted immediately after use to minimize exposure risk.
- **Authorization Headers:**
    - All API requests include authorization headers to ensure that only authenticated and authorized interactions occur with the HubSpot APIs.
- **Private File Access:**
    - When uploading files to HubSpot, the `options` object specifies `access: "PRIVATE"`, ensuring that uploaded files are not publicly accessible.

submitForm

This JavaScript file is designed as a serverless function to handle form submissions securely and efficiently. It serves as an intermediary between the client-side form submissions and the HubSpot Forms API. The function captures form data, submits it to HubSpot, handles any errors that occur during the process, and logs those errors to an external logging service (Pipedream) for monitoring and debugging purposes. By doing so, it ensures reliable form submission handling, error tracking, and provides feedback to the client about the success or failure of the submission.

Initialization and Environment Setup

- **Authentication Token Retrieval:**
    - The script begins by retrieving the **HubSpot API authentication token** from an environment variable named `uploadApi`.
    - This token (`authToken`) is essential for authorizing requests to the HubSpot Forms API.
    - **Security Note:** Storing sensitive tokens in environment variables is a best practice to prevent hardcoding credentials in the codebase.

Main Exported Function (`exports.main`)

- **Purpose:**
    1. Acts as the entry point for the serverless function.
    2. Handles incoming requests containing form data, submits the data to HubSpot, and manages error reporting.
- **Parameters:**
    1. **`context`**: An object that contains request data, including the request body (`context.body`) with the payload and form ID.
    2. **`sendResponse`**: A function used to send HTTP responses back to the client.
- **Process Flow:**
    1. **Setting Up Headers:**
        - Configures the HTTP headers required for API requests, including:
            - **Authorization Header:** Uses the `authToken` to set up bearer token authentication.
            - **Content-Type Header:** Specifies that the content type is `application/json` for JSON payloads.
    2. **Extracting Data from Request:**
        - Retrieves the **form payload** (`payload`) and **form ID** (`formId`) from the request body.
        - **`payload`**: Contains the form data to be submitted to HubSpot.
        - **`formId`**: Identifies the specific HubSpot form to which the data will be submitted.
    3. **Defining Helper Functions:**
        - Several helper functions are defined within the main function to encapsulate specific tasks.

Helper Functions

submitToRequestBin(errorDetails)

- **Purpose:**
  1. Sends error details to an external logging service (Pipedream) for monitoring and debugging.
  2. Ensures that any errors encountered during form submission are captured and can be reviewed.
- **Process:**
  1. **Endpoint Configuration:**
     - Defines the endpoint URL for the Pipedream service, which acts as a webhook to receive error logs.
  2. **Error Reporting:**
     - Sends a POST request to the Pipedream endpoint with the error details in the request body.
     - Sets the `Content-Type` header to `application/json`.
  3. **Error Handling:**
     - If the request to Pipedream fails, it logs an error message indicating the failure.
     - Uses a `try-catch` block to handle any network errors during the reporting process.

sendPayload(url, payload, functionName)

- **Purpose:**
  1. Sends a POST request to a specified URL with the given payload.
  2. Handles HTTP response validation and error reporting.
- **Parameters:**
  1. **`url`:** The endpoint URL to which the request will be sent.
  2. **`payload`:** The JSON data to be sent in the request body.
  3. **`functionName`:** The name of the calling function, used for contextual error reporting.
- **Process:**
  1. **Sending the Request:**
     - Logs a message indicating that a payload is being sent to the specified URL.
     - Sends a POST request using the `fetch` API with the configured headers and payload.
  2. **Response Handling:**
     - Checks if the response is successful (HTTP status code in the 200 range).
     - If the response is not OK, it proceeds to capture detailed error information.
  3. **Error Capture and Reporting:**
     - **Error Details Collected:**
       - **Status Code (`status`):** The HTTP status code returned.

        - **Status Text (`statusText`):** The textual representation of the status code.
        - **URL (`url`):** The endpoint URL that was called.
        - **Function Name (`functionName`):** The name of the function making the request.
        - **Correlation ID (`correlationId`):** Extracted from the response headers (`x-hubspot-correlation-id`) for tracing.
        - **Response Body (`responseBody`):** The response text from the server.
    - Logs the error details to the console for immediate visibility.
    - Calls `submitToRequestBin` to send the error details to Pipedream for external logging.
    - Throws an error with a message that includes the status code, function name, and status text.
4. **Exception Handling:**
    - In the `catch` block, handles any network errors or exceptions that occur during the request.
    - **Error Details Collected:**
        - **Message (`message`):** The error message.
        - **URL (`url`):** The endpoint URL.
        - **Function Name (`functionName`):** The name of the function making the request.
        - **Stack Trace (`stack`):** The error stack trace for debugging.
    - Logs the error details to the console.
    - Sends the error details to Pipedream via `submitToRequestBin`.
    - Rethrows the error to be handled by higher-level error management.

submitForm()

- **Purpose:**
    1. Submits the form data to the HubSpot Forms API.
- **Process:**
    1. **Endpoint Construction:**
        - Constructs the HubSpot Forms API endpoint URL using the portal ID and `formId`.
            - **Example Format:** `https://api.hsforms.com/submissions/v3/integration/secure/submit/{portalId}/{formId}`
    2. **Form Submission:**
        - Logs a message indicating that the form is being submitted to the specified endpoint.

        - Calls `sendPayload` with the endpoint, payload, and the function name `'submitForm'`.
        - Receives the response data from the form submission.
    3. **Return Value:**
        - Returns the response data obtained from the HubSpot API.

Error Handling and Logging in the Main Function

- **Try-Catch Block:**
    1. Encloses the main operations within a `try-catch` block to manage exceptions.
- **Process:**
    1. **Form Submission Attempt:**
        - Logs a message indicating the start of the form submission process.
        - Calls `submitForm()` to submit the form data to HubSpot.
        - Logs a success message along with the response data upon successful submission.
        - Uses `sendResponse` to send the response data back to the client, indicating a successful operation.
    2. **Error Handling:**
        - If an error occurs during form submission:
            - Logs an error message indicating the error encountered in the main function.
            - Calls `submitToRequestBin` to report the error details to Pipedream.
            - Uses `sendResponse` to send an error response back to the client with a 500 status code, indicating a server error.

Workflow Summary

1. **Initialization:**
    - Retrieves the authentication token and sets up request headers.
2. **Request Handling:**
    - Extracts the form payload and form ID from the incoming request.
3. **Form Submission:**
    - Attempts to submit the form data to the HubSpot Forms API via the `submitForm()` function.
4. **Error Management:**
    - If the submission fails or an error occurs, detailed error information is captured.
    - Errors are logged both to the console and sent to Pipedream for external logging.
    - An error response is sent back to the client with appropriate status codes and messages.
5. **Response to Client:**

    - On success, sends the response data from HubSpot back to the client.
    - On failure, sends an error message with a 500 status code.

External Services Integration

- **HubSpot Forms API:**
    - The script interacts with the HubSpot Forms API to submit form data securely.
    - Requires proper authentication via the bearer token.
- **Pipedream (External Logging Service):**
    - Used for capturing and storing error details.
    - Allows developers to monitor errors that occur in the serverless function for debugging and reliability purposes.

Security Considerations

- **Authentication Token Handling:**
    - The `authToken` is securely retrieved from environment variables, ensuring that sensitive credentials are not exposed in the codebase.
- **Error Reporting:**
    - When reporting errors to Pipedream, care is taken to include necessary details without exposing sensitive information.
    - Error details include status codes, messages, URLs, function names, and stack traces.
- **Data Privacy:**
    - Form data (`payload`) is handled securely and only transmitted to the intended API endpoints.

Error Handling and Logging

- **Console Logging:**
    - Logs significant events and errors to the console for immediate visibility during execution.
- **External Logging:**
    - Uses the `submitToRequestBin` function to send error details to Pipedream, enabling asynchronous monitoring and alerting.
- **Structured Error Information:**
    - Captures comprehensive error details, including correlation IDs from HubSpot, to facilitate easier debugging and tracing of issues.

handleActions

This JavaScript file is designed as a serverless function to interact with the HubSpot API, performing various operations such as updating object properties, retrieving object data, uploading images, and managing associations between objects. It handles incoming requests that specify the desired operation and executes the appropriate API calls to HubSpot. By facilitating these operations, the script serves as a backend utility to manipulate CRM data, manage content assets, and maintain relationships between different entities within HubSpot, such as contacts, companies, deals, or custom objects.

Module Imports and Dependencies

- **File System (`fs`):**
  - Node.js built-in module used for interacting with the file system.
  - In this script, it's required but not actively used in the provided code. It might be a remnant from earlier versions or reserved for future use.
- **Axios:**
  - A promise-based HTTP client for Node.js used to make HTTP requests to external APIs.
  - Employed extensively in this script for making GET, POST, PUT, PATCH, and DELETE requests to the HubSpot API endpoints.
- **Path:**
  - Node.js built-in module that provides utilities for working with file and directory paths.
  - Imported but not directly used in the current implementation.
- **FormData:**
  - An npm package (`form-data`) used to create readable "multipart/form-data" streams.
  - Essential for constructing form data when uploading files via HTTP POST requests.
- **Authentication Token (`authToken`):**
  - Retrieved from an environment variable (`process.env.apiKey`).
  - Used for authenticating API requests to HubSpot.
  - **Security Note:** Storing API keys in environment variables is a best practice to prevent exposing sensitive information in the codebase.

Main Exported Function (`exports.main`)

- **Purpose:**
  - Acts as the entry point for the serverless function.
  - Handles incoming requests, determines the operation to perform, and orchestrates the appropriate actions using the helper functions defined within.

- **Parameters:**
  - **`context`**: An object containing request data, including the body (`context.body`) where parameters like `properties`, `properties_to_get`, and other identifiers are expected.
  - **`sendResponse`**: A callback function used to send responses back to the client or calling service.
- **Headers Configuration:**
  - Sets up default headers for Axios requests, including:
    - **Authorization Header:** Uses the `authToken` for bearer token authentication with HubSpot APIs.
    - **Content-Type Header:** Set to `application/json` for JSON payloads.
- **Variable Initialization:**
  - Extracts `properties` and `properties_to_get` from `context.body`.
  - Initializes variables `newObjectId` and `fileURL` to store results from operations.

Helper Functions

updateObject(endpoint, properties, associations, headers)

- **Purpose:**
  - Updates an existing object in HubSpot by sending a PATCH request to the specified endpoint.
- **Parameters:**
  - **`endpoint`**: The HubSpot API endpoint URL for the object to be updated.
  - **`properties`**: An object containing the properties to update on the HubSpot object.
  - **`associations`**: An array of associations to update or maintain for the object.
  - **`headers`**: HTTP headers for the request, including authorization.
- **Process:**
  - Sends a PATCH request using Axios with the provided properties and associations.
  - Logs the result upon success.
  - Returns the response data containing the updated object information.
  - Handles and logs errors, rethrowing them for higher-level handling.

createAssociation(endpoint, body, headers)

- **Purpose:**
  - Creates an association between two objects in HubSpot by sending a PUT request.
- **Parameters:**
  - **`endpoint`**: The HubSpot API endpoint URL for creating the association.
  - **`body`**: The request body containing association details.

    - **`headers`**: HTTP headers for the request.
- **Process:**
    - Sends a PUT request with the association data.
    - Logs and returns the response data upon success.
    - Handles and logs errors, rethrowing them as needed.

deleteAssociation(endpoint, body, headers)

- **Purpose:**
    - Deletes an existing association between two HubSpot objects.
- **Parameters:**
    - Similar to `createAssociation`.
- **Process:**
    - Sends a DELETE request with the association data.
    - Logs and returns the response data upon success.
    - Handles and logs errors.

getObjects(endpoint)

- **Purpose:**
    - Retrieves properties of a HubSpot object using a GET request.
- **Parameters:**
    - **`endpoint`**: The HubSpot API endpoint URL for the object.
- **Process:**
    - Sends a GET request to the specified endpoint.
    - Logs and returns the retrieved object data.
    - Handles and logs errors.

unescapeHtml(escapedHtml)

- **Purpose:**
    - Unescapes HTML entities in a string, converting them back to their original characters.
- **Parameters:**
    - **`escapedHtml`**: The string containing escaped HTML entities.
- **Process:**
    - Uses regular expressions to replace common HTML entities with their corresponding characters.
    - Returns the unescaped string.
- **Usage:**
    - Applied to the `content` and `bio` properties if they exist, ensuring that any HTML content is correctly formatted before being sent to HubSpot.

uploadBase64AsFile(base64Image, uploadUrl)

- **Purpose:**
    1. Uploads a base64-encoded image to HubSpot's file manager.
- **Parameters:**
    1. **`base64Image`**: The base64-encoded image string, including the data URI scheme.
    2. **`uploadUrl`**: The HubSpot API endpoint for file uploads.
- **Process:**
    1. **Parsing Base64 Data:**
        - Uses a regular expression to extract the content type and base64 data from the data URI.
        - Validates the format, throwing an error if invalid.
    2. **Creating a Buffer:**
        - Converts the base64 data into a binary `Buffer`.
    3. **Generating a Unique Filename:**
        - Extracts the file extension from the content type (e.g., 'png', 'jpeg').
        - Creates a filename using a timestamp to ensure uniqueness.
    4. **Constructing Form Data:**
        - Creates an instance of `FormData`.
        - Appends the file buffer, `folderPath`, and `options` to the form data.
            - **`folderPath`**: Specifies the directory in HubSpot's file system where the file will be stored (e.g., '/uploads').
            - **`options`**: Includes settings like access level (`'PUBLIC_INDEXABLE'`), making the file publicly accessible.
    5. **Uploading the File:**
        - Sends a POST request to the `uploadUrl` with the form data.
        - Merges the form data headers with the authorization header.
        - Sets `maxContentLength` and `maxBodyLength` to `Infinity` to handle large files.
    6. **Handling the Response:**
        - Logs the success message and the URL of the uploaded file.
        - Returns the file URL.
    7. **Error Handling:**
        - Catches and logs any errors during the upload process.
        - Rethrows the error for higher-level handling.

Processing File Uploads

- **Variables:**
    - **`imageFileUrl`**: To store the URL of the uploaded `featured_image`.
    - **`profilePictureUrl`**: To store the URL of the uploaded `profile_picture`.

- **Upload Logic:**
  - **`featured_image`:**
    - Checks if `properties` contains a `featured_image`.
    - Calls `uploadBase64AsFile` to upload the image.
    - Updates `properties.featured_image` with the URL returned from the upload.
  - **`profile_picture`:**
    - Similar process as `featured_image`.
  - **Error Handling:**
    - Each upload is enclosed in a `try-catch` block.
    - Errors are logged, and the process continues, allowing the script to proceed even if one upload fails.

Main Logic to Handle Different Operations

The script determines the operation to perform based on the presence of certain parameters in the request:

Retrieving Object Properties

- **Condition:**
  1. If `properties_to_get` is present in the request.
- **Process:**
  1. **Extracting Identifiers:**
     - Retrieves `objectTypeId` and `objectId` from `context.body`.
  2. **Constructing the Endpoint:**
     - Joins the properties to get into a query string.
     - Builds the GET endpoint URL with the object type, ID, and requested properties.
  3. **Fetching Object Data:**
     - Calls `getObjects` with the constructed endpoint.
  4. **Sending Response:**
     - On success, sends the retrieved data back to the client.
     - On failure, sends an error message with a 500 status code.

Updating Object Properties

- **Condition:**
  1. If `properties` is present and `properties_to_get` is not.
- **Process:**
  1. **Extracting Identifiers:**

        - Retrieves `objectType`, `objectId`, and optional `associations` from `context.body`.
    2. **Constructing the Endpoint:**
        - Builds the PATCH endpoint URL using the object type and ID.
    3. **Updating Object:**
        - Calls `updateObject` with the endpoint, properties, associations, and headers.
    4. **Sending Response:**
        - On success, sends the updated object data with a 200 status code.
        - On failure, sends an error message with a 500 status code.

Creating or Deleting Associations

- **Condition:**
    1. If neither `properties_to_get` nor `properties` are present.
- **Process:**
    1. **Extracting Identifiers:**
        - Retrieves `fromObjectType`, `fromObjectId`, `toObjectType`, `toObjectId`, `assocTypeId`, and `option` from `context.body`.
    2. **Constructing the Endpoint:**
        - Builds the endpoint URL for associations using the object types and IDs.
    3. **Preparing the Body:**
        - Creates a body array with the association details, specifying the category as `'USER_DEFINED'` and including the `associationTypeId`.
    4. **Performing the Operation:**
        - If `option` is `'create'`, calls `createAssociation`.
        - If `option` is `'delete'`, calls `deleteAssociation`.
        - Throws an error if `option` is invalid.
    5. **Sending Response:**
        - On success, sends the response data with a 200 status code.
        - On failure, sends an error message with a 500 status code.

Error Handling and Response Management

- **Error Logging:**
    - All helper functions log errors to the console, providing immediate feedback during execution.
- **Response to Client:**
    - Uses `sendResponse` to send data or error messages back to the client.
    - Sets appropriate HTTP status codes (200 for success, 500 for server errors).

Security Considerations

- **Authentication:**
  - API key (`authToken`) is securely retrieved from environment variables.
- **Data Sanitization:**
  - Unescapes HTML content in `content` and `bio` properties to prevent malformed data.
- **File Uploads:**
  - Uploaded files are set with `'PUBLIC_INDEXABLE'` access, making them publicly accessible.
    - **Note:** Depending on the use case, this might need to be adjusted to `'PRIVATE'` if sensitive data is being uploaded.
- **Error Messages:**
  - Errors are logged but not overly detailed in responses to prevent exposure of sensitive information.

Workflow Summary

1. **Initialization:**
   - Imports necessary modules and retrieves the authentication token.
2. **Request Handling:**
   - Extracts parameters from the incoming request (`context.body`).
3. **Data Processing:**
   - Unescapes HTML entities in `content` and `bio` properties.
4. **File Uploads:**
   - Checks for `featured_image` and `profile_picture` in properties.
   - Uploads images to HubSpot's file manager and updates properties with the file URLs.
5. **Operation Determination:**
   - Decides which operation to perform based on the presence of `properties_to_get` and `properties`.
6. **Executing Operations:**
   - Retrieves object properties if requested.
   - Updates object properties if provided.
   - Creates or deletes associations if specified.
7. **Response Management:**
   - Sends responses back to the client with appropriate status codes and data.
8. **Error Handling:**
   - Logs errors and sends error messages back to the client when exceptions occur.

GraphQL

`student_data.graphql`: Fetches relevant student information.

Workflows

[PORTAL] Associate Applications

This function connects HubSpot CRM objects based on a user-selected program type and program data. It searches for relevant program and session objects, establishes associations between the current HubSpot record and program/session objects, and updates related application data. Essentially, it's building relationships between different data entities in the HubSpot CRM to reflect real-world relationships between programs, sessions, and applications based on the workflow input.

Main Logic

- If a `program` is selected, it performs the following:
    - Adjusts the program name by replacing "In-Person" with "Customized" if found.
    - Searches for a program object based on `program_name` using the `searchObjects` function.
    - Retrieves the parent program details using `getData`.
    - Searches for a session object associated with the program, obtaining the term and year properties.
    - Fetches any applications associated with the current object (`hs_object_id`) using `getAssociations`.

Associations and Updates

- Creates associations between the program and application using `createAssociation` by setting a user-defined association.
- Creates another association if a session object is found.
- Updates the associated application object with the parent program name using `updateObject`.

[PORTAL] Associate Current Institution

This function connects an application object in HubSpot to a specified company based on the `current_institution__c` field and also links the associated contact with the company. The key steps involve searching for the company, retrieving associations, and using HubSpot's API to create associations between different objects. This process effectively builds relationships between application, contact, and company data within the HubSpot CRM system, making it easier to manage interconnected data records.

Retrieving Associated Contact Objects

- Calls getAssociations to find contact objects associated with the application object (hs_object_id) using application_object_type_id.
- Retrieves the first associated contact's ID and stores it in this_contact.

Searching for the Company Object

- Uses searchObjects to search for a company with a name matching current_institution__c.
- Sets company_filters to filter by name and retrieves properties company_name and hs_object_id.
- Retrieves the list of companies matching the criteria and extracts the first company's id into toObjectId.

Creating Associations

- Constructs an endpoint URL to associate the application object (hs_object_id) with the identified company (toObjectId).
- Sets body with associationCategory as USER_DEFINED and associationTypeId as 150 and calls createAssociation to create this association.

Associating the Contact with the Company

- If this_contact is found, creates another association between the contact and the company.
- Constructs next_endpoint for the contact-company association and sets next_body with associationCategory as HUBSPOT_DEFINED and associationTypeId as 1.
- Calls createAssociation again to establish this association and logs the response.

[PORTAL] Check Passport Upload Complete Box When Submitted

Updates "Passport Upload Received?" to "Yes" when a form "Upload - Passport" is submitted.

[PORTAL] Copy Billing Address to Contact

Copies billing address from the payment form to the contact record, and updates state names to match the global picklist.

[PORTAL] Copy Contact Property Values to Application Properties

Takes all the info brought in from the application, creates an application record, and copies properties from the Contact record to the Application record.

Also copies (Contact) current institution to (Contact) company name.

[PORTAL] Copy Program Required Forms to Application Pending Forms

Updates "Pending Forms" with "Required Forms" when "Required Forms" is known.

[PORTAL] Create Combined Program Selection Property

This function is designed to update a property (`combined_program_selector`) in both a contact object and an application object within HubSpot. It retrieves upcoming session data based on the current date and builds a list of unique program names, which is then used to populate a selection property. The updated options are sent back to HubSpot to ensure that users have up-to-date program selections available in their CRM system. This makes the selection process dynamic and responsive to real-time data, ensuring that the CRM reflects the most current program options for applications and contacts.

Setting Initial Values

- The `program_object_id` is set to the `hs_object_id`.
- The `parent_program_id` is set to `parent_program_id__c`.
- Retrieves the current Unix timestamp (`today`).

Fetching Parent Program Data

- Calls `getData` to fetch the parent program's details (specifically `program_name`) using the `parent_program_id`.

Searching for Upcoming Sessions

- Sets `session_filters` to find sessions where `application_deadline__c` is greater than or equal to `today`.
- Sorts sessions by `application_deadline__c` in ascending order using `session_sorts`.
- Uses `searchObjects` to search for upcoming session objects and retrieves details such as `program_name`, `program_term`, and `program_year`.

Building the `options` Array

- Iterates over the `upcoming_sessions` to build an array of unique `program_name` values.
- For each unique program name, creates a `data` object with `label`, `value`, `hidden`, and `displayOrder`.
- Adds each `data` object to the `options` array.

Updating Properties with `options` Data

- Constructs `data` containing the `options` array.
- Uses `updateProperty` to update two different objects:
  - `prop_object` (a contact object represented by `"0-1"`).
  - `application_object_type_id` (custom application object).
- Both objects have their `combined_program_selector` property updated with the `options` array.

[PORTAL] Create Program Choice Dropdown

This function retrieves all program objects from HubSpot, extracts unique `program_name` values, and updates a property (`program_selection`) for contact objects in the HubSpot CRM with these values. The goal is to ensure that the `program_selection` property reflects the latest program names available, making it easier for users to select programs from an up-to-date list. This dynamic approach ensures that the HubSpot property stays current with any changes to the program data within the CRM.

Fetching Program Objects

- Defines `props_to_get` as an array containing `program_name`.
- Joins `props_to_get` into a comma-separated string (`prop_string`), which is used to specify the properties to retrieve.

- Constructs the `get_endpoint` URL to fetch all program objects with their `program_name` property.
- Calls `getObjects(get_endpoint)` to retrieve all program objects and stores them in `all_objects`.

Building the New Property Options

- Initializes `new_prop_options` as an empty array to store unique program name options.
- Initializes `counted` as an array to keep track of program names that have already been processed.
- Iterates through `all_objects`:
    - For each `program_name` not already in `counted`, creates a `data` object with:
        - `label`: Program name
        - `value`: Program name
        - `hidden`: Set to `false`
    - Adds this `data` object to `new_prop_options` and the program name to `counted`.

Updating the Property in HubSpot

- Calls `updateProp("0-1", property_name, new_prop_options)` to update the `program_selection` property for contact objects (`0-1` representing contacts) with the newly created options.
- Logs the updated options from the response to the console.

[PORTAL] Fix Old Apps Pending Fields

This function processes lists of submitted and pending forms to update a HubSpot object by removing any forms from the pending list that have already been submitted. It then updates the `pending_forms` property on the specified object with the filtered list, ensuring that only genuinely pending forms are stored. This logic helps maintain an up-to-date record of form submissions in the HubSpot CRM, ensuring that users can accurately track which forms still need to be completed.

Splitting and Processing Form Strings

- Converts `submitted_forms` into an array (`submitted_array`) using `split(";")` if the `submitted_forms` string contains a semicolon (`;`), otherwise keeps it as a single value.

- Converts `pending_forms` into an array (`pending_array`) using `split(";")` if the string contains a semicolon, otherwise sets it to `null`.

Filtering Pending Forms

- If both `submitted_array` and `pending_array` exist:
  - Uses `filter()` to remove items from `pending_array` that are present in `submitted_array`.
  - Joins the remaining items with `";"` to form the final string `final`.
- If only `pending_array` exists, joins it with `";"` to form `final`.
- If `pending_array` is `null`, `final` remains `null`.

Updating the Object in HubSpot

- Constructs an object `properties` with `pending_forms` set to `final`.
- Calls `updateObject("2-14511828", hs_object_id, properties)` to update the object with ID `hs_object_id` (likely representing an application object) with the new `pending_forms` data.

[PORTAL] Handle Application Associations

Summary: Updates "Parent Program" property when "Number of Associated Programs" is greater than 0.

[PORTAL] Handle Application Unique IDs

Updates the "Unique ID" property with the enrolled object's ID when the "Record ID" is known.

[PORTAL] Handle Arrival Date/Time

Puts the arrival date/time in the format needed for BASE.

[PORTAL] Handle File Upload JSON

Updates object properties with JSON data from uploaded files when specific document upload events are completed.

[PORTAL] Handle Form Submissions

Updates the "Forms Submitted" property when a contact submits the "Last Form Submitted" form.

[PORTAL] Handle Payment Success

This function looks up a contact based on the customer's email, finds the associated applications, identifies which one requires a confirmation deposit, and updates that application to indicate that the confirmation deposit has been paid. This is crucial for ensuring that the HubSpot CRM accurately reflects payment status, helping maintain clear records of deposit requirements for applications associated with a particular contact. This process enables seamless integration and updating of payment data within the HubSpot CRM system, making it easier to manage financial aspects related to customer applications.

Finding the Associated Contact

- Defines `contact_filters` to search for contacts with an `email` matching `hs_customer_email`.
- Uses `searchObjects` to find matching contacts.
- Retrieves the `associated_contact_id` from the search results.

Processing the Contact's Associated Application

- If an `associated_contact_id` is found, retrieves the associations between this contact and any applications using `getAssociations`.
- Extracts application IDs and stores them in `associated_application_ids` using `extractObjectIDs`.

Determining the Most Relevant Application

- If there is only one associated application, `this_application` is set to that ID.
- If there are multiple applications, retrieves details for each (`hs_createdate` and `is_confirmation_deposit_required__c`) using `getObjectsByID`.
- Filters to find applications that require a deposit (`is_confirmation_deposit_required__c`), storing them in `deposit_required`.
- If there's only one deposit-required application, sets `this_application` to that ID.
- If there are multiple deposit-required applications, sorts them by creation date and selects the most recent.

Updating the Application

- Sets `properties` to indicate `has_paid_confirmation_deposit__c: true`.
- Calls `updateObject` to update the application object (`this_application`) with this property change.
- Logs the response from the update.

[PORTAL] Handle Program Session Associations

Updates "Associated Sessions" property when "Number of Associated Sessions" is known.

[PORTAL] Handle Session Selection

Updates session associations when a contact completes the "Choose Your Session" event.

[PORTAL] Handle Single Checkboxes

Updates checkbox properties to "false" when an application is created due to BASE limitations.

[PORTAL] Pending Forms HubDB Sync

This function retrieves rows from a specified HubDB table, extracts relevant data, and updates the `pending_forms` property of a custom object in HubSpot with this data. The main goal is to ensure that the `pending_forms` property contains up-to-date options based on the data stored in the HubDB table, providing a dynamic and real-time way to manage selection options within the HubSpot CRM system. This keeps the property values in sync with the database, enabling users to work with the most current information in their CRM workflows.

Fetching Rows from the HubDB Table

- Calls `getRows(table_id)` to retrieve all rows from the specified HubDB table.
- Stores the rows in `all_doc_types`.

Building the New Property Options

- Initializes `new_prop_options` as an empty array to store the options that will be used to update the property.
- Iterates through `all_doc_types` to create `this_data` objects, containing:
    - `label`: The document type's `name` from the HubDB row.
    - `value`: The document type's `value` from the HubDB row.

  - `displayOrder`: The order in which the document should be displayed (`order` value from HubDB row).
  - `hidden`: Set to `false`.
- Adds each `this_data` object to `new_prop_options`.

Updating the Property in HubSpot

- Calls `updateProp(object_type, property_name, new_prop_options)` to update the `pending_forms` property for the custom object with the newly created options.
- Logs the updated options from the response to the console.

[PORTAL] Set Application Properties

Updates "Application pipeline stage" to "Incomplete" when "Application pipeline" is unknown.

[PORTAL] Set Form Page Link

Updates "Student Forms Page Link" with a generated URL when an an Application object is created.

[PORTAL] Set Student Status to Alumni

This function identifies applications associated with a specific session object and updates their pipeline stage in bulk, provided they don't already have a specific stage. The process includes retrieving associated application data, filtering based on pipeline stage, and performing a bulk update for eligible applications. This function is designed to efficiently manage and update multiple HubSpot CRM records in one operation, ensuring that the application data is kept in sync with the desired pipeline stage criteria. This is particularly useful for maintaining accurate status tracking of applications linked to sessions.

Retrieving Associated Applications

- Calls `getAssociations` to get the applications associated with the session (`session_id`).
- Filters the results to find applications that have the `application_to_session_association` type.
- Extracts their IDs using `extractObjectIDs` and stores them in the `associated_applications` array.

Preparing Data for Bulk Update

- Initializes `bulk_update_inputs` as an empty array to store data for updating applications.
- Iterates over `associated_applications` to:
    - Retrieve each application's `hs_pipeline_stage` property using `getData`.
    - Checks if the `hs_pipeline_stage` is not `72206386` or `72206387`.
    - If it meets this condition, prepares a payload to set `hs_pipeline_stage` to `72206385`.
    - Adds the payload to `bulk_update_inputs`.

Executing the Bulk Update

- If `bulk_update_inputs` contains data, calls `batchUpdateObjects` to perform the batch update for the application objects.
- Logs the response from the update.

[PORTAL] Set Student Status to Onsite

This function identifies applications associated with a specific session object and updates their pipeline stage in bulk, provided they are not already in stages `72206386` or `72206387`. The new stage is set to `72206384`. The process includes retrieving associated application data, filtering based on pipeline stage, and performing a batch update for eligible applications. This ensures that application data in HubSpot CRM reflects the desired pipeline stage, making it easier to manage application progress efficiently. This function helps automate the pipeline stage update process, ensuring accurate and up-to-date status tracking for applications linked to sessions.

Retrieving Associated Applications

- Calls `getAssociations` to get the applications associated with the session (`session_id`).
- Filters the results to find applications that have the `application_to_session_association` type.
- Extracts their IDs using `extractObjectIDs` and stores them in the `associated_applications` array.

Preparing Data for Bulk Update

- Initializes `bulk_update_inputs` as an empty array to store data for updating applications.

- Iterates over `associated_applications`:
    - Retrieves each application's `hs_pipeline_stage` property using `getData`.
    - Checks if the `hs_pipeline_stage` is not `72206386` or `72206387`.
    - If the condition is met, prepares a payload to set `hs_pipeline_stage` to `72206384`.
    - Adds the payload to `bulk_update_inputs`.

Executing the Bulk Update

- If `bulk_update_inputs` contains data, calls `batchUpdateObjects` to perform the batch update for the application objects.
- Logs the response from the update.

[PORTAL] Set University Approval Form Status

Updates "Advisor Approval Status" to "Not Submitted" when a student submits the university contact form.

Portal Features and Custom Logic

The Student Portal offers an all-encompassing solution for managing study abroad applications by integrating with HubSpot's Custom Objects, HubDB tables, serverless functions, and GraphQL queries. Key features include:

1. **Dynamic Program and Session Management:** Tailors the student experience based on the specific program and session the applicant is interested in, ensuring relevant information and requirements are displayed.
2. **Multi-Step Application Tracking:** Utilizes a combination of Custom Objects and HubDB to track students through every stage of the application process, offering real-time updates and status tracking for both pre-acceptance and post-acceptance requirements.
3. **Data Security:** Leverages AWS S3 for secure file uploads, utilizing server-side encryption, and integrates data handling mechanisms to maintain the confidentiality of sensitive documents.
4. **User-Friendly Interface:** Provides a seamless user interface for students to manage their applications, track form submissions, and monitor their application's progress.

Application Process Flow

The application process is streamlined into the following stages:

1. **Application Intake and Registration:** Students initiate the process by providing their personal and contact information, selecting their desired program, and identifying the appropriate session.
2. **Login and Access:** Secure login allows students to access their applications, review progress, and complete required actions, ensuring only authorized users can manage their information.
3. **Personal & Contact Information Module:** This module ensures that all necessary personal details are collected, validated, and stored efficiently.
4. **Pre-Acceptance & Post-Acceptance Requirements:** The portal dynamically determines the required forms and documentation for each stage, guiding students on the forms that need completion. This includes conditional messaging based on the application's progress and automatically updates students when new requirements are met.
5. **Completion Flow:** Once all required forms and documents are completed, the student receives a confirmation, and the system automatically advances them to the next stage of the process.

Required Forms Flow

The portal dynamically presents students with the forms they need to complete based on their selected program and status. The form flow is designed to be intuitive and adaptable:

1. **Dynamic Form Determination:** The portal retrieves the list of required forms based on the student's selected program, application stage, and unique requirements, utilizing HubDB to track the necessary document types.
2. **Pre-Filled Data Integration:** Data retrieved from HubSpot’s CRM pre-fills the student’s application, ensuring consistency and reducing effort.
3. **Progress Tracking:** The system keeps track of the student’s progress, displaying pending and completed forms, and adjusting the application stage based on submitted documents.

4. **User-Friendly Interface:** The module provides clear visual cues on form status, allowing students to navigate easily between required forms, see completion status, and receive error messages or alerts for missing information.
5. **Conditional Logic:** The portal uses conditional logic to manage various scenarios, such as handling different stages of the application process, managing debug modes, and ensuring adaptability to program-specific requirements.

Pre-Acceptance

- Shown to students who have applied but not yet been accepted into a program.
- Contains required sections and fields specific to the application phase.

Post-Acceptance

- Displayed once a student has been accepted into the program.
- Includes any forms related to travel arrangements, financial aid, and additional program requirements.

Uploads

- Students can upload required documents in either PDF or JPG format.
- The system handles file compression and converts file types as needed.

Conditional Messaging

- Displays personalized messages based on the student's completion status and program requirements.

Form Functionality

Before Form Completion

- Students must choose their sections based on the program they are applying for.
- Options are filtered based on eligibility (only sessions available for the selected program are shown).
- Financial Aid forms display conditional messages:
    - A 'Yes' or 'No' option influences which fields become visible.
- Real-time data validation ensures correct formatting, particularly for fields such as Financial Aid information.

After Form Completion

- Shows answers for review.
- Redirects the student to the next incomplete form.
- If all pre-acceptance forms are complete, it prompts the next phase (e.g., payment or further requirements).

File Functionality

- **Before Submit**:
  - Converts files into PDF or JPG formats for consistency.
  - Compresses files if needed for optimized upload.
- **After Submit**:
  - Creates JSON data from the uploaded files for easy retrieval.
  - Files are stored securely and linked to the student's application.

Related Serverless Functions:

- `uploadFile.js`: Manages the file upload process.
- `protectData.js`: Ensures that sensitive information remains secure.
- `handleActions.js`: Handles actions triggered by form submissions.

Base Accommodations

The portal provides functions to manage base accommodations for students:

- **Handle Single Checkboxes**: Manages individual selections (e.g., dietary restrictions, special needs).
- **Handle Arrival Date/Time**: Captures data about when students will be arriving at their study abroad destination.

Payment Processing

The payment processing integrates seamlessly with the application process:

1. **Secure Payment Integration:** Uses HubSpot's serverless functions and external APIs to handle payment submissions, ensuring that transactions are secure and comply with data privacy standards.
2. **Deposit Tracking:** Tracks deposits for the program and automatically updates the application stage once payment is verified, providing real-time updates to the student.
3. **Payment Confirmation:** Once a payment is completed, the system sends automated confirmations and updates the student's portal status, advancing them to the next stage of their application process.
4. **Error Handling:** Incorporates robust error handling mechanisms to manage any payment failures, ensuring that students receive clear messaging and guidance on resolving any issues encountered during payment submission.

This enhanced version integrates the functionality described in the document to provide a comprehensive, secure, and user-friendly application and payment experience within the Student Portal.

When students are ready to confirm their application, they are redirected to a payment page:

- Payment completion is handled by the function `handlePaymentSuccess` which updates the student's status and marks the pre-acceptance requirements as finished.