Mobile Application

Introduction

The Mobile Application for Real-Time Management Information System (RTMIS) plays a pivotal role in facilitating remote data collection, primarily designed to support offline data submission for enumerators. Enumerators, who are an integral part of the data collection process, are assigned the responsibility of collecting critical information beyond the scope of Data Collectors. This mobile application serves as an indispensable tool, equipping enumerators with the means to efficiently gather data, even in areas with limited or no connectivity.

The Mobile Application for Real-Time Management Information System (RTMIS) is built upon a module derived from the National Management Information System (NMIS) Mobile Application (https://github.com/akvo/nmis-mobile). The NMIS Mobile App serves as a generic data collection tool designed to accommodate the needs of multiple services and organizations.

Within this context, the RTMIS Mobile Application takes center stage as a specialized module tailored to support the unique requirements of real-time data collection for management information. Specifically crafted to cater to the demands of the RTMIS, this mobile application empowers enumerators and data collectors with a targeted set of features and functionalities.

Requirements

Initial Setup

1. **Setup New Expo Application:**
   - Create a new Expo application as a foundation for the RTMIS Mobile App.
   - Configure the Expo environment with the necessary dependencies.
2. **Integration from nmis-mobile Repository:**
   - Copy the entire **app** folder from the **nmis-mobile** repository to the RTMIS repository.
   - Ensure that the integration includes all relevant code, assets, and configurations.
   - Make the necessary modifications to the module to align it with the specific requirements and functionalities of the RTMIS back-end.
3. **Docker Compose Setup for Development:**

- Implement Docker Compose setup to enable seamless development of the Mobile App within the RTMIS project.
- Integrate the Mobile App into the RTMIS development environment to ensure compatibility and ease of testing.

4. **Authentication Method Enhancement:**
   - Implement changes to introduce a new and improved authentication method for the RTMIS Mobile App.
   - Ensure that the new authentication method aligns with the security requirements and standards of the RTMIS project.
   - Update relevant documentation and user instructions to reflect the changes.
5. **CI/CD Setup for Mobile App Deployment:**
   - Establish a robust CI/CD pipeline for the RTMIS Mobile App, enabling automated deployment to the Expo platform.
   - Configure the pipeline to trigger builds and deployments based on code changes and updates to the Mobile App repository.
   - Ensure that the CI/CD setup includes proper testing and validation procedures before deploying to Expo
6. **Integration of Django Mobile Module:**
   - Incorporate the Django mobile module from the **National Wash MIS** repository folder: **v1_mobile** into the RTMIS back-end.

Overview

To support the integration of the mobile application, several critical updates are required for both the RTMIS platform's back-end and front-end components. These modifications encompass a range of functionalities designed to seamlessly accommodate the needs of the mobile application. Key updates will include, but are not limited to:

1. Back-end

1. **Authentication and Authorization API for Mobile Users:**
   - Integrate automated pass-code generation functionality to generate unique 6-digit alpha-numeric pass-codes for multiple mobile data collector assignment.
   - Establish an API mechanism to authenticate and authorize mobile users based on a pass-code. This ensures secure access to the RTMIS platform while simplifying user management for mobile data collectors.
2. **Form List and Cascade Retrieval API:**
   - Develop Cascade SQLite generator for both Entities and Administration.
   - Implement an API that enables the mobile application to retrieve forms and cascades from the RTMIS platform. This functionality is vital for data collection activities performed by enumerators and data collectors in the field.
3. **Data Monitoring API:**
   - Modify data/batch submission-related models and API to support monitoring submission.

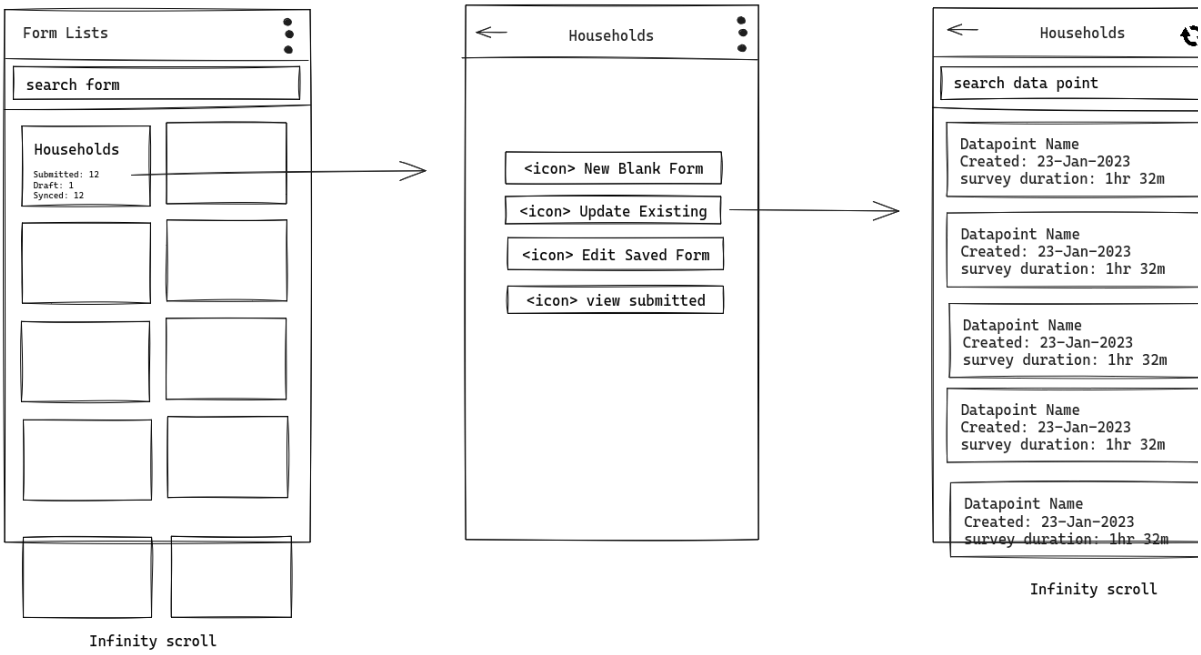
- Modify approval workflow-related models and API to support monitoring submission.

4. **Data Synchronisation API:**
   - Make the necessary modifications to the v1_mobile module to align it with the specific requirements and functionalities of the RTMIS back-end:
     - Preload existing data-points.
     - Modify Mobile Form submission-related models and API to support monitoring submission.
5. **Data Entry Staff Data Editing and Approval Workflow:**
   - Develop functionality for Data Entry Staff to add Mobile Assignments. The Data Entry Staff user can have multiple mobile assignments, which will require village ID and form ID. When a mobile assignment is created, it will generate a pass-code that will be used by Enumerators to collect data in the field via the Mobile App.
   - Develop functionality for Data Entry Staff to edit data submitted via the mobile application.
6. **Form Updates:**
   - Develop New Question Type: Data-point Question
   - New Question Parameters: Display Only

2. Front-end

1. **Dedicated "Mobile Data Collectors" Section:**
   - Create a dedicated section within the RTMIS front-end, labeled "Mobile Data Collectors," where Data Entry Staff can easily access and manage mobile data collector assignments.
2. **"Add Mobile Data Collector" Feature:**
   - Implement a user-friendly feature within the "Mobile Data Collectors" section that allows Data Entry Staff to initiate the process of adding mobile data collectors.
3. **Assignment Details Form:**
   - Develop a user-friendly form that Data Entry Staff can use to input assignment details:
     - the name of the assignment
     - Level (for scoping the administration selection)
     - Multiple Administration village selection
     - and form(s) selection.
   - Once the Data Entry Staff presses "create," the back-end will process it and return a 6-digit Alphanumeric code that will be used for mobile authentication.
4. **Communication of Pass-codes:**
   - Provide a mechanism within the front-end that allows Data Entry Staff to easily communicate the generated pass-codes to the respective mobile data collectors.
5. **User Guidance (RTD Updates):**
   - Include user guidance elements and feedback mechanisms in the front-end to assist Data Entry Staff throughout the process, ensuring that they understand the workflow and status of each assignment.

3. Mobile App

1. **Mobile App User Schema:**
   - Modify Authentication Method
2. **Mobile Database Modification**
   - Modify the Database Schema to support Monitoring and Cascade Sync Updates
   - Read more: **Mobile Database Modification**
3. **Mobile UI Modification**
   - Develop a screen where user can see and sync the list of existing data-points
   - Develop a screen where user can choose to add new data-point or edit existing data-point
     

   - Read more: **Mobile UI Modification**

Back-end

Back-end Database Migrations

Mobile Assignment Schema

1. Mobile Group (PENDING)

- Table name: **mobile_assignment_group**
- Model name: **MobileAssignmentGroup**
- Path to Model: **api.v1.v1_mobile.models**
- Migrations: **New Table**

pos	column	null	dtype	len	default

1	id	No	Integer	-	(Auto-increment)
3	name	No	Text	6	-
4	created_by				

2. Mobile Assignment Table

- Table name: **mobile_assignment**
- Model name: **MobileAssignment**
- Path to Model: **api.v1.v1_mobile.models**
- Migrations: **Alter Table**, add **name**

pos	column	null	dtype	len	default
1	id	No	Integer	-	(Auto-increment)
2	name	No	Text	255	-
3	passcode	No	Text	6	(Auto-generated)
4	token	No	Text	255	JWT String
5	created_by	No	Integer	-	(Primary Key)

Explanation: The `MobileAssignment` table stores information about mobile data collector assignments. The `id` column serves as the primary key and a unique identifier for each assignment. The `name` column holds the assignment's name or description, while the `passcode` column stores a unique pass-code for mobile data collector access.

3. Mobile Assignment Form Administration Table (Junction):

- Table name: **mobile_assignment_form_administration**
- Model name: **MobileAssignmentFormAdministration**
- Path to Model: **api.v1.v1_mobile.models**
- Migrations: **New Table**

pos	column	null	dtype	len	default
1	id	No	Integer	-	(Auto-increment)
2	assignment_id	No	Integer	-	-
3	form_id	No	Integer	-	-
4	administration_id	No	Integer	-	-

Explanation: This table serves as a junction table that establishes the many-to-many relationship between mobile assignments (`MobileAssignment`), forms (`form_id`), and administrative level (`administration_id`). The `id` column remains as the primary key, and the other columns associate

the rows with the respective assignment, form, and administrator.

Current Schema Updates

1. Data-point Table

- Table name: **data**
- Model name: **FormData** & **PendingFormData**
- Path to Mode; **api.v1.v1_data.models**
- Migrations: **Alter Tabl**e, add **uuid**

pos	table	column	null	dtype	len	default
1	data	id	NO	bigint		data_id_seq
2	data	form_id	NO	bigint		
3	data	administration_id	NO	bigint		
4	data	name	NO	text		
5	data	geo	YES	jsonb		
6	data	created	NO	datetime		
7	data	updated	YES	datetime		
8	data	created_by_id	NO	bigint		
9	data	updated_by_id	YES	bigint		
10	data	uuid	NO	uuid		uuid.uuid4

1. Question Table

- Table name: **question**
- Model name: **Questions**
- Path to Model: **api.v1.v1_forms.models**
- Migrations: **Alter Table,** add **fn, tooltip, display_only, meta_uuid,** and **monitoring**

pos	table	column	null	dtype	len	default
1	question	id	NO	bigint		question_id_seq
2	question	order	YES	bigint		

pos	table	column	null	dtype	len	default
3	question	text	NO	text		
4	question	name	NO	character varying	255	
5	question	type	NO	int		
6	question	meta	NO	bool		
7	question	required	NO	bool		
8	question	rule	YES	jsonb		
9	question	dependency	YES	jsonb		
10	question	form_id	NO	bigint		
11	question	question_group_id	NO	bigint		
12	question	api	YES	jsonb		
13	question	extra	YES	jsonb		
14	question	tooltip	YES	jsonb		
15	question	fn	YES	jsonb		
16	question	display_only	YES	bool		
17	question	meta_uuid	YES	bool		
18	question	disabled	YES	jsonb		
18	question	hidden	YES	jsonb		

2. Option Table

- Table name: **option**
- Model name: **QuestionOptions**
- Path to Model: **api.v1.v1_forms.models**
- Migrations: **Alter Table,** add **color**

pos	table	column	null	dtype	len	default

1	option	id	NO	bigint		option_id_seq
2	option	order	YES	bigint		
3	option	code	YES	character varying	255	
4	option	name	NO	text		
5	option	other	NO	bool		
6	option	question_id	NO	bigint		
7	option	color	YES	text		

API Endpoints

New Endpoints

1. Create Mobile Assignment

- Endpoint: **api/v1/mobile-assignment/<id>**
- Method: **POST / PUT**
- Authentication: **Bearer Token**
- Payload:

```
{
  "name": "Kelewo Community Center Health Survey",
  "administrations": [321,398],
  "forms": [1,2,4],
}
```

- Success Response (for POST request):

```
{
  "id": 1,
  "passcode":"4dadjyla",
}
```

- Explanation:
  - **id:** id of the assignment
  - **name**: represents the name of assignment (can be person name, community or organization).
  - **administrations**: list of **administration_ids** from **administration** table.
  - **forms**: list of forms for the mobile assignment
  - **passcode**: generated from **CustomPasscode** in **utils.custom_helper** via **MobileAssignmentManager**

2. Get List of Mobile Assignment

- Endpoint: **api/v1/mobile-assignment**
- Method: **GET**
- Authentication: **Bearer Token**
- Payload: **None**
- Success Response:


```
{
  "current":1,
  "total":11,
  "total_page":2,
  "data": [{
    "id":1,
    "name": "Kelewo Community",
    "passcode": "3a45562",
    "forms": [{
      "id":1,
      "name": "Health Facilities"
    },{
      "id":2,
      "name": "CLTS",
    },{
      "id":3,
      "name": "Wash In Schools"
    }],
    "administrations": [{
      "id":765,
      "name": "Kelewo"
    }]
  }]
}
```


Token Modifications

In the updated RTMIS Mobile application, a significant change is being introduced to enhance security and access control. This change involves modifying the token generation process for Mobile Data Collector Assignments. Here's a detailed description of this update:

1. Context and Need for Change

- **Previous System**: In the earlier version of the NMIS app, tokens were generated using **RefreshToken** from **rest_framework_simplejwt.tokens**. This approach was suitable

when the Mobile App users were Data Entry Users themselves. Previous token:

```
class MobileAssignmentManager(models.Manager):
    def create_assignment(self, user, name, passcode=None):
        token = RefreshToken.for_user(user)
        if not passcode:
            passcode = generate_random_string(8)
        mobile_assignment = self.create(
            user=user,
            name=name,
            token=token.access_token,
            passcode=CustomPasscode().encode(passcode),
        )
        return mobile_assignment
```

- **New Requirement**: With the introduction of Mobile Data Collector Assignments, there's a need to restrict token access to prevent unauthorized use of other endpoints.

2. Custom Token Generation for Enhanced Security

- **Custom Token Implementation**: The token generation process will be customized to create tokens that are specifically restricted in their access capabilities.
- **Restricted Access**: The custom token will only grant access to endpoints with the prefix **api/v1/mobile/device/***. This ensures that Mobile Data Collectors can access only the necessary data and functionalities relevant to their assignments.
- **Security Benefit**: This approach significantly enhances the security of the system by ensuring that each token can only interact with a limited set of endpoints, thereby reducing the risk of unauthorized access to sensitive data or functionalities.

3. Example Custom Token Generation

```
import jwt
from rtmis.settings import SECRET_KEY

def generate_assignment_jwt(assignment_id, allowed_forms_ids, administration_ids, secret_key):
    # Custom claim for Mobile Assignment
    custom_claim = {
        "assignment_id": assignment_id,
        "allowed_endpoints": "api/v1/mobile/device/*",
        "forms": allowed_forms_ids,
        "administrations": administration_ids
    }
    # Payload of the JWT without an expiration time
```

```
    payload = {
        "assignment": custom_claim
    }
    # Generate JWT token
    token = jwt.encode(payload, SECRET_KEY, algorithm="HS256")
    return token

# Example usage
secret_key = "your_secret_key"  # Secure, unguessable string
assignment_id = "assignment_123"  # Unique identifier for the mobile assignment
allowed_forms_ids = [101, 102, 103]  # Example list of allowed form IDs
administration_ids = [201, 202]  # Example list of allowed administration IDs

token = generate_assignment_jwt(assignment_id, allowed_forms_ids, administration_ids,
secret_key)
```

4. Token Payload

```
{
  "user_id": "<the user who create assignment>",
  "assignment_id": "<assignment_id>",
  "allowed_endpoints": "api/v1/mobile/device/*",
  "administration_ids": ["administration_id"],
  "allowed_forms_ids": ["form_id"],
  "exp": 1701468103,
  "iat": 1701424903,
  "jti": "923cfad9ff244e6897bfef2260dde4ee",
  ...other_stuff
}
```

5. Example Custom Authentication

```
from rest_framework.authentication import BaseAuthentication
from rest_framework import exceptions
import jwt

class MobileAppAuthentication(BaseAuthentication):
    def authenticate(self, request):
        # Retrieve the token from the request
        token = request.META.get('HTTP_AUTHORIZATION')
```

```
        if not token:
            return None  # Authentication did not succeed

        try:
            # Decode the token
            decoded_data = jwt.decode(token, 'your_secret_key', algorithms=["HS256"])

            # Check if the token has the required claims
            assignment_info = decoded_data.get('assignment')
            if not assignment_info:
                raise exceptions.AuthenticationFailed('Invalid token')

            # Add more checks here if needed (e.g., allowed_forms_ids, administration_ids)

            # You can return a custom user or any identifier here
            return (assignment_info, None)  # Authentication successful

        except jwt.ExpiredSignatureError:
            raise exceptions.AuthenticationFailed('Token expired')
        except jwt.DecodeError:
            raise exceptions.AuthenticationFailed('Token is invalid')
        except jwt.InvalidTokenError:
            raise exceptions.AuthenticationFailed('Invalid token')
```

6. Token Implementation Considerations

- **Token Scope**: The scope of the token is strictly limited to the specified API endpoints, ensuring that Mobile Data Collectors cannot access other parts of the system.
- **Compatibility**: The new token generation method should be compatible with the existing system's infrastructure and authentication mechanisms.
- **User Experience**: The change in token generation should be seamless to the users, with no negative impact on the user experience for legitimate access.

Endpoint Modifications

1. Get List of Assigned Forms

Unlike **nmis-mobile,** In the RTMIS Mobile application, the option to add users manually from the device will not be available (removed from the latest nmis-mobile). Consequently, when logging in, the response will now include information about the **assignmentName**. The remaining data will adhere to the existing structure of the previous Authentication API.

- Endpoint: **api/v1/device/auth**
- Method: **GET**
- Authentication: **None**
- Request Body:

```
{"code": "<assignment_code_provided_by_admin>"}
```

- New Response:


```
{
  "name": "Kelewo Community",
  "syncToken": "Bearer eyjtoken",
  "formsUrl": [
    {
      "id": 519630048,
      "url": "/forms/519630048",
      "version": "1.0.0"
    },
    {
      "id": 533560002,
      "url": "/forms/533560002",
      "version": "1.0.0"
    },
    {
      "id": 563350033,
      "url": "/forms/563350033",
      "version": "1.0.0"
    },
    {
      "id": 567490004,
      "url": "/forms/567490004",
      "version": "1.0.0"
    },
    {
      "id": 603050002,
      "url": "/forms/603050002",
      "version": "1.0.0"
    }
  ],
  "certifications": []
}
```

2. Get Individual Form

- Endpoint: **api/v1/device/form/<form_id>**
- Method: **GET**
- Authentication: **None**
- Authorization: **Bearer Token**

The Individual Form will be the same as the previous response endpoint, with the only change being in the schema of the cascade-type question as defined in the **Mobile Cascade Modification** section. In the previous cascade-type question, the `parent_id` was an integer, acting as the initial level cascade filter, so the first level of the cascade showed the children of the `parent_id`. Now, we support multiple `parent_id`s, so the first level of the cascade represents the `parent_id`s themselves.

Initial Result:

```
"source": {
  "file": "cascade-296940912-v2.sqlite",
  "parent_id": 273
},
```

Final Result:

```
"source": {
  "file": "cascade-296940912-v2.sqlite",
  "parent_id": [273,234]
},
```

Form Updates

New Question Type

Data-point Question

This new question type is similar to an option-type question, but instead of custom options created by the user, the options will be populated from the "data-point-name" field in the data table (refer to: https://wiki.cloud.akvo.org/books/rtmis/page/low-level-design#bkmrk-database-overviews).

Requirements:

- New API for Web-form which retrieve list of data-point based on the user token to filter the data-point list

- SQLite generation for the data-point list, the SQLite generation cycle will triggered when data is approved.
- File format for the SQLite: "/sqlite/<**form_id**>-<**administration_id**>-data.sqlite"

Parameters:

- Name: **type**
- Type: **Enum**
- Enum Name: **data_point**

New Question Parameter

Display Only

The "Display Only" parameter is a helper that can be used to display a question for which the answer should not be sent to the server. The "Display Only" parameter is used to assist users in running data calculations, dependency population, or auto-answering for other questions.

Example use case:

- Q1: Do you want to update or create new data?
  - When the answer is "yes," Q2 and Q3 appear.
  - When the answer is "no," Q2 and Q3 do not appear.

Requirements:

- The "Display Only" question parameter shall be defined as a feature in the survey/questionnaire creation tool.
- The primary purpose of the "Display Only" parameter is to allow the inclusion of questions in a survey for informational or display purposes only.
- The survey tool shall include appropriate error handling mechanisms to prevent "Display Only" questions from being treated as regular questions during data processing.
- This will not become a part of a bulk template, and data download

Parameters:

- Name: **displayOnly**
- Type: **Boolean**

Database Migration: Question

String Function

The latest version of the questionnaire introduces a new type of question, released in akvo-react-form v2.2.6, known as **autofield**. This question type necessitates a new parameter, with **fn** as the object name. To accommodate this, modifications to the database are required to store this new

parameter effectively.

Example use case:

```
{
  "id": 1701810579091,
  "name": "Outcome result - Functional toilet with privacy",
  "order": 4,
  "type": "autofield",
  "required": false,
  "meta": false,
  "fn": {
    "fnColor": {
      "G1": "#38A15A",
      "G0": "#DB3B3B"
    },
    "fnString": "function() {(#1699422286091.includes(\"G1\") &&
#1699423357200.includes(\"G1\") && #1699423571454.includes(\"G1\")) ? \"G1\" : \"G0\";}",
    "multiline": false
  }
}
```

- **Context:** The questionnaire includes three questions related to toilet facilities in a household, each with options categorized as "G0", "G0+", and "G1". The autofield question aims to provide an overall outcome based on responses to these questions.
- **Questions:**
  - **Household Toilet Observed** (Question ID: 1699422286091)
    - Options: "G0 No toilet" and "G1 Toilet observed"
    - Determines if a toilet facility is visible in the household.
  - **Functional Toilet** (Question ID: 1699423357200)
    - Options: "G0 Non-functional toilet", "G0+ Partly functional toilet", and "G1 Fully functional toilet"
    - Assesses the functionality of the toilet facility.
  - **Toilet Privacy** (Question ID: 1699423571454)
    - Options: "G0 No toilet privacy", "G0+ Inadequate toilet privacy", and "G1 Good toilet privacy"
    - Evaluates the privacy aspect of the toilet facility.
- **Autofield Question:**
  - **ID**: 1701810579091
  - **Type**: "autofield"
  - **Function (fnString)**: Evaluates the responses to the above questions and determines the overall outcome. The function checks if all three questions have a "G1" response. If so, the result is "G1"; otherwise, it defaults to "G0".

- **Use Case Scenario:**
  - A household is being surveyed for toilet facilities.
  - The enumerator observes that there is a toilet (G1 for Question 1699422286091), it is fully functional (G1 for Question 1699423357200), and it provides good privacy (G1 for Question 1699423571454).
  - The autofield function evaluates these responses and, since all are "G1", the overall outcome is "G1".
  - The autofield question then displays this result, using the color associated with "G1" (#38A15A - a shade of green) as defined in `fnColor`.
- **Outcome:** The autofield question effectively summarizes the overall status of the household's toilet facilities based on specific criteria, providing a quick and visually intuitive result. This helps in making informed decisions or assessments based on the survey data.

Requirements:

- **fnColor**: Maps result values to specific color codes in hex format. Each color must correspond to a potential result of the `fnString` function.
- **fnString**: A JavaScript function that evaluates conditions based on responses to other questions (identified by their question_id, referenced with a hashtag #) and returns a result.
- **multiline**: A boolean value indicating whether the result should be displayed in a single line (false) or multiple lines (true).
- **Integration with Questionnaire Logic**: The `fn` parameter must integrate with the overall questionnaire logic, dynamically evaluating and displaying results based on responses.
- **User Interface Display**: The result and its associated color, as defined in `fnColor`, should be clearly displayed in the questionnaire interface.
- **Validation and Error Handling**: Ensure `fnString` is a valid function and `fnColor` contains valid color codes. The system should handle errors effectively if the function fails or returns an undefined color code.

Parameters:

- Name: **fn**
- Type: **Object**

Database Migration: Question

Meta UUID

The "Meta UUID" parameter is a useful utility that generates a universally unique identifier (UUID) for each data point, allowing you to easily track and distinguish individual records within your dataset. This unique identifier can be used as a parent datapoint when performing data monitoring, grade claims, and certification

Example use case:

```
{
  "id": 1702914803732,
  "order": 4,
  "name": "hh_code",
  "label": "Household Code",
  "type": "text",
  "required": true,
  "meta": false,
  "meta_uuid": true
}
```

Requirements:

- The "Meta UUID" question parameter shall be defined as a feature in the survey/questionnaire creation tool.
- The "Meta UUID" UUID allows for efficient lookup, linking, and querying of specific datapoints, ensuring that identical data records can be uniquely identified and managed.

Parameters:

- Name: **meta_uuid**
- Type: **Boolean**

Database Migration: Question

Hidden

Example use case:

```
{
  "id": 1716283800,
  "order": 34,
  "name": "community_outcomes_achieved",
  "label": "Have all of the community outcomes for this grade been achieved?",
  "type": "option",
  "required": true,
  "meta": false,
  "options": [
    {
      "order": 1,
      "label": "Yes",
```

```
      "value": "yes",
      "color": "green"
    },
    {
      "order": 2,
      "label": "No",
      "value": "no",
      "color": "red"
    }
  ],
  "hidden": {
    "submission_type": ["registration", "monitoring", "certification"]
  }
}
```

Parameters:

- Name: **hidden**
- Type: **Object**

Database Migration: Question

Disabled

Example use case:

```
{
  "id": 1699354849382,
  "order": 2,
  "name": "hh_location",
  "label": "What is the location of the household?",
  "short_label": null,
  "type": "administration",
  "required": true,
  "meta": false,
  "fn": null,
  "disabled": {
    "submission_type": ["monitoring", "verification", "certification"]
  }
}
```

Parameters:

- Name: **disabled**
- Type: **Object**

Database Migration: Question

Default value

Example use case:


```
{
  "id": 1699354220734,
  "order": 1,
  "name": "reg_or_update",
  "label": "New household registration or Monitoring update?",
  "type": "option",
  "required": true,
  "meta": false,
  "options": [
    {
      "order": 1,
      "label": "New",
      "value": "new"
    },
    {
      "order": 2,
      "label": "Update",
      "value": "update"
    }
  ],
  "default_value": {
    "submission_type": {
      "monitoring": "update",
      "registration": "new",
    }
  },
  "dependency": null,
  "fn": null
}
```

Parameters:

- Name: **default_value**
- Type: **Object**

Database Migration: Question

Pre-filled

Example use case:

```
{
  "id": 1699417958748,
  "order": 1,
  "name": "resp_position",
  "label": "Household respondent position in household",
  "type": "option",
  "required": true,
  "meta": false,
  "options": [
    {
      "order": 1,
      "label": "Household head",
      "value": "hh_head"
    },
    {
      "order": 2,
      "label": "Spouse of household head",
      "value": "spouse_of_hh_head"
    },
    {
      "order": 3,
      "label": "Parent of household head",
      "value": "parent_of_hh_head"
    }
  ],
  "pre": {
    "reg_or_update": {
      "new": ["hh_head"]
    }
  }
```

```
}
```

Parameters:

- Name: **pre**
- Type: **Object**

Database Migration: Question

New Option Parameter

Option Color

Additionally, new functionalities have been introduced to enhance the visual appeal of options in **option** and **multiple_option** types of questions by incorporating color. To support this feature, a new column named **color** needs to be migrated into the **option** table.

Database Migration: Option

Front-end

User Stories

1. Adding an Assignment

Step 1: Access the "Mobile Data Collectors" Section

- **Action**: Navigate to the dedicated "Mobile Data Collectors" section within the RTMIS front-end.
- **Purpose**: This section is specifically designed for managing mobile data collector assignments.

Step 2: Initiate Adding a Mobile Data Collector

- **Action**: Use the "Add Mobile Data Collector" feature available in this section.
- **Purpose**: This feature allows the Data Entry Staff to start the process of creating a new assignment for mobile data collectors.

Step 3: Fill in the Assignment Details Form

- **Action**: Complete the user-friendly form provided for assignment details.
- **Details to Include**:
  - **Name of the Assignment**: Provide a descriptive name or title for the assignment.
  - **Level:** Choose level for Mobile Assignment (not for sending to back-end)

- **Administration Selection**: Choose the relevant administrative area for the assignment (one or multiple).
- **Form(s) Selection**: Select the specific form(s) that the mobile data collector will use for data collection.

Step 4: Create the Assignment

- **Action**: After filling in all the necessary details, click the "create" button.
- **Backend Processing**: On clicking "create," the RTMIS backend processes the provided information.

Step 5: Receive the Assignment Pass-code

- **Outcome**: Once the backend processing is complete, a unique 6-digit alphanumeric code is generated.
- **Purpose**: This pass-code is used for mobile authentication by the enumerators or data collectors in the field.
- **Note:** When Data Entry Staff add a new assignment for Mobile Data Collectors in the RTMIS system, it's important to note the following:
  - **Informing the Enumerator**: The Data Entry Staff who adds Mobile Data Collectors should personally inform the Enumerator about the assignment. This communication is typically done during a training session or a designated briefing.
  - **Pass-code Availability**: The unique 6-digit alphanumeric pass-code generated for each assignment will also be displayed in the Mobile User list within the RTMIS system.
  - **Responsibility of Communication**: It is the responsibility of the Data Entry Staff to ensure that Enumerators are aware of and understand the pass-code and its usage.

2. Submitting a Pending Batch of Data

Step 1: Data Collection by Mobile Data Collector/Enumerator

- **Action**: As a Mobile Data Collector/Enumerator, I collect data in the field using the RTMIS mobile application.
- **Outcome**: After data collection, I submit the data. The data is uploaded and appears as a pending submission.

Step 2: Pending Submission Review by Data Entry User

- **Action**: As a Data Entry User, I review the pending submissions that have come in from various Mobile Data Collectors/Enumerators.
- **Visibility**: The submissions are clearly marked as pending and are queued for batch processing.

Step 3: Batch Creation for Submission

- **Action**: I create a batch of the pending data for submission.

- **Details**: While creating the batch, I ensure that the name of the submitter (Mobile Data Collector/Enumerator) is recorded for each data entry. This is a new feature in the updated RTMIS system.

Step 4: Data Submission

- **Action**: I submit the batch of data for processing.
- **New Feature**: Unlike the previous system, the RTMIS now records the name of the actual submitter (Mobile Data Collector/Enumerator) rather than the Data Entry User.

Step 5: Data Approval Process (Unchanged):

- **Note**: The rest of the data approval process remains unchanged. The submitted data undergoes the usual verification and approval workflow as per the existing RTMIS protocols.

Mobile

User Stories

1. User Authentication

1.a. When there's no user in the users database:

- Open App
- Login with the user pass-code
- Store token (response from server) to **users** table and state
- Fill the information about the user in **users** database from the server response. Unlike the previous version, in this version, the logged in user CANNOT fill the user information themselves.
- Form list opened

1.b. When user is available in the users database:

- Open App
- User selection page opened: on the bottom of the page, there should be a button for adding new user
- User click add new user
- Login with the user pass-code
- Store token (response from server) to **users** table and state
- Fill the information about the user in **users** database from the server response. Unlike the previous version, in this version, the logged in user CANNOT fill the user information themselves.
- Form list opened

2. Download Data-points (for monitoring)

- Open App
- User selection page opened
- Select the user from user list
- Press download data
- Server will give the list of data-points which can be downloaded
  - ```
    [{
      "id": 1,
      "updated_at": 1701070914356
    },{
      "id": 2,
      "updated_at": 1701070914356
    }]
    ```

- Mobile download the data-points 1 by 1 (queue) and store it to **datapoints** database
  - Before download, check if the **datapointId** is exist in the **datapoints** database
  - And compare:
    - If **updated_at** > **createdAt** (in **datapoints** table): Replace the datapoint
    - If **updated_at** < **createdAt** (in **datapoints** table): Don't download
  - User will get notified when:
    - server send error response
    - download is finished

Mobile Database Modifications

1. Form Database

Table name: **forms**

Column Name	Type	Example
id	INTEGER (PRIMARY KEY)	1
userId	INTEGER	1
formId	INTEGER	453743523
version	VARCHAR(255)	"1.0.1"
latest	TINYINT	1
name	VARCHAR(255)	'Household'
json	TEXT	See: Example JSON Form
createdAt	DATETIME	`new Date().toISOString()`

Changes:
    ```

- Add **userId** column to (from **users** database), so every form has owner.

2. User Database

Table name: **users**

Column Name	Type	Example
id	INTEGER (PRIMARY KEY)	1
active	TINYINT	1 (default: 0)
name	INTEGER	1
password	TEXT	crypto
token	TEXT	token
certifications	TEXT	jsonb (administration)
lastSyncedAt	DATETIME	`new Date().toISOString()`

Changes:

- Add **token** column to store (token from authentication response)
- Add **certifications** column table to store the certification assignments for users to complete the Grade Certification form.
- Add **lastSycedAt** column table to store the timestamp of the user's last sync.

3. Form Submission / Datapoints Database

Table name: **datapoints**

Column Name	Type	Example
id	INTEGER (PRIMARY KEY)	1
form	INTEGER	1 (represent **id** in **forms** table, NOT formId)
user	INTEGER	1 (represent **id** in **users** table)
submitter	TEXT	'John'
name	VARCHAR(255)	'John - St. Maria School - 0816735922'

Column Name	Type	Example
submitted	TINYINT	1
duration	REAL	45.5 (in Minutes)
createdAt	DATETIME	`new Date().toISOString()`
submittedAt	DATETIME	`new Date().toISOString()`
syncedAt	DATETIME	`new Date().toISOString()`
json	TEXT	`'{"question_id": "value"}'`
submission_type	INTEGER	1 (represents the enum value of the submission type i.e. **registration**)
uuid	VARCHAR(191)	`Crypto.randomUUID()`

Changes:

- **user** should be NULLABLE when form submission data is synced from RTMIS database sync
- Add **submitter** column
- Add **submission_type** column
- Add **uuid** column

Mobile Cascade Modification

The updated Mobile App Development introduces a significant change in handling cascade drop-down options, particularly in how multiple **parent_ids** are managed. This change affects the way options are displayed and selected in the cascade type of questions. Here's a detailed explanation of the new functionality:

Updated Functionality

Previous Functionality

- **Single parent_id**: Initially, the cascade drop-down supported only a single parent_id.
- **Children Display**: The parent_id would query the SQLite database to display its children levels as options in the cascade drop-down.

Example:

```
"source": {
  "file": "cascade-296940912-v2.sqlite",
  "parent_id": 273
},
```

Updated Functionality with Multiple Parent Ids

- **Array of parent_ids**: The new system supports an array of parent_ids, allowing for more complex cascade structures.
- **First Cascade Level**: The **parent_id** array itself becomes the first level of the cascade to select from.

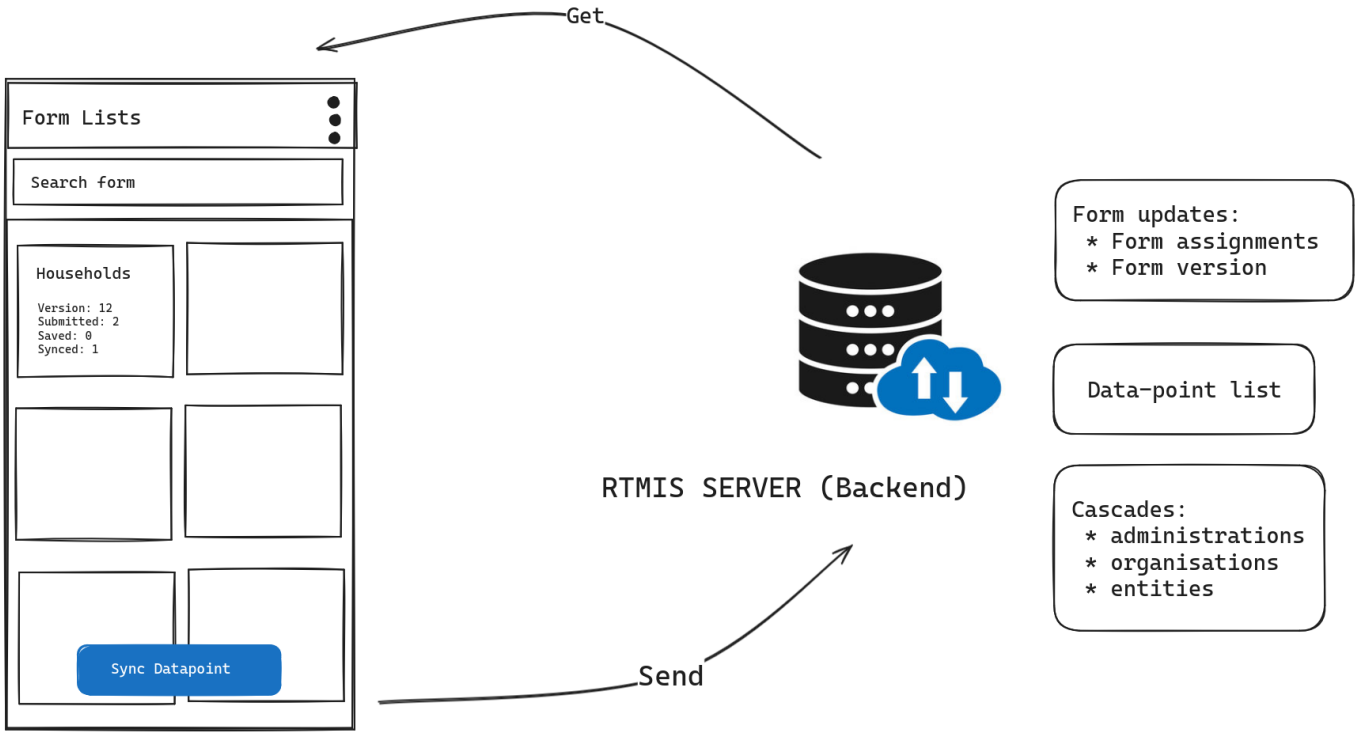
Example:

```
"source": {
  "file": "cascade-296940912-v2.sqlite",
  "parent_id": [273]
},
```

Handling Different Scenarios

1. **Single parent_id in Array**:
   - If the parent_id array contains only one **administration_id**, the first cascade option should automatically display the children of this single `parent_id`.
   - **Example**: "parent_id": `[273]` would directly show the children of `273` as the cascade options.
2. **Multiple parent_ids in Array**:
   - If the parent_id array contains multiple administration_ids, the first cascade level will allow selection among these `parent_id`s.
   - **Example**: `"parent_id": [273, 123]` means the first cascade level will have options to select either `273` or `123`.
3. **Single `parent_id` Without Children**:
   - In a scenario where the `parent_id` array has one `administration_id` and this `administration` does not have any children, the app should automatically select this `parent_id` as the value by default.
   - **Example**: If `273` has no children, it becomes the default selected value

Data Synchronization



Mobile App - Home screen

To ensure that the mobile app is up-to-date with the latest information from the server, users can synchronize data points with a simple process. This ensures that all forms, data points, and master data are current and accurate.

Syncing Data Points

Step-by-Step Process:

1. **Initiate Sync**: The mobile user can easily initiate the synchronization process by clicking the "Sync Datapoint" button on the Mobile app's Home screen.
2. **Request to Backend**: When the user clicks "Sync Datapoint", the app sends a request to the backend server to retrieve three main categories of data:
   - **Form Updates**: Retrieves the current form assignments for the mobile user, including any updates indicated by form versions. This ensures the user is aware of any changes made to the forms they use.
   - **Data-point List**: Obtains the latest routine data based on the mobile user's form assignments. This includes all relevant and recent data points necessary for the user's tasks.
   - **Cascades**: Retrieves the latest master data, such as administration details, organization information, and entity lists. This data is critical for aligning the app with real-world conditions and reflecting any additions, updates, or removals.
3. **Completion of Sync Process**: Once the synchronization process is complete, the mobile user can access the updated data. They can then navigate to the desired form with all the latest information available.

Data-point List API

Here is the following JSON response from data-point list API:


```
{
  "current": 1,
  "total": 7,
  "total_page": 1,
  "data": [
    {
      "id": 11,
      "form_id": 1699353915355,
      "name": "DATA #1",
      "administration_id": 57443,
      "url": "https://rtmis.akvotest.org/b4b00592-b949-4424-b4ba-448a0d410ecf.json",
      "last_updated": "2024-05-30T04:31:58.539349Z"
    }
  ]
}
```


> The **url** field in the **data** array will contain a URL to the JSON file that the mobile app will download as a **data-point**. This JSON URL is a direct link to a static file and is not generated by the back-end API, allowing for high traffic downloads.

Data-point JSON

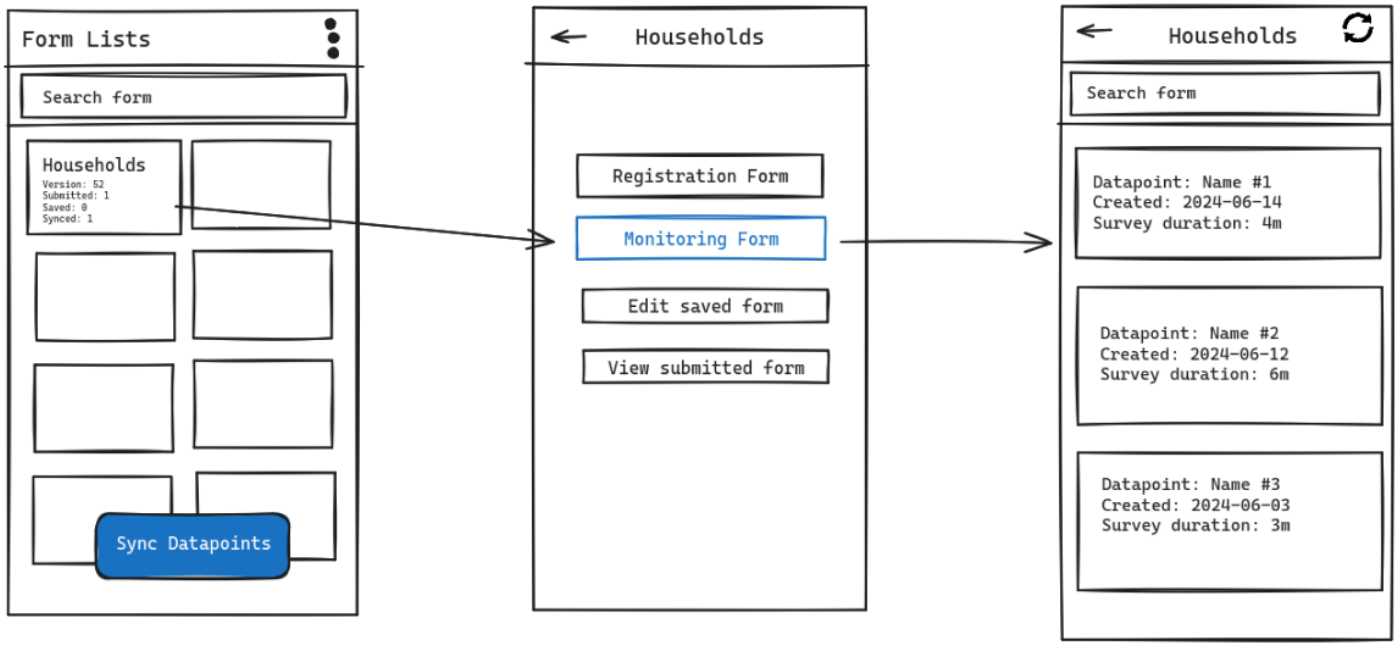
After obtaining all the JSON URLs asynchronously, the mobile app will fetch the following JSON schema and store it in the mobile database:


```
{
  "id": 21,
  "datapoint_name": "Testing Data County",
  "submission_type": 1,
  "administration": 2,
  "uuid": "025b218b-d80a-454f-8d69-8eef812edc82",
  "geolocation": [
    6.2088,
    106.8456
  ],
  "answers": {
    "101": "Jane",
```

```
    "102": [
      "Male"
    ],
    "103": 31208200175,
    "104": 2,
    "105": [
      6.2088,
      106.8456
    ],
    "106": [
      "Parent",
      "Children"
    ],
    "109": 2.5
  }
}
```

By following this process, mobile users can maintain a high level of productivity and accuracy in their tasks, leveraging the most current data available from the server.

Monitoring Support



In this version of RTMIS mobile, we introduce monitoring support for data-points. This monitoring is similar to a normal submission but includes previous answers. The form's shape will depend on the **submission_type** equal to 2 (enum value for monitoring) in the question-level object. Users will only answer questions that have a monitoring flag in the question. When synced to the server, it will be treated as the same data-point, except they will have the same meta

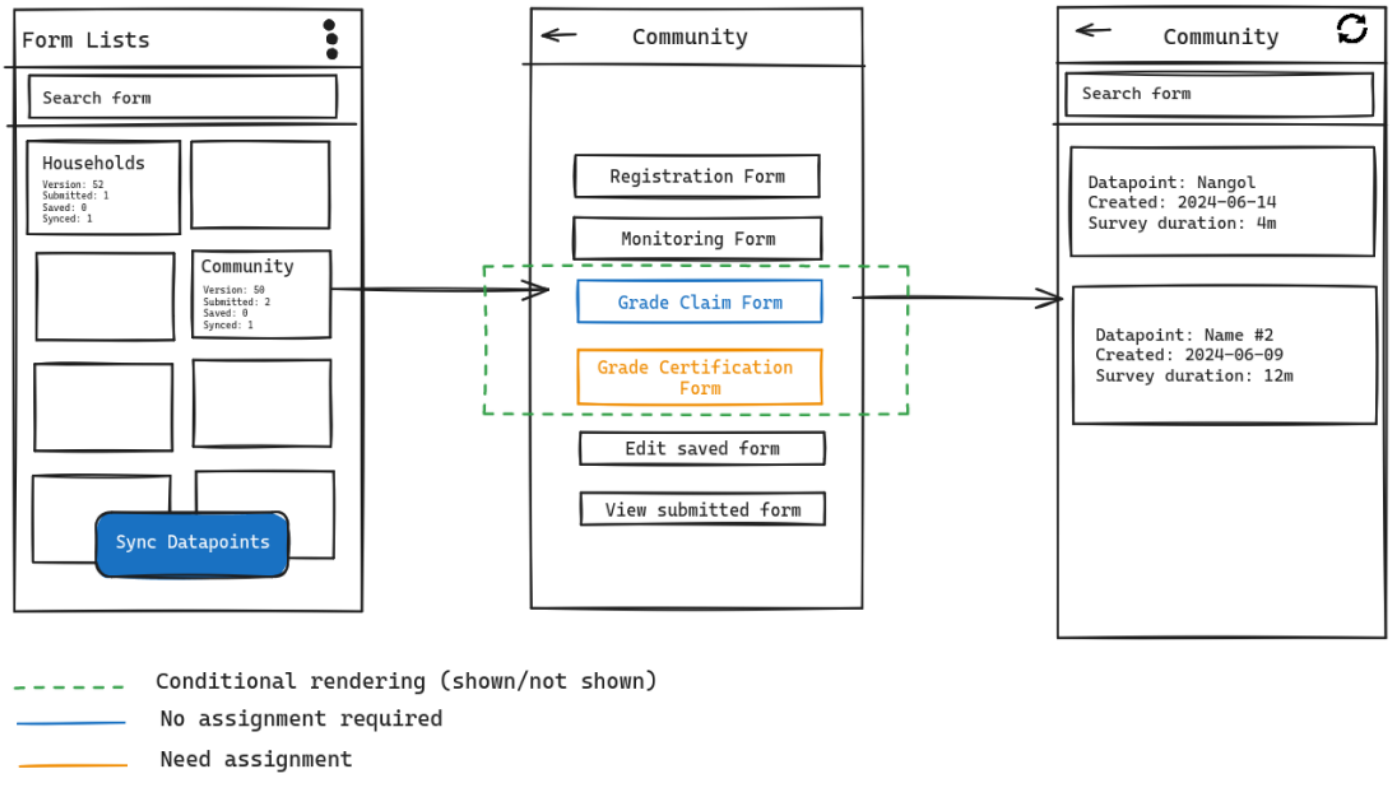
UUID as their parent data-point.

Storing the Monitoring Data-point

The following table represents the schema for storing monitoring data-points:

Column Name	Type	Example
id	INTEGER (PRIMARY KEY)	1
formId	INTEGER	1 (represent **id** in **forms** table, NOT formId)
name	VARCHAR(255)	'Testing Data County'
administrationId	TINYINT	1
uuid	VARCHAR(255)	025b218b-d80a-454f-8d69-8eef812edc82
syncedAt	DATETIME	`new Date().toISOString()`
json	TEXT	`'{"question_id": "value"}'`

Grade Claim Support



The Grade Claim feature within the mobile app is designed to streamline the verification and certification of grades. Below is a detailed description of how this feature operates and its dependencies.

Overview

The Grade Claim feature has two submission types:

1. **Verification**: Utilized through the Grade Claim form.
2. **Certification**: Utilized through the Grade Certification form.

Example Form Configuration

Feature Dependencies and Behavior

- **Submission Type Dependency**:
  - The availability of the Grade Claim feature is dependent on the submission type definitions at the form level.
  - If `verification` or `certification` submission types are defined in the form, the corresponding button will appear on the mobile app's Manage Form screen.
- **Approval Process**:
  - Neither the Grade Claim form nor the Grade Certification form requires an approval process, simplifying the workflow for users.
- **Certification Assignment Requirement**:
  - The Grade Certification process requires a certification assignment, which is managed by sub-county users via the dashboard.
  - If a `certification` submission type exists but the mobile user does not have an assignment, the certification button will not be displayed in the app.
- **UUID Requirement**:
  - The Grade Claim feature also requires a UUID to link the grade claim or certification to the parent data-point. This ensures accurate data tracking and association.

How to Use the Grade Claim Feature

1. **Initiate Grade Claim**:
   - Navigate to the Manage Form screen in the mobile app.
   - If `verification` or `certification` submission types are available, the respective buttons will be visible.
2. **Complete the Form**:
   - Select the appropriate form (Grade Claim or Grade Certification) based on the submission type.
   - Fill out the necessary information and submit the form.
3. **No Approval Needed**:
   - Once submitted, the forms do not require an approval process, allowing for immediate processing.
4. **Certification Assignments**:

- Ensure that certification assignments are managed via the dashboard by sub-county users to enable the certification feature on the mobile app.

5. **UUID Linking**:
   - Ensure that each submission is linked with the parent data-point using the provided UUID to maintain data integrity.

By following this documentation, users can effectively utilize the Grade Claim feature, ensuring a smooth and efficient workflow for verifying and certifying grades.

Revision #68
Created 2023-11-20 05:56:33 UTC by Deden Bangkit
Updated 2024-06-14 08:42:08 UTC by Iwan Firmawan