# Django API for Remote Data Collection Using Twilio for WhatsApp

This design document outlines the architecture and features of a new Django API app that enables remote data collection using Twilio for WhatsApp.

# Introduction

## Overview

This document outlines the design for a new **Django API app** that will be used to collect remote data using **Twilio**. The purpose of this app is to provide a convenient way for users to collect data from remote locations using **WhatsApp**, which is a popular messaging app used by millions of people around the world. The new Django API app will use Twilio's API to integrate with WhatsApp and collect data from remote locations.

## Architecture

The app can be attached to any projects that are using Django Framework including a remote data collection module.

## Modules

### Form Module

The form data management module will be responsible for storing and send information about the questionnaire which should be align with Twilio's API for WhatsApp.

### Data Module

The data module will be responsible for collecting data from remote locations. The module will be designed to handle different types of data, including text, photos, and Geo-points. The module will also be designed to handle administration using Google Maps API.

## User Interface

The user interface will provide a simple table interface with expandable details for admin to manage data.

# Use Cases

> In the first phase the app will be used by our partners to collect data from remote locations, such as water point complains for **National WASH MIS** and **RTMIS**.

The new Django API app for remote data collection using Twilio for WhatsApp can be used in various scenarios. Some potential use cases for the app are:

## Environmental Monitoring

The app can be used by environmental organizations to collect data from remote locations, such as forests, rivers, and oceans. The app can collect data on water quality, air quality, and biodiversity, among others.

## Field Research

The app can be used by researchers to collect data in the field, such as wildlife sightings, vegetation surveys, and soil samples. The app can collect data from multiple sources, including sensors and cameras.

## Disaster Response

The app can be used by emergency response teams to collect data in disaster-stricken areas, such as earthquake zones and flood-prone areas. The app can collect data on the extent of damage, the number of people affected, and the condition of infrastructure.

## Agricultural Monitoring

The app can be used by farmers to collect data on their crops, such as growth rates, yields, and disease outbreaks. The app can collect data from sensors and cameras installed in fields and greenhouses.

## Wildlife Conservation

The app can be used by conservation organizations to collect data on endangered species, such as population sizes, migration patterns, and habitat preferences.

Overall, the app can be used in various contexts where remote data collection is required. The app is designed to be flexible and adaptable, allowing it to be customized to suit different use cases.

# Requirements

The new Django API app for remote data collection using Twilio for WhatsApp has various functional and non-functional requirements. These requirements are outlined below.

## Functional Requirements

1. It should be able to collect data from remote locations using Twilio's API for WhatsApp.
2. It should be able to handle various types of data, including text, images, and geo-location.
3. It should be able to store detail information about geo-location using Google Maps API.
4. It should be able to store the collected data in a database.
5. It should be able to provide an API for external systems (e.g front-end webpage) to access the collected data.
6. It should be able to perform data validation to ensure that the incoming data from individual session is correct and complete.
7. It should be able to provide feedback to users when errors occur.

## Non-functional Requirements

1. It should be reliable, with a robust and reliable architecture that can handle errors and failures.
2. It should be accessible, with a user interface.
3. It should be documented thoroughly, with technical manuals, and API documentation.
4. It should undergo various testing stages, including unit testing, and integration testing.
5. The API Module should be modular and designed to be reusable, with separate and independent modules, each with its own functionality and purpose.

The functional requirements ensure that the app can collect, store, and manage data efficiently, while the non-functional requirements ensure that the app is secure, reliable, and accessible.

# Technical Details

| | |
|---|---|
| Package Name | **AkvoDjangoFormGateway** |
| Source Code | https://github.com/akvo/Akvo-DjangoFormGateway |
| Coverage | https://coveralls.io/github/akvo/Akvo-DjangoFormGateway |
| Documentation | - |
| Database Schema | https://dbdocs.io/dedenbangkit/akvo-django-form-gateway |
| Issues | https://github.com/akvo/Akvo-DjangoFormGateway/issues |

# Data Module

The data module of the new Django API app for remote data collection using Twilio for WhatsApp includes four main models: Form, Data, Question, and Answer. These models are used to collect, store, and manage the collected data.

## Form Model

The Form model is used to define the structure of the data that will be collected. It includes information such as the name of the form, a description of the form, and the fields that will be included in the form.

```
class AkvoGatewayForm(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField(null=True, default=None)
    version = models.IntegerField(default=1)

    def __str__(self):
        return self.name

    class Meta:
        db_table = "ag_form"
```

## Question Model

The children of the Form model is Question model. Each Question includes fields for the ID of the form that the question belongs to, the name of the question, and the type of the question. The type field is defined using the Question Type enumeration, which includes options such as text, number, photo, geo-location, option and multiple-option.

```
class QuestionTypes:
    geo = 1
    text = 2
    number = 3
    option = 4
    multiple_option = 5
    photo = 6
    date = 7

    FieldStr = {
        geo: 'Geo',
        text: 'Text',
        number: 'Number',
        option: 'Option',
        multiple_option: 'Multiple_Option',
        photo: 'Photo',
        date: 'Date',
    }

class AkvoGatewayQuestion(models.Model):
    form = models.ForeignKey(
        to=AkvoGatewayForm,
        on_delete=models.CASCADE,
        related_name="ag_form_questions",
    )
    order = models.BigIntegerField(null=True, default=None)
    text = models.TextField()
    type = models.IntegerField(choices=QuestionTypes.FieldStr.items())
    required = models.BooleanField(null=True, default=True)

    def __str__(self):
        return self.text

    class Meta:
        db_table = "ag_question"
```

```
class AkvoGatewayQuestionOption(models.Model):
    question = models.ForeignKey(
        to=AkvoGatewayQuestion,
        on_delete=models.CASCADE,
        related_name="ag_question_question_options",
    )
    order = models.BigIntegerField(null=True, default=None)
    code = models.CharField(max_length=255, default=None, null=True)
    name = models.TextField()

    def __str__(self):
        return self.name

    class Meta:
        db_table = "ag_option"
```

# Data Model

The Data model is used to store the collected data. It includes fields for the ID of the form that the data belongs to, the name of the data, and the submitter (phone number), as well as metadata such as the date and time that the data was collected and the location of the data collection.

```
class StatusTypes:
    draft = 1
    submitted = 2

class AkvoGatewayData(models.Model):
    name = models.TextField()
    form = models.ForeignKey(
        to=AkvoGatewayForm,
        on_delete=models.CASCADE,
        related_name="ag_form_data",
    )
    geo = models.JSONField(null=True, default=None)
    phone = models.CharField(max_length=25)
    status = models.IntegerField(default=StatusTypes.draft)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(default=None, null=True)

    def __str__(self):
```

```
        return self.name

    class Meta:
        ordering = ["id"]
        db_table = "ag_data"
```

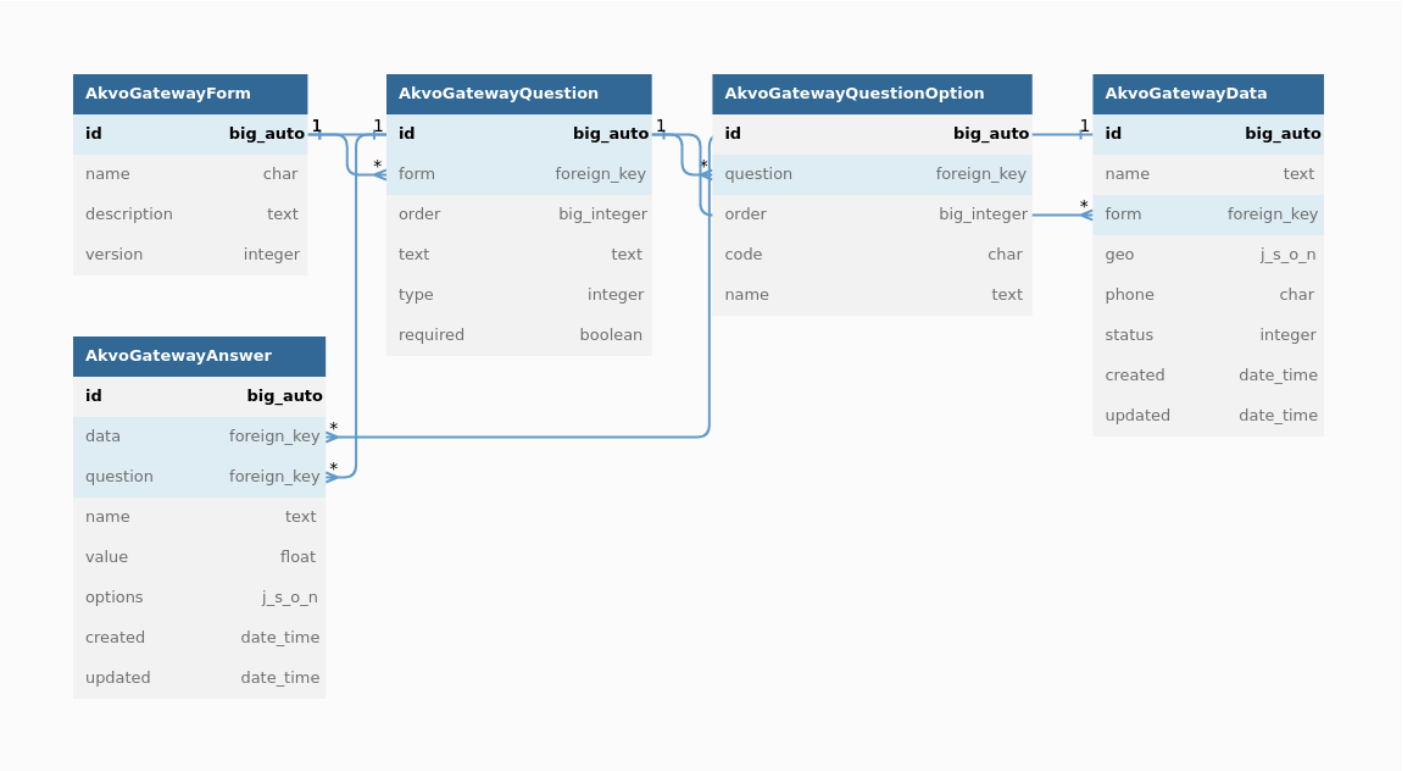
## Answer Model

The Answer model is used to store the answers to the questions in the Form model. It includes fields for the ID of the answer, the ID of the form data, the ID of the question, and the answer itself. The Answer model is also use to store the status of the submitter session. When the submitter completes all the question in the session, the data model should marked as done.

```
class AkvoGatewayAnswer(models.Model):
    data = models.ForeignKey(
        to=AkvoGatewayData,
        on_delete=models.CASCADE,
        related_name="ag_data_answer",
    )
    question = models.ForeignKey(
        to=AkvoGatewayQuestion,
        on_delete=models.CASCADE,
        related_name="ag_question_answer",
    )
    name = models.TextField(null=True, default=None)
    value = models.FloatField(null=True, default=None)
    options = models.JSONField(default=None, null=True)

    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(default=None, null=True)

    def __str__(self):
        return self.data.name

    class Meta:
        db_table = "ag_answer"
```

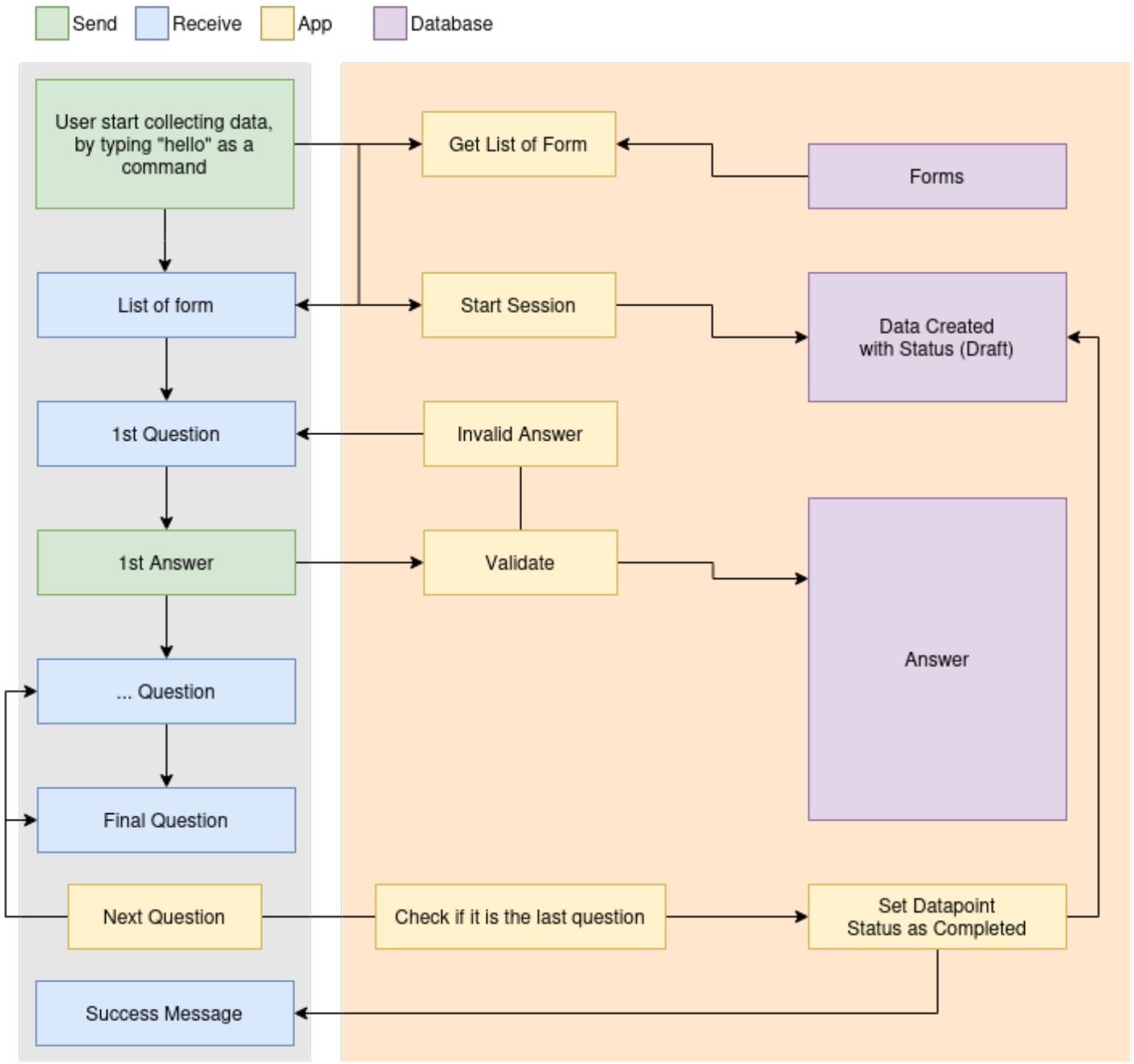
# Database Schema

Overall here is the database schema from the Data Models




# Workflow

The workflow image below illustrates the process.




1. At first the Django API APP receives incoming messages and checks if the message contains pre-configured parameter which linked to a particular form, e.g when user type "*complain*"; there will be a form in which will then be initiated for the user. This also initiates a data record and sets status as draft
2. Retrieve the questions for the selected form and start asking the user the questions one by one. Once the user answers a question, validate the answer. If the answer is invalid, notify the user that their answer was wrong and prompt them to answer again. If the answer is valid, store the answer in the **Answer model** and move on to the next question.
3. Once all questions have been answered, the **draft** status in the **Data record** changed to **completed**.Still Draft