# Form Builder Application

## Project Overview

The Form Builder Application is a Django-based project that allows users to create forms, collect responses, and view detailed analytics on those responses. The application supports different types of questions, including text inputs, dropdowns, and checkboxes, with features for managing forms, questions, and responses. Analytics provide insights into the responses, such as common answers and trends.

---

## Installation Steps

To set up and run the Form Builder Application locally, follow the steps below:

### Prerequisites

- **Python** (Version 3.8 or higher)
- **Django** (Version 4.0 or higher)
- **Django Rest Framework** (Version 3.12 or higher)
- **SQLite** (Used by default as the database)

### Step 1: Clone the Repository

Clone the repository from your source control (e.g., GitHub, GitLab):

```
git clone https://github.com/yourusername/form-builder.git
cd form-builder
```

Continue with the next steps in the original instructions to complete the installation:

- Set up a virtual environment.
- Install dependencies.
- Apply migrations.
- Create a superuser.
- Start the development server.

Refer to the complete installation steps to ensure successful configuration.

### Step 2: Create a Virtual Environment (Optional but Recommended)

Create a virtual environment to isolate dependencies:

```
python -m venv venv
```

Activate the virtual environment:

- For Windows:

  ```
  venv\Scripts\activate
  ```

- For macOS/Linux:

  ```
  source venv/bin/activate
  ```

### Step 3: Install Dependencies

Install the required dependencies using `pip`:

```
pip install -r requirements.txt
```

### Step 4: Apply Migrations

Apply the migrations to create the necessary database tables:

```
python manage.py migrate
```

### Step 5: Create a Superuser (Admin User)

Create a superuser to access the Django admin interface:

```
python manage.py createsuperuser
```

Follow the prompts to enter your username, email, and password.

### Step 6: Run the Development Server

Start the Django development server:

```
python manage.py runserver
```

This will start the server at `http://127.0.0.1:8000/`. You can access the admin panel by navigating to `http://127.0.0.1:8000/admin/`.

---

## Models Overview

### 1. **Form Model**

The **Form** model represents a form that can contain multiple questions. It includes:

- **title**: A brief title for the form.
- **description**: An optional description field to provide more context.
- **created_at**: Timestamp indicating when the form was created.

**Validation**: The `clean()` method ensures that a form can have no more than 100 questions. This validation is enforced before saving the form to the database.

### 2. **Question Model**

The **Question** model represents individual questions within a form. It supports three types of questions:

- **Text**: A simple text input.
- **Dropdown**: A list of options for the user to select.
- **Checkbox**: A set of multiple checkboxes for selection.

Each question includes:

- **question_text**: The question text itself.
- **question_type**: The type of the question (Text, Dropdown, or Checkbox).
- **options**: A JSON field to store options for Dropdown or Checkbox questions.
- **order**: Defines the position of the question in the form.

### 3. **Answer Model**

The **Answer** model stores the user’s response to each question within a form. It includes:

- **response**: A foreign key link to the response to which this answer belongs.
- **question**: A foreign key link to the question being answered.
- **answer_text**: A text field for text-based answers.
- **answer_choice**: A JSON field for storing multiple-choice answers (for checkboxes).

### 4. **Response Model**

The **Response** model represents a submission by a user for a particular form. It includes:

- **form**: A foreign key link to the form being submitted.
- **created_at**: A timestamp to track when the response was submitted.

---

## Serializers Overview

Serializers are used to transform model instances into JSON data and vice versa.

### 1. **FormSerializer**

Serializes the **Form** model and includes its related **Question** objects. This serializer provides data on the form and its questions in a structured format.

### 2. **QuestionSerializer**

Serializes the **Question** model and includes its related **Answer** objects. This serializer ensures that questions are displayed with the responses attached.

### 3. **AnswerSerializer**

Serializes the **Answer** model, including fields such as `answer_text` and `answer_choice`.

### 4. **ResponseSerializer**

Serializes the **Response** model along with the nested **Answer** objects. It allows for the creation of responses and answers in a single API request.

---

## Views Overview

### 1. **FormViewSet**

A **ModelViewSet** that provides CRUD operations for the **Form** model, enabling the creation, listing, retrieval, update, and deletion of forms.

### 2. **QuestionViewSet**

A **ModelViewSet** for managing **Question** objects, allowing CRUD operations on the questions in the database.

### 3. **AnswerViewSet**

A **ModelViewSet** for managing **Answer** objects, providing API endpoints for creating, updating, or retrieving answers.

### 4. **ResponseViewSet**

A **ModelViewSet** for managing **Response** objects, enabling the creation, listing, and viewing of user responses.

### 5. **Analytics Views**

- **get_form_analytics**: An API endpoint that calculates and returns analytics for a specific form, including common answers for text, checkbox, and dropdown questions. This view utilizes a helper function `get_common_choices` to aggregate common choices for checkbox and dropdown questions.
- **render_form_analytics**: A view that renders form analytics in a tabular format on a webpage for admins to view. It provides insights such as common words in text responses and most selected choices for dropdown or checkbox questions.

---

## URLs Overview

The **URLs** configuration defines the routes for the application:

1. **Router Configuration**

The **DefaultRouter** from the **rest_framework** is used to automatically generate REST API endpoints for the models:

- `forms`: Manages form creation and retrieval.
- `questions`: Manages question creation and retrieval.
- `answers`: Manages answer creation and retrieval.
- `responses`: Manages response creation and retrieval.

2. **Analytics URLs**

- `/analytics/<form_id>/`: An API endpoint for retrieving form analytics based on form ID.
- `/analytics/table/<form_id>/`: A URL that renders form analytics in a tabular format.

---

## Admin Panel Configuration

The **admin.py** file configures how the models are displayed in the Django admin interface. Customizations include:

- Inline editing for **Question** and **Answer** models.
- Displaying relevant fields such as form ID, question text, and answer choices in the admin list view.
- Registration of models to ensure they are available in the admin interface.

---

## Additional Features

- **Jazzmin Integration**: This project includes the **Jazzmin** package to enhance the Django admin interface with a modern and user-friendly design. The admin panel includes customizable themes, a responsive layout, and widgets that display key statistics.
- **Security Settings**: The **SECRET_KEY** is specified, and Django's password validators are enabled to ensure secure authentication.
- **Database Configuration**: The project uses SQLite as the database backend by default, configured in the `DATABASES` setting.

---

## Screenshots

1. **Admin Panel**

Admin Panel

2. **Forms**

Forms

3. **Questions**

Questions

### 4. **Responses**

Responses

### 5. **Django REST Framework API for Responses**

- **Page 1**
  Page 1
- **Page 2**
  Page 2

### 6. **Postman (POST Request)**

Postman POST Request

### 7. **Analytics**

Analytics

## Conclusion

This Django application allows the creation and management of dynamic forms, with features for collecting responses, generating analytics, and rendering the results in both API and tabular formats. The integration of **Jazzmin** ensures a modern and customizable admin experience, while the use of Django's built-in tools like viewsets and serializers simplifies API management.