Activating Admin Interfaces, Using Admin Interfaces, Customizing Admin Interfaces, Reasons to use Admin Interfaces. Form Processing, Creating Feedback forms, Form submissions, custom validation, creating Model Forms, URLConf Ticks, Including Other URLConfs

1. **Discuss Migration of Database with an example.**
2. **Create a simple Django project called urls_dispatcher_example with two application(articles and Blog).**
3. **Explain steps of Configuring URLs in Django.**
4. **Discuss Django Form Submission.**

Django Admin Interface

The Django admin interface is a built-in feature of the Django web framework that provides a web-based interface for managing application data. It allows developers and administrators to perform CRUD (Create, Read, Update, Delete) operations on models without having to write any custom code for these basic operations. This interface is highly customizable and can be tailored to fit the specific needs of a project.

Key Features of Django Admin Interface:

1. **Automated Admin Interface**:
   - Automatically generates a user-friendly interface for all models registered with it.
   - Provides forms for creating, updating, and deleting records.
2. **User Authentication and Authorization**:
   - Integrates with Django's authentication system to manage user permissions.
   - Allows setting different permission levels for different users (e.g., superusers, staff).
3. **Data Management**:
   - Supports listing, searching, filtering, and ordering data.
   - Enables bulk actions (e.g., deleting multiple records at once).
4. **Customization**:
   - Admin classes (ModelAdmin) can be used to customize the admin interface for specific models.
   - Supports customizing form fields, list displays, and other aspects of the interface.
5. **Extensibility**:
   - Allows adding custom actions, views, and templates to the admin interface.
   - Supports plugins and third-party extensions to enhance functionality.
6. **Security**:
   - Access to the admin interface is restricted to authenticated and authorized users.
   - Sensitive operations are protected by CSRF (Cross-Site Request Forgery) tokens and other security measures.

Activating Admin Interfaces

When you create a Django model, you can easily add it to the admin interface by registering it with Django's admin site. Here is an example:

Step 1: Define a Model:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey('Publisher', on_delete=models.CASCADE)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)
```

Step 2: Register the Model with the Admin Site:

```
from django.contrib import admin
from .models import Book
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    search_fields = ('title', 'authors__name')
```

Step 3: Access the Admin Interface:

- Add django.contrib.admin to INSTALLED_APPS.

```
# settings.py
INSTALLED_APPS = [
    'django.contrib.admin',]
```

- Include the admin URL patterns in your urls.py.

```
from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

    # other URL patterns

    url(r'^admin/', include(admin.site.urls))]
```

After the Run Development Server python manage.py runserver

Using the Admin Interface

The Django admin interface provides a comprehensive and user-friendly way to manage your application's data. Here's a detailed look at what you can do with it, including some specific examples:

1. Login Screen

To access the admin interface, you need to log in with an admin or staff user account. The login screen ensures that only authorized users can access and manage the data.

- **URL**: `/admin/login/`
- **Fields**: Username and Password
- **Actions**: Login button to authenticate the user.

2. Admin Index Page

After logging in, you're redirected to the admin index page. This page lists all the models that are registered with the admin site.

- **URL**: `/admin/`
- **Features**:
    - A list of all registered models grouped by the application.
    - Links to the change list views of each model.

3. Change List View

The change list view displays all the records for a specific model. It provides a way to list, search, filter, and perform bulk actions on records.

- **URL**: `/admin/app_name/model_name/`
- **Features**:
  - **List Display**: Shows selected fields of each record in a tabular format.
  - **Search**: A search bar to find records by specific fields.
  - **Filters**: Sidebar filters to narrow down the list based on certain criteria (e.g., by date, by related model).
  - **Actions**: Bulk actions like delete, export, or custom actions defined by the developer.
  - **Add Button**: A button to add a new record for the model.

4. Change Form View

The change form view is used for creating new records or editing existing ones. It provides a form based on the model's fields.

- **URL**: `/admin/app_name/model_name/add/` (for adding new records) or `/admin/app_name/model_name/{id}/change/` (for editing existing records).
- **Features**:
  - **Form Fields**: Inputs for all the fields defined in the model.
  - **Inline Editing**: Allows editing related objects directly within the form.
  - **Save Buttons**: Save, Save and continue editing, and Save and add another.

5. Deleting Records

You can delete records from the change list view or the change form view.

- **URL**: Typically accessed via the change form view or bulk action.
- **Features**:
  - A confirmation page to confirm the deletion of the record(s).

Customizing the Admin Interface

Customizing the Django admin interface allows you to tailor the admin site to better fit your needs and improve the user experience. Here's a detailed guide on how to customize the admin interface:

Displaying, Searching, and Filtering

When you first set up the Django admin interface for your Book model, it may only show the string representation of each book, which isn't very useful for large datasets. You can enhance the interface with additional display, searching, and filtering capabilities.

Book Model Definition:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    publication_date = models.DateField()


    class Admin:
        list_display = ('title', 'publisher', 'publication_date')
        list_filter = ('publisher', 'publication_date')
        ordering = ('-publication_date',)
        search_fields = ('title',)
```

Explanation of Options

1. **list_display**:
   - Controls which columns appear in the change list table.
   - By default, it displays only the string representation.
   - Here, it is set to show title, publisher, and publication_date.
2. **list_filter**:
   - Creates a filtering bar on the right side of the list.
   - Allows filtering by publisher and publication_date.
3. **ordering**:
   - Controls the order in which the objects are presented.
   - -publication_date orders the books by publication date in descending order.
4. **search_fields**:
   - Creates a search field that allows text searches.

Customizing Admin Interface Look and Feel

The default admin interface displays "Django administration" at the top, which you may want to customize. Django's admin interface uses its own template system, allowing you to modify these templates to fit your needs.

Steps to Customize:

1. **Locate the Default Template:**
   - The header is found in admin/base_site.html in the Django admin template directory (django/contrib/admin/templates).
2. **Copy to Custom Directory:**
   - Copy base_site.html to your custom template directory specified in TEMPLATE_DIRS.
3. **Edit the Template:**
   - Replace the placeholder text with your own site's name.

Example: If your TEMPLATE_DIRS includes "/home/mytemplates", copy django/contrib/admin/templates/admin/base_site.html to /home/mytemplates/admin/base_site.html and edit it.

Customizing the Admin Index Page

The admin index page, which lists all available applications, can also be customized.

Steps to Customize:

1. **Locate and Copy Template:**
   - The template is admin/index.html.
   - Copy it to your custom template directory.
2. **Edit the Template:**
   - The template uses {% get_admin_app_list as app_list %} to retrieve all installed applications.
   - You can hard-code links to specific admin pages or modify the layout as desired.

Shortcut: Run python manage.py adminindex <app> to get a template code chunk for inclusion in the admin index template.

Use Cases for the Django Admin Interface

The Django admin interface is a powerful tool that offers a range of functionalities, making it indispensable for various scenarios in a web development workflow. Here are detailed use cases where the admin interface excels:

1. Data Entry for Non-Technical Users

- **Scenario:**
  - In a newsroom setting, a journalist is tasked with publishing a report on water quality.
  - The journalist has collected data and needs to input it into the website.

- **Workflow:**
    - A developer creates a Django model to represent the data.
    - The developer sets up the Django admin interface for this model.
    - The journalist uses the admin interface to enter the data without needing to understand any code.
- **Benefit:**
    - Enables non-technical staff to manage and enter data, freeing developers to work on other tasks.

2. Inspecting Data Models

- **Scenario:**
    - A developer is designing a new data model for a feature.
    - They need to ensure the model is correctly defined before integrating it into the application.
- **Workflow:**
    - The developer registers the new model with the admin interface.
    - They enter some dummy data through the admin interface to see how it looks and behaves.
    - Any mistakes or necessary adjustments in the model become apparent through this visual inspection.
- **Benefit:**
    - Provides a quick and efficient way to validate and debug new models.

3. Managing Acquired Data

- **Scenario:**
    - A website, such as a crime data aggregator, pulls in data from external sources automatically.
    - Occasionally, the data may have inconsistencies or errors that need manual correction.
- **Workflow:**
    - The admin interface is set up for the relevant models.
    - When issues are identified in the data, administrators can use the admin interface to quickly make corrections.
- **Benefit:**
    - Simplifies the process of managing and correcting large datasets that are automatically imported.

4. Content Management for Websites

- **Scenario:**

- A content-heavy website, such as a blog or news site, needs to manage articles, categories, and tags.

- **Workflow:**
  - Editors use the admin interface to create, edit, and organize articles.
  - They can also manage categories and tags to ensure the content is well-organized and easy to navigate.
- **Benefit:**
  - Streamlines the content management process, allowing for quick updates and consistent categorization.

5. E-Commerce Product Management

- **Scenario:**
  - An online store needs to manage its inventory, including product listings, categories, and pricing.
- **Workflow:**
  - The admin interface is used to add new products, update existing listings, and manage product categories.
  - Price changes and inventory updates can be made in real-time.
- **Benefit:**
  - Provides a user-friendly interface for managing a complex product inventory, facilitating quick updates and changes.

Form Processing

Form processing is a critical part of web application development, ensuring that user input is handled securely and efficiently. Django provides robust tools to manage forms, making it easier to create, validate, and process form data. Here's a detailed overview based on the provided references.

1. Adding a Search View

Step-by-Step Implementation:

1. **Update URLconf:**
   - Add the search view to your URL configuration (`mysite/urls.py`):

     from django.urls import path

```
from mysite.books.views import search

urlpatterns = [ path('search/', search, name='search'),]
```

2. **Define the Search View:**

- Create the search view in `mysite/books/views.py`

```
from django.db.models import Q

from django.shortcuts import render

from mysite.books.models import Book


def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render(request, "books/search.html", {
        "results": results,
        "query": query
    })
```

2. Explanation of Key Concepts

Accessing GET Data:

- **request.GET:**
  - `request.GET` is a dictionary-like object that contains all GET parameters from the URL.

- Use `request.GET.get('q', '')` to safely retrieve the search query, defaulting to an empty string if the parameter isn't present.

Using Q Objects for Complex Queries:

- **Q Objects:**
  - Q objects allow for complex database queries, enabling OR conditions.
  - Example: `Q(title__icontains=query) | Q(authors__first_name__icontains=query)`

Ensuring Unique Results:

- **distinct():**
  - Use `.distinct()` to ensure that duplicate results (e.g., a book with multiple matching authors) are not returned multiple times.

3. Creating the Search Template

HTML Template (books/search.html):

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.01//EN">
<html lang="en">
<head>
  <title>Search {% if query %}Results{% endif %}</title>
</head>
<body>
  <h1>Search</h1>
  <form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>
  {% if query %}
    <h2>Results for "{{ query|escape }}":</h2>
    {% if results %}
      <ul>
      {% for book in results %}
```

```
            <li>{{ book }}</li>
        {% endfor %}
        </ul>
    {% else %}
        <p>No books found</p>
    {% endif %}
{% endif %}
</body>
</html>
```

4. Creating the Perfect Form

Characteristics of a Perfect Form:

- **User-Friendly:**
  - Proper use of `<label>` elements and helpful contextual messages to guide users.
- **Extensive Validation:**
  - Always validate incoming data to ensure security and integrity.
- **Error Handling:**
  - Redisplay the form with detailed error messages if the user makes a mistake, preserving the previously entered data.
- **Iterative Correction:**
  - The form should continue to prompt the user until all fields are correctly filled.

Creating Feedback forms

Creating a feedback form is a great way to engage with your users and gather valuable insights. Django makes this easy with its forms framework. Here's a step-by-step guide to creating a simple feedback form.

1. Define the Form

Create a new `forms.py` file in your application directory and define your form using Django's form classes.

Example:

```
from django import forms

TOPIC_CHOICES = (
    ('general', 'General enquiry'),
    ('bug', 'Bug report'),
    ('suggestion', 'Suggestion'),
)


class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)
```

2. Create the View

In your `views.py`, create a view to handle the form.

```
from django.shortcuts import render, redirect
from .forms import ContactForm
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process form data
            topic = form.cleaned_data['topic']
            message = form.cleaned_data['message']
            sender = form.cleaned_data.get('sender', 'noreply@example.com')
            # Save data or send email
            return redirect('thanks')  # Redirect after POST
    else:
```

```
    form = ContactForm()
  return render(request, 'contact.html', {'form': form})
```

3. Create the Template

Create a `contact.html` template to render the form.

Example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
  <title>Contact us</title>
</head>
<body>
  <h1>Contact us</h1>
  <form action="." method="POST">
    <table>
      {{ form.as_table }}
    </table>
    <p><input type="submit" value="Submit"></p>
  </form>
</body>
</html>
```

Form submissions

When the user submits a form that passes validation, we need to process the data, often by sending an email. Here's a step-by-step guide to achieve this using Django.

1. Validating the Form Data

First, ensure that the form data is valid using the `is_valid()` method. If the form is valid, access the cleaned data.

Example:

```
form = ContactForm(request.POST)
if form.is_valid():
    topic = form.cleaned_data['topic']
    message = form.cleaned_data['message']
    sender = form.cleaned_data.get('sender', 'noreply@example.com')
    # Continue processing the data
```

- `is_valid()`: Checks if the submitted data is valid.
- `cleaned_data`: Accesses the validated and converted form data.

2. Sending an Email

Use Django's `send_mail` function to send the feedback via email.

Example:

```
from django.core.mail import send_mail
send_mail(
    'Feedback from your site, topic: %s' % topic,
    message,
    sender,
    ['administrator@example.com']
)
```

`send_mail`: A function that sends an email with the specified subject, message, sender, and recipient(s).

3. Redirecting After Submission

After processing the form, redirect the user to a confirmation page.

Example:

```
from django.http import HttpResponseRedirect
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
```

```
            topic = form.cleaned_data['topic']
            message = form.cleaned_data['message']
            sender = form.cleaned_data.get('sender', 'noreply@example.com')
            send_mail(
                'Feedback from your site, topic: %s' % topic,
                message,
                sender,
                ['administrator@example.com']
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

4. Custom Validation Rules

To enforce custom validation, such as requiring a minimum number of words in the message, define a `clean_<fieldname>` method.

Example:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)
     def clean_message(self):
        message = self.cleaned_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

4. Custom Validation Rules

To enforce custom validation, such as requiring a minimum number of words in the message, define a `clean_<fieldname>` method.

Example:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)
    def clean_message(self):
        message = self.cleaned_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Creating Model Forms

1. Define the Publisher Model

First, create a model for your data in `models.py`.

```
from django.db import models
class Publisher(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=100)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()
    def __str__(self):
        return self.name
```

2. Create a Model Form

Generate a form from the model using `ModelForm` in `forms.py`.

forms.py:

```
from django import forms
from .models import Publisher
class PublisherForm(forms.ModelForm):
    class Meta:
        model = Publisher
        fields = ['name', 'address', 'city', 'state_province', 'country', 'website']
```

3. Create a View to Handle the Form

Write a view function to display and process the form in `views.py`.

views.py:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import PublisherForm
def add_publisher(request):
    if request.method == 'POST':
        form = PublisherForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/add_publisher/thanks/')
    else:
        form = PublisherForm()
    return render(request, 'add_publisher.html', {'form': form})
```

4. Create a Template for the Form

Create a template file to render the form in `templates/add_publisher.html`.

add_publisher.html:

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <title>Add Publisher</title>
</head>
<body>
    <h1>Add a New Publisher</h1>
    <form action="." method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

1) **Discuss Migration of Database with an example**

Migration of Database in Django

Database migration in Django is a way to propagate changes you make to your models (like adding a field or deleting a model) into your database schema. Django's built-in migration system uses a series of migration files to keep track of changes and apply them to the database.

Step 1 : Setup Your Django App

2. Define a Model

Create a model in your app's `models.py` file.

myapp/models.py:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
```

3. Create Initial Migrations

Run the `makemigrations` command to create the initial migration files based on your model definition.

python manage.py makemigrations myapp

This command will create a migration file in the `migrations` directory of your app.

4. Apply Migrations

Run the `migrate` command to apply the migrations and create the corresponding table in the database.

python manage.py migrate

5. Make Changes to the Model

Now, let's say you want to add a new field to the `Book` model.

myapp/models.py:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13, null=True, blank=True)  # New field added
```

6. Create Migrations for the Changes

Run the `makemigrations` command again to create a new migration file for the changes.

python manage.py makemigrations myapp

7. Apply the New Migrations

Run the `migrate` command to apply the new migration and update the database schema.

python manage.py migrate

2) Explain steps of Configuring URLs in Django.

Configuring URLs in Django involves mapping URLs to views, allowing your Django application to respond to specific URLs with the appropriate view functions or class-based views. Here are the steps to configure URLs in Django:

Steps to Configure URLs in Django

1. Define URL Patterns in Your App

In Django, URL patterns are defined using regular expressions (regex) in a urls.py file within each Django app. Here's how you define URL patterns:

a. Create a new urls.py file:

Inside your Django app directory (e.g., myapp), create a file named urls.py if it doesn't already exist.

b. Define URL patterns:

In urls.py, import Django's path function and any view functions or class-based views you want to associate with URLs.

Example (myapp/urls.py):

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.index, name='index'),
```

```
    path('about/', views.about, name='about'),
    path('books/<int:book_id>/', views.book_detail, name='book_detail'),
    # Add more paths as needed
]
```

2. Include App URLs in Project URLs

After defining URL patterns for each app, include these URLs in the project-level urls.py file. This step connects your app-specific URLs to the main URL configuration of your Django project.

a. Edit the project-level urls.py file:

Open urls.py in your Django project directory (e.g., myproject/myproject/urls.py).

b. Include app-specific URLs:

Use include() to include URLs from your app's urls.py.

Example (myproject/urls.py):

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),
]
```

3) Create a simple Django project called urls_dispatcher_example with two application(articles and Blog).

1. Setup Django Environment

```
pip install Django
```

2. Create Django Project

```
django-admin startproject urls_dispatcher_example
```

3. Create Django Applications

Now, inside the `urls_dispatcher_example` directory, create two Django applications: `articles` and `blog`.

cd urls_dispatcher_example

python manage.py startapp articles

python manage.py startapp blog

4. Configure Django Apps

Edit `urls_dispatcher_example/settings.py` to add the newly created apps to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'articles',  # Add this line
    'blog',      # Add this line
]
```

5. Define URLs for Each App

Each app will have its own set of URL patterns defined in their respective `urls.py` files.

For `articles` app:

Create `articles/urls.py` and define URL patterns for the `articles` app.

```
# articles/urls.py
from django.urls import path
from . import views
urlpatterns = [
    path('', views.article_list, name='article_list'),
    path('<int:article_id>/', views.article_detail, name='article_detail'),]
```

For `blog` app:

Create `blog/urls.py` and define URL patterns for the `blog` app.

```
# blog/urls.py

from django.urls import path

from . import views

urlpatterns = [

    path('', views.post_list, name='post_list'),

    path('<int:post_id>/', views.post_detail, name='post_detail'),]
```

6. Include App URLs in Project URLs

In `urls_dispatcher_example/urls.py`, include the URLs from each app using `include()`.

```
# urls_dispatcher_example/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('articles/', include('articles.urls')),  # Include articles app URLs

    path('blog/', include('blog.urls')),          # Include blog app URLs

]
```

7. Define Views for Each URL Pattern

Finally, define views for each URL pattern in your `views.py` files inside `articles` and `blog` apps (`articles/views.py` and `blog/views.py`).

8. Create Templates (Optional)

Create HTML templates for rendering the views. Place these templates in `articles/templates/articles/` and `blog/templates/blog/` directories respectively.

9. Run Migrations and Start Development Server

Run the following commands to apply migrations and start the Django development server:

Run server last python manage.py runserver