# Django Interview Preparation

## 1. Types of inheritance in Django

There are three types of inheritance that Django supports

- Abstract Base Class
- Multi-table Inheritance
- Proxy Models

Abstract Base Class

1. Parent Class has attributes common for many child classes
2. Parent Class is only used for inheritance, not saved in database
3. In Parent's Meta class, 'abstract' is marked as True
4. Child class can inherit from many parent classes.
5. Parent class can't be used for creating objects

```
class Person(models.Model):
    # common fields
    class Meta:
        abstract = True

class Student(Person):
    pass

class Teacher(Person):
    pass

Person() #Not callable|Can't create objects
```

Multi-table inheritance

1. Every model is a model in itself.
2. One-to-One link is created b/w tables automatically.

```
class Student(models.Model):
    # Student Attributes
    pass

class Teacher(Student):
    # Teacher inherits the properties of Student
    pass
```

Proxy Models

1. Proxy(*something authorized to act on behalf of another*) models altering some properties of base model like ordering or adding new method
2. Changes the behavior of original model, won't be saved in DB
3. Proxy model will also operate on the base model

```
class Person(models.Model):
    pass
```

```
class MyPerson(Person):
    class Meta:
        proxy = True
        ordering = ["last_name"]

    def do_something(self):
        pass
```

## 2. What is a manage.py file in Django? List some commands

Manage.py file is automatically created when a new project is created & is used for administrative tasks. It is more like django-admin but also sets DJANGO_SETTINGS_MODULE for project settings(which db, middleware)

| Command | Function | Example |
|---|---|---|
| runserver | Starts a lightweight dev server on local machine on port 8000 and ip 127.0.0.1. Automatically reloads code on each HTTP request. Can provide local machine ip to make project accessible to other users. | `$ python manage.py runserver 192.168.1.110:8000` |
| check | Check apps for common problems | `$ python manage.py check appname` |
| dumpdata | Returns all data within all apps on shell. Can be exported to a json, also can be exported for an app, all tables or a single table. | `$ python manage.py dumpdata appname && python manage.py dumpdata appname > appname.json && python manage.py dumpdata appname.model_class_name` |
| loaddata | Loads named fixtures into DB | `$ python manage.py loaddata fixture_name.json` |
| flush | Irreversibly destroy data currently in DB & return each table to an empty state | `$ python manage.py flush` |

| | | |
|---|---|---|
| makemigrations | Migrations are kind of a version control system for DB schema. makemigrations pack up the changes detected in the models into migration files(similar to a commit).<br><br>New migrations are compared with the old migrations before creation. | `python $ python manage.py makemigrations --name changed_my_model appname1, appname2` |
| migrate | Synchronizes the database state with the current set of models and migrations | `$ python manage.py migrate changed_my_model --fake` |

# 3. What is a Fixture?

Fixtures are initial data for your database which can be used with *manage.py loaddata* command. Django will look for a fixtures folder in your app or the list of directories provided in the FIXTURE_DIRS in settings, and use its content instead.

# 4. Explain Migration in Python Django.

- Model with a dependency is created first, say Teacher has foreign key to Person model, the Person model is created first
- Each app has migrations stored in migration file with class representation. Django.db.migrations.Migration is the migration class
- Each migration class lists dependencies(a list of mig this depends on) and operations(what was changed in models)
- These operations are then used to create the SQL which then makes the changes
- It is possible to write migration files manually without running the 'makemigrations' command
- Initial migrations are marked with an initial = True class attribute on the migration class
- If you have manually changed your db, Django won't be able to migrate changes | errors
- Migrations can be reverted by passing the migration number `$ python manage.py migrate app 002`
- Migrations can also be named `$ python manage.py makemigrations --name changed_my_model your_app_label`

# 4a. How does Django knows which migration to apply?

Django creates a 'django_migrations' table in db with fields <ID | App | Name | Applied> when you apply any migration the first time. A new row is inserted for each migration that is applied or faked.

## 4b. Explain fake migration in Django

A **fake migration** marks a migration as applied (or unapplied) in the database *without* actually running the SQL commands. It's useful when:

- You've manually applied schema changes.
- You're synchronizing migrations across environments.

**Command**:

```
python manage.py migrate --fake <app_name> <migration_name>
```

**Example**:

```
python manage.py migrate myapp 0002_auto --fake
```

**Caution**: Use only when the database state matches the migration's expected state to avoid inconsistencies.

## 5. Difference Between a Project & App?

A project is whole idea behind your application and an app is the sub-module of that idea. Example- Users as a micro-service project with profile app for profile related management, authentication app and so on. In a general way, a project could be the whole website and an app could be functionalities it offers, like request app, result app, frontend app and so on.

## 6. How do we initialize a project & an app?

**Project**

- Create the project `$ django-admin.py startproject project_name`
- Update your project settings in ./settings.py like installed_apps, middlewares, db, project variables & so on.
- Validate installment | use Django dev test server
- Version control project, creates requirements.txt

**App**

- Create the app `$ python manage.py startapp app_name`
- Add app into project settings.py installed_apps
- Make migrations & migrate models

# 7. What does the settings.py file do?

The `settings.py` file in Django is the main configuration file for your project. It contains all the essential settings, such as:

- **Database Configuration** (e.g., `DATABASES`)
- **Installed Apps** (`INSTALLED_APPS`)
- **Middleware** (`MIDDLEWARE`)
- **Static & Media Files** (`STATIC_URL`, `MEDIA_ROOT`)
- **Templates** (`TEMPLATES`)
- **Security Settings** (`SECRET_KEY`, `DEBUG`, `ALLOWED_HOSTS`)
- **Internationalization** (`LANGUAGE_CODE`, `TIME_ZONE`)

It controls how Django behaves and allows customization for different environments (development, production).

**Example**:

```
DEBUG = True  # Turns debug mode on (never use in production!)
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db.sqlite3',
    }
}
```

Modify it carefully, as incorrect settings can break your app.

# 8. How do you set up a database connection?

**1. SQL Databases (e.g., PostgreSQL, MySQL, SQLite)**

Edit `settings.py`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',  # or 'mysql', 'sqlite3'
        'NAME': 'mydatabase',
        'USER': 'mydbuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',  # or IP
        'PORT': '5432',       # default PostgreSQL port
    }
}
```

**For SQLite (default in Django):**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',  # auto-created if missing
    }
}
```

---

### 2. NoSQL Databases (e.g., MongoDB, Firebase)

Django doesn't natively support NoSQL, but you can use third-party packages:

#### A. MongoDB (Using `djongo` or `django-mongodb-engine`)

1. **Install:**

   ```
   pip install djongo
   ```

2. **Configure `settings.py`:**

   ```
   DATABASES = {
       'default': {
           'ENGINE': 'djongo',
           'NAME': 'mydb',
           'CLIENT': {
               'host': 'mongodb://localhost:27017',
           }
       }

   }
   ```

#### B. Firebase (Using `firebase-admin`)

1. **Install:**

   ```
   pip install firebase-admin
   ```

2. **Set up manually (no Django DB backend). Example:**

   ```
   import firebase_admin
   from firebase_admin import credentials, firestore

   cred = credentials.Certificate("path/to/serviceAccountKey.json")
   firebase_admin.initialize_app(cred)

   db = firestore.client()  # Use like a Python dict
   ```

### Key Notes:

- **SQL databases** work out-of-the-box with Django's ORM.

- **NoSQL databases** require extra packages and may not support Django's ORM fully.
- For **Firebase/CouchDB**, you'll mostly interact via their SDKs instead of Django models.

## 9. Django Templating Language

Django Templating Language (DTL) is Django's built-in templating system used to dynamically generate HTML. It uses tags (`{% %}`), variables (`{{ }}`), and filters (`{{ var|filter }}`) to render data from views. It supports inheritance, control structures (like loops and conditions), and auto-escaping for security.

## 10. What are CSRF Tokens?

CSRF (Cross-Site Request Forgery) tokens are security tokens used to prevent unauthorized commands from being transmitted from a user that the web application trusts. In Django, `{% csrf_token %}` is added in forms to ensure that POST requests are from authenticated sources.

## 11. What are Django Signals?

Django Signals allow decoupled components to get notified when certain actions occur (like saving or deleting a model). They use a dispatcher to connect **senders** and **receivers**. Common signals: `pre_save`, `post_save`, `pre_delete`, `post_delete`.

## 12. How can we set restrictions on views?

Restrictions on views in Django can be set using:

1. **Login Required**: `@login_required` decorator to restrict access to authenticated users.
2. **Permission Required**: `@permission_required` decorator for specific permissions.
3. **User Passes Test**: `@user_passes_test` for custom conditions.
4. **Class-Based Views**: Use `LoginRequiredMixin` or `PermissionRequiredMixin`.

## 13. Explain transaction in Django

A **transaction** in Django ensures a set of database operations are executed atomically — either all succeed or none do. It uses `django.db.transaction`. Key methods:

- `@transaction.atomic`: Wraps code in a transaction block.
- `transaction.set_rollback(True)`: Manually trigger rollback. Useful for maintaining data integrity.

## 14. What is the use case of REST API? Why should we use it?

**REST API** is used to allow communication between client and server over HTTP using standard methods (GET, POST, PUT, DELETE).

**Use cases:**

- Mobile apps, SPAs, or third-party services interacting with your backend.
- Decoupling frontend and backend.
- Scalable, reusable APIs.

**Why use it:**

- Platform-independent.
- Lightweight and stateless.
- Easy integration and maintenance.

# 15. Explain Serialization.

**Serialization** in Django (especially Django REST Framework) is the process of converting complex data types (like querysets or model instances) into native Python data types that can then be rendered as JSON, XML, etc.

It also supports **deserialization** — converting incoming data (e.g., JSON) back into Django model instances after validation. Used for API data exchange.

# 16. Explain all the files that are created within a new project.

When you create a new Django project (`django-admin startproject project_name`), it creates:

1. **manage.py** – Command-line utility to manage the project (e.g., runserver, migrations).
2. **project_name/** – Inner directory with:
    - **init.py** – Marks the directory as a Python package.
    - **settings.py** – Project settings/config (DB, apps, middleware, etc.).
    - **urls.py** – URL declarations (routing).
    - **asgi.py** – Entry point for ASGI-compatible web servers.
    - **wsgi.py** – Entry point for WSGI-compatible web servers.

# 17. Explain Django Architecture.

**Django Architecture** follows the **MTV pattern** (Model-Template-View):

1. **Model** – Handles data and business logic (database layer).
2. **Template** – Deals with the presentation layer (HTML UI).
3. **View** – Controls logic and interacts with both model and template.

It's similar to MVC:

- **View (in MTV)** = Controller (in MVC)
- **Template** = View (in MVC)
- **Model** = Model (same in both)

Django also uses **URL dispatcher** to map URLs to views.

# 18. Explain Sessions in Django

**Sessions** in Django store data on the server to keep track of user interactions across requests (e.g., login state, cart items).

- Each user gets a unique **session ID** stored in a cookie.
- Data is stored in the database, cache, or file (configurable).
- Access via `request.session` (like a dictionary).
- Supports expiration and session management.

# 19. Explain Squashing Migration in Django.

Squashing is the act of reducing an existing set of many migrations down to one (or sometimes a few) migrations which still represent the same changes. Django takes all the operations from the migrations, puts them in sequence and optimizes them.

- Create model and Delete model --> No operation
- You can name squashed migrations `python manage.py squashmigrations myapp --squashed-name squashed_migration_01`
- All migration files must be deleted after a squashed migration(leaving the squashed one)
- Update dependencies of deleted migrations to squashed migration.
- Remove 'replaces' from squashed migration file. 'replaces' is what makes it a squashed migration

# 20. Difference between class based views and function based views. When to use which one?

**Function-Based Views (FBVs):**

- Simple Python functions.
- Easy to read/write for simple logic.
- Better for small, one-off views.

**Class-Based Views (CBVs):**

- Use OOP.
- Reusable, extensible, and organized.
- Ideal for complex views (e.g., `ListView`, `CreateView`).

**When to use:**

- **FBVs**: When logic is simple and straightforward.
- **CBVs**: When reusability, scalability, or built-in generic views are needed.

# 21. Explain the request flow in Django.

**Django Request Flow:**

1. **Browser sends request** → Django receives it via **WSGI/ASGI**.
2. **URL Dispatcher** (`urls.py`) matches the URL to the appropriate **view**.
3. **View** processes logic, may interact with **models** and fetch data.
4. View passes data to a **template** (if rendering HTML).
5. Template renders and returns an **HTTP response** to the browser.

In short:
**Request → URLconf → View → Model/Template → Response**.

# 22. Explain middlewares in Django. Triggers and all

**Middleware** in Django is a framework of hooks that process requests/responses globally before they reach the view or after the response leaves the view.

## Common Triggers (Middleware Hooks):

- `__init__()` – Runs once when server starts.
- `process_request(request)` – Before view is called.

- `process_view(request, view_func, view_args, view_kwargs)` – Before view logic executes.
- `process_template_response(request, response)` – After view, before template rendering.
- `process_response(request, response)` – After view returns a response.
- `process_exception(request, exception)` – If view raises an exception.

Used for:

- Security (e.g., `CsrfViewMiddleware`)
- Authentication
- Logging
- Session management
- Request/response modification

# 23. Explain mixins. What makes mixins different.

**Mixins** in Django (especially Django REST Framework) are reusable classes that add specific behavior to views or classes without needing full inheritance.

### What makes mixins different:

- **Modular**: Provide small, focused pieces of functionality (e.g., `CreateModelMixin`, `UpdateModelMixin`).
- **Reusable**: Can be combined with other classes.
- **Do one thing well**: Unlike full views or base classes.

### Common Use:

Used with **Generic Views** to add CRUD operations:

```
class MyView(CreateModelMixin, ListModelMixin, GenericAPIView):
    ...
```

They promote **DRY** (Don't Repeat Yourself) principles.

# 24. How to migrate from one database to another? Use of two setting files.

To **migrate from one database to another** in Django, follow these steps:

### 1. Create Two Settings Files

- `settings_old.py` → contains old DB config.
- `settings_new.py` → contains new DB config.

### 2. Export Data from Old DB

```
python manage.py dumpdata --settings=settings_old > data.json
```

### 3. Migrate New DB Schema

```
python manage.py migrate --settings=settings_new
```

### 4. Import Data into New DB

```
python manage.py loaddata data.json --settings=settings_new
```

### 5. (Optional) Switch default `settings.py`

Update your default `settings.py` with the new DB config.

This approach helps keep environments separate and safe during migration.

## 25. Explain bulk_create on post in DRF.

In **Django REST Framework (DRF)**, `bulk_create` allows you to insert multiple objects into the database efficiently with a single query.

### Use Case:

When posting a list of objects (e.g., list of users, tasks) to an endpoint.

### Example:

```
def create(self, request):
    serializer = MySerializer(data=request.data, many=True)
    serializer.is_valid(raise_exception=True)
    objs = [MyModel(**item) for item in serializer.validated_data]
    MyModel.objects.bulk_create(objs)
    return Response(serializer.data, status=201)
```

### Key Points:

- `many=True` in serializer.
- No signals (`save()` isn't called), so use carefully.
- Much faster than saving objects one by one.

# 26. Explain REST HTTP methods

| Method | CRUD | Entire Collection | Specific Item |
|---|---|---|---|
| GET | Read | 200(OK) | 200(OK), 404(Not Found) |
| POST | Create | 201(Created) | 404(Not found), 409(Conflict) |
| PUT | Update | 405(Method Not Allowed) | 200(OK), 204(No Content), 404(Not Found) |
| PATCH | Modify | 405(Method Not Allowed) | 200(OK), 204(No Content), 404(Not Found) |
| DELETE | Delete | 405(Method Not Allowed) | 200(OK), 404 (Not Found) |

## Status Codes

- Informational 1XX
- Successful 2XX
- Redirection 3XX
- Client Error 4XX
- Server Error 5XX

## Difference between a POST and PUT request.

Calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly make have side effects of creating the same resource multiple times.

# 27. How would you create models from legacy DB?

1. Configure `settings.py` – Set up the legacy database connection.

2. Run `inspectdb`
   `python manage.py inspectdb > models.py`
3. Edit Models – Set `managed=False` in `Meta` to prevent Django from altering tables.

4. Fake Migrate
   `python manage.py migrate --fake`

Note: Works only for SQL databases (PostgreSQL, MySQL, etc.). For NoSQL, use SDKs directly.

# 28. Difference between Django and Flask

### Django vs. Flask

**1. Framework Type**

- **Django** – Full-featured, "batteries-included" (ORM, Admin, Auth, etc.).
- **Flask** – Microframework (lightweight, minimal core, extensible via plugins).

**2. Complexity & Use Cases**

- **Django** – Best for structured, large-scale apps (e.g., CMS, e-commerce).
- **Flask** – Ideal for small/APIs, microservices, or custom lightweight apps.

**3. Key Features**

| Feature | Django | Flask |
|---|---|---|
| **ORM** | Built-in (Django ORM) | None (use SQLAlchemy/Peewee) |
| **Admin Panel** | Yes (auto-generated) | No (build manually) |
| **Auth** | Built-in | Manual/Flask-Login |
| **Templating** | Django Templates | Jinja2 (default) |
| **Routing** | URL-based (`urls.py`) | Decorator-based (`@app.route`) |

**4. Flexibility**

- **Django** – Opinionated (follows "Django way").
- **Flask** – Unopinionated (choose your tools).

**5. Performance**

- Similar (Flask is slightly faster for trivial tasks; Django optimizes for scale).

**Choose Django** for rapid development with built-in tools.

**Choose Flask** for minimal control or lightweight APIs.

---

**One-Liner:**

*"Django is a ready-to-use SUV; Flask is a customizable bike."* 🚗 vs. 🚲

# 29. Explain Middlewares. What methods are required within a middleware?

**Middleware** is a lightweight plugin that processes **requests/responses globally** before they reach the view or after they leave it.

---

### Required Methods in a Middleware

A custom middleware must include at least **one** of these methods:

1. `process_request(request)`
   - Runs **before** the view.
   - Can return `None` (continue) or `HttpResponse` (short-circuit).
2. `process_response(request, response)`
   - Runs **after** the view.
   - Must return an `HttpResponse`.
3. `process_view(request, view_func, view_args, view_kwargs)`
   - Runs **just before** the view is called.
4. `process_exception(request, exception)`
   - Runs if the view **raises an exception**.
5. `process_template_response(request, response)`
   - Modifies template responses.

---

### Example Middleware

```python
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Code before view (process_request)
        print("Before view")
        response = self.get_response(request)
        # Code after view (process_response)
        print("After view")
        return response
```

---

### Key Notes

- Order matters (defined in `MIDDLEWARE` in `settings.py`).
- Used for auth, logging, CORS, etc. (e.g., `django.middleware.security.SecurityMiddleware`).

**One-Liner:**

*"Middleware is Django's bouncer—it checks requests/responses before they enter/exit."* 🚧

# 30. Explain the request flow when a url is hit.

1. **URL Dispatcher**
   - Matches the URL to a view in `urls.py` (via `path()` or `re_path()`).
2. **Middleware (Before View)**
   - Runs `process_request()` & `process_view()` in order (e.g., security, sessions).
3. **View Execution**
   - The matched view processes the request (e.g., `def my_view(request):`).
4. **Middleware (After View)**
   - Runs `process_response()` in reverse order (post-processing).
5. **Response Sent**
   - Final `HttpResponse` is returned to the client.

---

### Key Points

- **Short-circuiting**: Middleware can return early (e.g., auth failure).
- **Order Matters**: Middleware sequence affects behavior.
- **Error Handling**: `process_exception()` runs if the view crashes.

**One-Liner:**

*"URL → Middleware (pre) → View → Middleware (post) → Response."* 🔄