# What Is The Command To Install Django & To Know About It's Version?

Command To Install Django:

```
pip install django
```

Command To Check Django Version:

```
python -m django --version
```

Command To Check all the versions of installed modules:

```
pip freeze
```

# What Is The Command To Create A Project & An App In Django?

Command To Create A Project:

```
django-admin startproject nitman
```

Command To Create An App:

```
python manage.py startapp nitapp
```

*where **nitman** is project name & **nitapp** is app name.*

# What Is The Command To Run A Project In Django?

Command To Run A Project:

```
python manage.py runserver
```

*By default, this command starts the development server on the internal IP at port 8000.*

If you want to change the server's port, pass it as a command-line argument.
For instance, this command starts the server on port 8080:

```
python manage.py runserver 8080
```

If you want to change the server's IP, pass it along with the port, use:

```
python manage.py runserver 0.0.0.0:8000
```

# What Is The Command For Migrations In Django?

Command to create a migration file inside the migration folder:

```
python manage.py makemigrations
```

After creating a migration, to reflect changes in the database permanently execute migrate command:

```
python manage.py migrate
```

To see raw SQL query executing behind applied migration execute the command:

```
python manage.py sqlmigrate app_name migration_name
python manage.py sqlmigrate nitapp 0001
```

To see all migrations, execute the command:

```
python manage.py showmigrations
```

To see app-specific migrations by specifying app-name, execute the command:

```
python manage.py showmigrations nitapp
```

# What Is The Command To Create A Superuser In Django?

Command To Create A SuperUser:

```
python manage.py createsuperuser
```

Enter your desired username and press enter.

```
Username: admin
```

You will then be prompted for your desired email address:

```
Email address: admin@example.com
```

The final step is to enter your password twice,
the second time as a confirmation of the first.

```
Password: **********
Password (again): *********
Superuser created successfully.
```

# What Is The Django Command To View A Database Schema Of An Existing (Or Legacy) Database?

Command to view a database schema of an existing (or legacy) database:

```
python manage.py inspectdb
```

# How To View All Items In The Model Using Django QuerySet?

Django Command To View All Items In A Model:

```
Users.objects.all()
```

*where "User" is a model name.*

# How To Filter Items In The Model Using Django QuerySet?

Django Command To Filter Items In A Model:

```
Users.objects.filter(name="Nitin")
```

*where “User” is a model name.*

# How To Get A Particular Item In The Model Using Django QuerySet?

Django Command To Get A Particular Item In A Model:

```
Users.objects.get(id=25)
```

*where "User" is a model name.*

# How to Delete/Insert/Update An Object Using QuerySet In Django?

QuerySet To Delete An Object:

```
Users.objects.filter(id= 54).delete()
```

QuerySet To Update An Object:

```
user_to_be_modify = User.objects.get(pk = 3)
user_to_be_modify.city = "Pune"
user_to_be_modify.save()
```

QuerySet To Insert/Add An Object:

```
new_user = User(name = "Nitin Mangotra", city="Gurgaon")
new_user.save()
```

# How Can You Combine Multiple QuerySets In A View?

**Let's suppose the following two models in Django.**

```
class Blog(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField(blank=True)
```

```
class Email(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField(blank=True)
```

**Let's suppose the following three querysets generated from above models, that you want to combine.**

```
>>> query_set_1 = Blog.objects.filter(id__gte=3)
>>> query_set_2 = Email.objects.filter(id__lte=11)
>>> query_set_3 = Blog.objects.filter(id__lte=2)
```

# How Can You Combine Multiple QuerySets In A View?

```
>>> query_set_1 = Blog.objects.filter(id__gte=3)
>>> query_set_2 = Email.objects.filter(id__lte=11)
>>> query_set_3 = Blog.objects.filter(id__lte=2)
```

## 1. Using Union Operator:

If both querysets belong to the **same model**, such as **query_set_1** & **query_set_3** above, then you can use union operator "|" to easily combine those querysets.

```
query_set_result = query_set_1 | query_set_3
```

You can use the union operator to combine two or more querysets as shown below.

```
combined_result= query_set_1 | query_set_3 | query_set_4 ...
```

# How Can You Combine Multiple QuerySets In A View?

```
>>> query_set_1 = Blog.objects.filter(id___gte=3)
>>> query_set_2 = Email.objects.filter(id___lte=11)
>>> query_set_3 = Blog.objects.filter(id___lte=2)
```

## 2. Using Itertools:

If both querysets belong to the **different model**, such as **query_set_1** & **query_set_2** above, then you can use itertools combine those querysets.

```
from itertools import chain
combined_list = list(chain(query_set_1,query_set_2))
```

You just need to mention the querysets you want to combine in a comma-separated manner in chain function. You can also use it to combine more than two querysets.

```
combined_list = list(chain(query_set_1, query_set_2, query_set_3))
```

***There is an issue with this approach, you won't get a queryset, you'll get a list containing instances.***

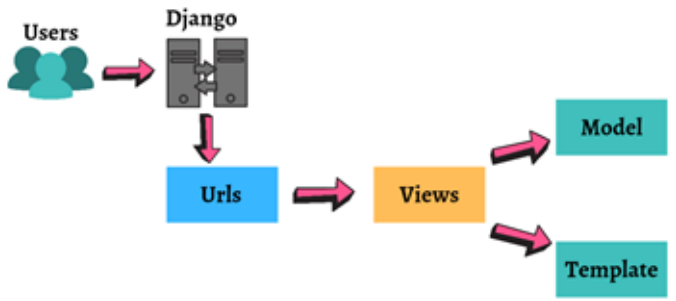
# Explain Django Architecture? Also Explain Model, Template And Views.

- Django follows a software design pattern called as **MVT (Model View Template) Architecture.**
- It is based on the Model View Controller architecture (MVC). But is slightly different from the MVC pattern as it maintains its own conventions, so, the controller is handled by the framework itself.
- **Model:** It helps in handling the database (Models). They provide the options to create, edit and query data records in the database
- **Template**: The template is a presentation layer. It defines the structure of file layout to represent data in a web page. It is an HTML file mixed with Django Template Language (DTL).
- **View**: The View is used to execute the business logic and interact with a model to carry data and renders a template.

# Explain Django Architecture? Also Explain Model, Template And Views.

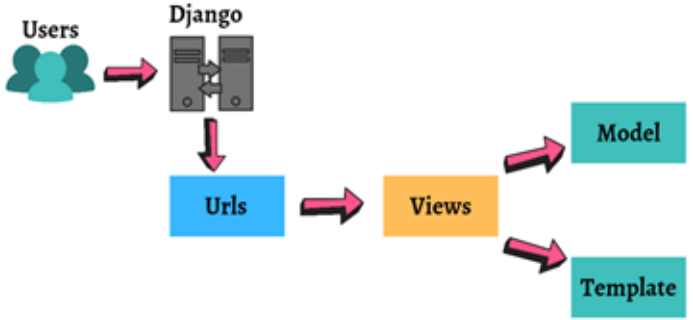
The developer provides the model, the view, and the template then maps it to a URL, and finally, Django serves it to the user.

For Example:




- Here, a user **requests** for a resource to the Django, Django works as a controller and check to the available resource in URL. (urls.py file)
- If URL maps, **a view** is called that interact with **model** and **template**, it renders a **template**.
- Django responds back to the user and sends a template as a **response**.

# Explain How A Request Is Processed In Django?




- Here, a user requests for a resource to the Django, Django works as a controller and check to the available resource in URL.
- When Django server is started, the manage.py file searches for settings.py file, which contains information of all the applications installed in the project, middleware used, database connections and path to the main urls config.
- **Manage.py >> Setting.py >> urls.py >> views.py >>**
- Django first determines which root URLconf or URL configuration module is to be used
- Then, that particular Python module urls is loaded and then Django looks for the variable urlpatterns
- Then check each URL patterns in urls.py file, and it stops at the first match of the requested URL
- Once that is done, the Django then imports and calls the given view.
- In case none of the URLs match the requested URL, Django invokes an error-handling view
- If URL maps, a view is called that interact with model and template, it renders a template.
- Django responds back to the user and sends a template as a response.

# What Is The Difference Between A Project And An App In Django?

**In Simple Words:**

- **A Project** is the entire Django application and an **App** is a module inside the project that deals with one specific use case.
  For Example:- payment system(app) in the eCommerce app(Project).

- **An App** is basically a web Application that is created to perform a specific task.
- **A project**, on the other hand, is a collection of these apps.
- Therefore, a single project can consist of 'n' number of apps and a single app can be in multiple projects.

# What Is The Difference Between A Project And An App In Django?

Command To Create A Project:

```
django-admin startproject nitman
```

Command To Create An App:

```
python manage.py startapp nitapp
```

*where **nitman** is project name & **nitapp** is app name.*

# What Is The Difference Between A Project And An App In Django?

To start a project in Django, use command

```
$ django-admin.py  nitman
```

A Typical Django Project Directory Structure:

```
nitman/
   manage.py
   nitman/
         ___init___.py
         settings.py
         urls.py
         wsgi.py
```

The last four files are inside a directory, which is at the same level of manage.py.

**manage.py**: A command-line utility that allows you to interact with your Django project & this file is used to control your Django project on the server or even to begin one.

**___init___.py**: An empty file that tells Python that the current directory should be considered as a Python package

**settings.py**: Comprises the configurations of the current project like DB connections, middlewares etc

**urls.py**: All the URLs of the project are present here

**wsgi.py**: This is an entry point for your application which is used by the web servers to serve the project you have created.

# What Is The Difference Between A Project And An App In Django?

To start an app in Django, use command

```
$ python manage.py  startapp nitapp
```

A Typical Django App directory structure:

```
nitapp/
   ___init___.py
   admin.py
   apps.py
   migrations/
                ___init___.py
   models.py
   tests.py
   views.py
```

**___init___.py** - An empty file that tells Python that the current directory should be considered as a Python package

**admin.py:** Reads model metadata and provides an interface to manage app content

**app.py:** Application configuration details for the app are included e.g custom app name.

**migrations/:** Contains migrated model details with the corresponding database table structure

**models.py:** A class for each model is defined with the model structure layout

**tests.py:** App unit test automation classes are included in this

**views.py:** Web based requests and response is configured in this file

# Which Is The Default Database In Settings File In Django?

Database Name: **SQLite**

# Why Is Django Called A Loosely Coupled Framework?

- Django is called **a loosely coupled framework** because of its MVT architecture, which is a variant of the MVC architecture.
- MVT helps in separating the server code from the client-related code.
- Django's Models and Views are present on the client machine and only templates return to the client, which are essentially HTML, CSS code and contains the required data from the models.
- These components are totally independent of each other and therefore, front-end developers and backend developers can work simultaneously on the project as these two parts changing will have little to no effect on each other when changed.
- **Therefore, Django is a loosely coupled framework.**

# Which Is The Default Port For The Django Development Server?

Default Port For Django Development Server: **8000**

*By default, the server runs on port 8000 on the IP address 127.0. 0.1*

# Explain The Migration In Django.

- **Migration** in Django is to make changes to our models like deleting a model, adding a field, etc. into your database schema.
- A migration in Django is a Python file that contains changes we make to our models so that they can be converted into a **database schema** in our **DBMS**.
- So, instead of manually making changes to our database schema by writing queries in our DBMS shell, we can just make changes to our models.
- Then, we can use Django to **generate migrations** from those model changes and run those migrations to make changes to our database schema.

# Explain The Migration In Django.

There are several commands you use to interact with Migrations In Django:

- **makemigration** - This command is used to create a migration file.
- **migrate** - This command creates a table according to the schema defined in migration file.
- **showmigrations** - This command list all the migrations and their status.
- **sqlmigrate** - This command is used to show a raw SQL query of the applied migration.

# Explain The Migration In Django.

Command to create a migration file inside the migration folder:

```
python manage.py makemigrations
```

After creating a migration, to reflect changes in the database permanently execute migrate command:

```
python manage.py migrate
```

To see raw SQL query executing behind applied migration execute the command:

```
python manage.py sqlmigrate app_name migration_name
python manage.py sqlmigrate nitapp 0001
```

To see all migrations, execute the command:

```
python manage.py showmigrations
```

To see app-specific migrations by specifying app-name, execute the command:

```
python manage.py showmigrations nitapp
```

# What is Django ORM?

- ORM stands for **Object-relational Mapper.**
- This ORM enables us to interact with databases in a more pythonic way like we can avoid writing raw queries.
- It is possible to retrieve, save, delete and perform other operations over the database without ever writing any SQL query.
- It helps us with working with data in a more object-oriented way.

**Let's consider a simple SQL Query where Employee table to retrieve a employee name.**

```
Select * from EMPLOYEE where name = "Nitin";
```

The Equivalent Django ORM query will be:

```
emp = employees.objects.filter(name='Nitin')
```

**Limitation of Django ORM:**

- If the data is complex and consists of multiple joins using the SQL will be clearer.
- If Performance is a concern for your, ORM aren't your choice.
- Generally. Object-relation-mapping are considered good option to construct an optimized query, SQL has an upper hand when compared to ORM.

# How You Can Set Up The Database In Django?

To set up a database in Django, you can find its configurations in setting.py file that representing Django settings.

By default, Django uses SQLite database. It is easy for Django users because it doesn't require any other type of installation.

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.sqlite3',
      'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
   }
}
```

# How You Can Set Up The Database In Django?

In the case of other database you have to the following keys in the DATABASE 'default' item to match your database connection settings.
**Engines**: you can change database by using 'django.db.backends.sqlite3' , 'django.db.backeneds.mysql', 'django.db.backends.postgresql_psycopg2', 'django.db.backends.oracle' and so on

```
'ENGINE': 'django.db.backends.postgresql_psycopg2',
```

Now we should replace the above code with our connection credentials to Mysql. The updated code should look like the code below.

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'NAME': 'helloworld',
    'USER': '<yourname>',
    'PASSWORD': 'password',
    'HOST': 'localhost',
    'PORT': '',
  }
}
```

# What do you mean by the CSRF Token?

- **CSRF** stands for **Cross Site Request Forgery.**
- The **csrf_token** is used for protection against Cross-Site Request Forgeries.
- This kind of attack takes place when a malicious website consists of a link, some JavaScript or a form whose aim is to perform some action on your website by using the login credentials of a genuine user.
- CSRF tokens can prevent CSRF attacks by making it impossible for an attacker to construct a fully valid HTTP request suitable for feeding to a victim user.
- A CSRF token is a unique, secret, unpredictable value that is generated by the server-side application and transmitted to the client in such a way that it is included in a subsequent HTTP request made by the client.
- When the later request is made, the server-side application validates that the request includes the expected token and rejects the request if the token is missing or invalid.

# What Is A QuerySet In Django?

- **QuerySet** is a collection of SQL queries.
- A QuerySet in Django is basically a collection of objects from our database.
- QuerySets are used by the Django ORM. When we use our models to get a single record or a group of records from the database, they are returned as QuerySets.
- It is comparable to a database **select** operation.

  E.g

```
users.objects.all()
users.objects.filter(name="nitin")
users.objects.get(id=3)
```

# What Is The Difference Between select_related & prefetch_related?

## select_related:

- Returns a QuerySet that will "follow" foreign-key relationships, selecting additional related-object data when it executes its query.
- This is a performance booster which results in a single more complex query but means later use of foreign-key relationships won't require database queries.

## prefetch_related:

- We use prefetch_related when we're going to get a set of things.
- That means forward ManyToMany and backward ManyToMany, ForeignKey.
- prefetch_related does a separate lookup for each relationship, and performs the "joining" in Python.

Though both the functions are used to fetch the related fields on a model but their functioning is bit different from each other.
In simple words, select_related uses a foreign key relationship, i.e. using join on the query itself while on the prefetch_related there is a separate lookup and the joining on the python side.

# What Is The Difference Between select_related & prefetch_related?

**Let's suppose the following 3 models in Django.**

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

class Entry(models.Model):
    blog = models.ForeignKey(Blog,
                             on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    authors = models.ManyToManyField(Author)
```

The following examples illustrate the difference between plain lookups and select_related() lookups. Here's standard lookup:

```
# Hits the database.
e = Entry.objects.get(id=5)

# Hits the database again to get the related Blog object.
b = e.blog
```

**And here's select_related lookup:**

```
# Hits the database.
e = Entry.objects.select_related('blog').get(id=5)

# Doesn't hit the database, because e.blog has been
prepopulated in the previous query.
b = e.blog
```

You can use select_related() with any queryset of objects

# What Is The Difference Between select_related & prefetch_related?

```
from django.db import models
class Country(models.Model):
   country_name = models.CharField(max_length=5)
class State(models.Model):
   state_name = models.CharField(max_length=5)
   country = model.ForeignKey(Country)
```

**select_related:**

```
>> states = State.objects.select_related('country').all()
>> for state in states:
...   print(state.state_name)
```

```
Query Executed
SELECT state_id, state_name, country_name FROM State INNER JOIN Country ON (State.country_id = Country.id)
```

**prefetch_related:**

```
>> country =
Country.objects.prefetch_related('state').get(id=1)
>> for state in country.state.all():
...   print(state.state_name)
```

```
Query Executed
SELECT id, country_name FROM country WHERE id=1;
SELECT state_id, state_name WHERE State WHERE
country_id IN (1);
```

# What Is The Difference Between Emp.object.filter(), Emp.object.get() & Emp.objects.all() in Django Queryset?

## Emp.objects.all():

In order to view all the items from your database, you can make use of the ‘all()’ function as mentioned below:

```
Users.objects.all()
```

where ***Users*** is some class that you have created in your models

# What Is The Difference Between Emp.object.filter(), Emp.object.get() & Emp.objects.all() in Django Queryset?

## Emp.object.filter() & Emp.object.get():

To filter out some element from the database, you either use the **get()** method or the **filter()** method as follows:

```
Users.objects.filter(name="Nitin")
Users.objects.get(name="Nitin")
```

Basically use get() when you want to get **a single unique object**, &
filter() when you want to get **all objects that match your lookup parameters**

get() throws an error if there's no object matching the query.
filter() will return an empty queryset.

get() raises **MultipleObjectsReturned** if more than one object was found. The **MultipleObjectsReturned** exception is an attribute of the model class.
get() raises a **DoesNotExist** exception if an object wasn't found for the given parameters. This exception is also an attribute of the model class.

# Which Companies Uses Django?

- ❏ Instagram
- ❏ Mozilla
- ❏ Spotify
- ❏ Pinterest
- ❏ Disqus
- ❏ Bitbucket
- ❏ Eventbrite
- ❏ Prezi
- ❏ Dropbox
- ❏ Youtube
- ❏ National Geographic

# How Static Files Are Defined In Django? Explain Its Configuration & It's Uses.

- Websites generally need to serve additional files such as images. Javascript or CSS. In Django, these files are referred to as "**static files**", Apart from that Django provides **django.contrib.staticfiles** to manage these static files.
- These files are created within the project app directory by creating a subdirectory named as **static**.
- Static files are stored in the folder called **static** in the Django app.

# How Static Files Are Defined In Django? Explain Its Configuration & It's Uses.

## How to configure static files?

- Ensure that django.contrib.staticfiles is added to your **INSTALLED_APPS**
- In your settings file. define STATIC_URL for ex.

```
STATIC_URL = '/static/'
```

- In your Django templates, use the static template tag to create the URL for the given relative path using the configured STATICFILES_STORAGE.

```
{% load static %}
<img src="{% static 'my_app/example.jpg' %}" alt="My image">
```

- Store your static files in a folder called static in your app. For example my_app/static/my_app/example.jpg

# How Static Files Are Defined In Django? Explain Its Configuration & It's Uses.

## How can you set up static files in Django?

There are three main things required to set up static files in Django:
1) Set the **STATIC_ROOT** setting to the directory from which you'd like to serve these files, e.g:

```
STATIC_ROOT = "/var/www/example.com/static/"
```

2) Run the collectstatic management command:

```
python manage.py collectstatic
```

This will copy all files from your static folders into the **STATIC_ROOT** directory.

3) set up a Static Files entry on the PythonAnywhere web tab

# What Is The Difference Between Flask, Pyramid And Django?

**Flask** is a "microframework" primarily build for a small application with simpler requirements. In flask, you have to use external libraries. Flask is ready to use.

**Pyramid** are build for larger applications. It provides flexibility and lets the developer use the right tools for their project. The developer can choose the database, URL structure, templating style and more. Pyramid is heavy configurable.

**Django** can also used for larger applications. It includes an ORM.

# What Is The Difference Between Flask, Pyramid And Django?

| Comparison Factor | Django | Flask |
|---|---|---|
| Project Type | Supports large projects | Built for smaller projects |
| Templates, Admin and ORM | Built-in | Requires installation |
| Ease of Learning | Requires more learning and practice | Easy to learn |
| Flexibility | Allows complete web development without the need for third-party tools | More flexible as the user can select any third-party tools according to their choice and requirements |
| Visual Debugging | Does not support Visual Debug | Supports Visual Debug |
| Type of framework | Batteries included | Simple, lightweight |
| Bootstrapping-tool | Built-it | Not available |

# What Is The Difference Between Flask, Pyramid And Django?

- Django is a high-level Python framework while Flask is a low-level Python Framework providing you with the minimum functionality, a server would require.
- Django comes with lots of built-in functionality like Django ORM, Admin Panel, Web-templating System, Inheritance, serialization while Flask comes with a development server, NoSQL support, support for unit testing, which are already there in Django.
- Flask is more customizable than Django as Flask comes with no predefined structure or scaffold while Django's file structure is fixed.
- Flask settings are user made and can be altered completely by the user. Django settings are not customizable to that degree, it has variables where only values are modifiable.
- Flask has more speed than Django when it comes to processing requests but that comes without any APIs or functionality which Django gives you in-built.
- Flask is for the developers who want more flexibility on their website and don't need lots of built-in extra functions, while Django is for developers who want rapid development of their applications that can sustain dynamic changes to its environment.

# Explain Django Admin & Django Admin Interface.

## Django Admin Panel:

- Django admin panel is a kind of **graphical user interface** that is used for **administrative tasks.**
- Django comes with a fully customizable in-built admin interface.
- You get the quick setup of the admin panel to manage your data and to access it.
- To use a database table with the admin interface, we need to register the model in the admin.py file.
- The development process becomes faster and also it becomes easy for the developers to perform administrative activities.
- The application Django admin is imported from the **django.contrib package.**
- This imported application is also expected to get control by the corresponding organization hence it does not require an additional front end.

**The Django admin interface provides a number of advanced features like:**

- **Authorization access**
- **Managing multiple models**
- **Content management system**

# Explain Django Admin & Django Admin Interface.

## django-admin:

It is the command-line utility of Django for administrative tasks.
Using the django-admin you can perform a number of tasks some of which are listed Below:

- **django-admin version** - used to check your Django version.
- **django-admin check** - used to inspect the entire Django project for common problems.
- **django-admin runserver** - Starts a light-weight Web server on the local machine for development. The default server runs on port 8000 on the IP address 127.0.0.1. You can pass a custom IP address and port number explicitly if you want.
- **django-admin startapp** - Creates a new Django app for the given app name within the current directory or at the given destination.
- **django-admin startproject** - Creates a new Django project directory structure for the given project name within the current directory or at the given destination.
- **django-admin test** - Runs tests for all installed apps.
- **django-admin testserver** - Runs a Django development server (which is also executed via the runserver command) using data from the given fixture(s).

# Explain Django Admin & Django Admin Interface.

- **django-admin changepassword** - offers a method to change the user's password.
- **django-admin createsuperuser** - Creates a user account with all permissions(also known as superuser account).
- **django-admin showmigrations** - Shows all migrations present in the project.
- **django-admin makemigrations** - Generates new migrations as per the changes detected to your models.
- **django-admin migrate** - Executes SQL commands after which the database state with the current set of models and migrations are synchronized.
- **django-admin sqlmigrate** - Prints the SQL statement for the named migration.
- **django-admin inspectdb** - It generates django models from the existing database tables.
- **django-admin sqlflush** - Prints the SQL statements that would be executed for the flush command mentioned above.
- django-admin shell - Starts the Python interactive interpreter.
- django-admin dumpdata - Used to the dumpdata from the database.
- django-admin flush - Flush all values from the database and also re-executes any post-synchronization handlers specified in the code.
- django-admin loaddata - loads the data into the database from the fixture file.
- django-admin makemessages - Used for translation purpose and it generates a message file too.

# Explain Django Admin & Django Admin Interface.

- django-admin sqlsequencereset - output the SQL queries for resetting sequences for the given app name(s).
- django-admin squashmigrations - Squashes a range of migrations for a particular app_label.
- django-admin remove_stale_contenttypes - removes stale content types (from deleted models) in your database.
- django-admin sendtestemail - This is used to confirm email sending through Django is working by sending a test email to the recipient(s) specified.
- django-admin help - used to display usage information and a list of the commands provided by each application.
- django-admin compilemessages - Compiles .po files created by makemessages to .mo files for use with the help of built-in gettext support.
- django-admin createcachetable - Creates the cache tables for use in the database cache backend.
- django-admin dbshell - Runs the command-line client for the database engine specified in your ENGINE setting(s), with the connection parameters (USER, PASSWORD, DB_NAME, USER etc.) specified settings file.
- django-admin diffsettings - Shows the difference between the existing settings file and Django's default settings.
- django-admin clearsessions - Can be used to clean out expired sessions or as a cron job.

# What Databases Are Supported By Django?

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Apart from these, Django also supports databases such as ODBC, Microsoft SQL Server, IBM DB2, SAP SQL Anywhere, and Firebird using third-party packages.

Among these the best-suited database is ***PostgreSQL***.

***Note: Officially Django doesn't support any No-SQL databases.***

# What Are The Advantages And Disadvantages Of Using Django?

## Advantages Of Django:

- Django is a Python's framework which is easy to learn.
- Django follows the DRY or the Don't Repeat Yourself Principle which means, one concept or a piece of data should live in just one place
- Django Offers Better CDN connectivity and Content Management
- Django is a Batteries Included Framework
- Django Offers Rapid-development
- Django is highly Scalable
- Django provide high Security
- Django facilitates you to divide code modules into logical groups to make it flexible to change.
- Django provides auto-generated web admin to make website administration easy.
- Django provides template system to define HTML template for your web page to avoid code duplication.
- Django enables you to separate business logic from the HTML.

# What Are The Advantages And Disadvantages Of Using Django?

## Disadvantages of Django:

- ❏ Django is Monolithic. You must know the full system to work with it.
- ❏ Django's monolithic size makes it unsuitable for smaller projects
- ❏ Everything must be explicitly defined due to a lack of convention.
- ❏ Django's modules are bulky.
- ❏ Django is completely based on Django ORM.
- ❏ Components are deployed together.

# What Is The Django Shortcut Method To More Easily Render An HTML Response?

Django shortcut method to more easily render an html response is:

" **render_to_response**"

# What Is The Difference Between Authentication And Authorization?

**Authentication** - Who Are You?
**Authorization** - What Permissions Do You Have?

**Authentication** is the process of verifying who someone is, whereas **Authorization** is the process of verifying what specific applications, files, and data a user has access to.

**Authentication** is the process or action of verifying the identity of a user or process.

# What Is The Difference Between Authentication And Authorization?

| S.No | Authentication | Authorization |
|---|---|---|
| 1. | In authentication process, the identity of users are checked for providing the access to the system. | While in authorization process, person's or user's authorities are checked for accessing the resources. |
| 2. | In authentication process, users or persons are verified. | While in this process, users or persons are validated. |
| 3. | It is done before the authorization process. | While this process is done after the authentication process. |
| 4. | It needs usually user's login details. | While it needs user's privilege or security levels. |
| 5. | Authentication determines whether the person is user or not. | While it determines **What permission do user have?** |

*Source- https://www.geeksforgeeks.org/difference-between-authentication-and-authorization/*

# What Is The Difference Between Authentication And Authorization?

## What Is Token Based Authentication System ?

A token based authentication system is a security system that authenticates the users who attempt to log in to a server, a network, or some other secure system, using a security token provided by the server

# What is django.shortcuts.render function?

- When a view function returns a webpage as **HttpResponse** instead of a simple string, we use **render**().
- Render function is a shortcut function that lets the developer easily pass the **data dictionary** with the template.
- This function then combines the template with a data dictionary via templating engine.
- Finally, this render() returns as HttpResponse with the rendered text, which is the data returned by models.
- Thus, Django render() bypasses most of the developer's work and lets him use different template engines.
- The basic syntax:

```
render(request, template_name, context=None, content_type=None, status=None, using=None)
```

The request is the parameter that generates the response.
The template name is the HTML template used.
The context is a dict of the data passed on the page from the python.
You can also specify the content type,
The status of the data you passed,
And the render you are returning.

# Explain Q objects in Django ORM?

**Q object django.db.models.Q** is an object to encapsulate a collection of keyword arguments specified as FIELD LOOKUPS.
**Q objects** are used to write complex queries, as in filter() functions just "AND" the conditions while if you want to "OR" the conditions you can use Q objects.

**Let's see an example:**

```
from django.db import models
from django.db.models import Q
```

```
Models.objects.get( Q(question__startswith='When'), Q(answer__startswith='On') | Q(answer__startswith='At') )
```

[Q Objects can be combined with the help of the | and & operators to get a new Q Object]

This is equivalent to the following SQL WHERE Clause:

```
SELECT * FROM Model WHERE question LIKE 'When%' And (answer="On%" OR answer="At%")
```

# What Is The Significance Of manage.py File In Django?

- The manage.py file is automatically generated whenever you create a project.
- This is basically a command-line utility that helps you to interact with your Django project in various ways.
- It does the same things as django-admin but along with that, it also sets the DJANGO_SETTINGS_MODULE environment variable in order to point to your project's settings.
- When Django server is started, the manage.py file searches for settings.py file, which contains information of all the applications installed in the project, middleware used, database connections and path to the main urls config.

# What Is The Use Of The “include” Function In The urls.py File In Django?

- As in Django there can be many apps, each app may have some URLs that it responds to.
- Rather than registering all URLs for all apps in a single urls.py file, each app maintains its own urls.py file, and in the project’s urls.py file we use each individual urls.py file of each app by using the include function.

# What Is The Use Of The “include” Function In The urls.py File In Django?

## Example:

**nitman -- urls.py**

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('nitapp/', include('nitapp.urls')),
    path('myapp/', include('myapp.urls')),
]
```

**nitapp -- urls.py**

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index),  # nitapp homepage
]
```

**myapp -- urls.py**

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index),  # myapp homepage
]
```

# What Does “{% include %}” Does In Django?

- It loads a template and renders it with the current context.
- This is a way of “including” other templates within a template.
- The template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.

**Syntax**: `{% include "template_name.html" %}`

**Example**: `{% include "poll/base.html" %}`

# What Does "{% include %}" Does In Django?

Now create a view through which we will access the template,
In nitman/views.py,

```
# import Http Response from django
from django.shortcuts import render

# create a function
def render_view(request):
    # return response
    return render(request, "nitman.html")
```

Create a url path to map to this view. In nitman/urls.py,

```
from django.urls import path

# importing views from views.py
from .views import render_view

urlpatterns = [
    path('', render_view), ]
```

Now we will create three templates to demonstrate include tag. Create a base template in nitman.html,

```
<html>
    <!-- Include header -->
    {% include "component1.html" %}
    <h4>Body Here</h4>
    <!-- Include Footer -->
    {% include "component2.html" %}
</html>
```

Create two components in templates/component1.html

```
<!-- component1.html -->
<h2> Header Here </h2>>
```

and templates/component2.html

```
<!-- component2.html -->
<h4>Footer here</h4>
```

OUTPUT:
Header Here
Body Here
Footer here

# What Is Django Rest Framework(DRF)?

- Django Rest Framework (DRF) is a powerful module for building web APIs.
- The Django Rest Framework (DRF) is a framework that helps you quickly create RESTful APIs.
- It's very easy to build model-backed APIs that have authentication policies and are browsable.
- RESTful APIs are perfect for web applications since they use low bandwidth and are designed such that they work well with communications over the Internet like GET, POST, PUT, etc.
- DRF is especially useful if we have an existing Django web application and we wish to quickly generate an API for it.

The following are the significant reasons that are making REST framework perfect choice:

1. Web browsable API
2. Serialization
3. Authentication policies
4. Extensive documentation and excellent community support.
5. Perfect for web apps since they have low bandwidth.

# What Is A Middleware In Django?

- Middleware is something that executes **between the request and response.**
- In simple words, you can say it acts as a bridge between the request and response.
- Middleware in the Django framework is the component that operates on request and transfers it to the view and before passing it to the template engine, it starts operating on a response.
- Django provides various built-in middleware and also allows us to write our own middleware.

```
// settings.py
        MIDDLEWARE = [
          'django.middleware.security.SecurityMiddleware',
          'django.contrib.sessions.middleware.SessionMiddleware',
          'django.middleware.common.CommonMiddleware',
          'django.middleware.csrf.CsrfViewMiddleware',
          'django.contrib.auth.middleware.AuthenticationMiddleware',
          'django.contrib.messages.middleware.MessageMiddleware',
          'django.middleware.clickjacking.XFrameOptionsMiddleware',
        ]
```

# What Is A Middleware In Django?

**Some usage of Middlewares in Django is:**

- ❏ Session management,
- ❏ Use authentication
- ❏ Cross-site request forgery protection(CSRF)
- ❏ Content Gzipping

# What Is Sessions In Django?

- Sessions are fully supported in Django.
- Using the session framework, you can easily store and retrieve arbitrary data based on the per-site-visitors.
- This framework basically stores data on the server-side and takes care of sending and receiving cookies.
- These cookies consist of a session ID but not the actual data itself unless you explicitly use a cookie-based backend.
- A session is a mechanism to store information on the server side during the interaction with the web application.
- By default, session stores in the database and also allows file-based and cache based sessions.

# What Are Django Signals?

- Django consists of a signal dispatcher that helps allow decoupled applications to get notified when actions occur elsewhere in the framework.
- Django provides a set of built-in signals that basically allow senders to notify a set of receivers when some action is executed.
- They're especially useful when many pieces of code may be interested in the same events.

**Two important parameters in signals are as follows:**

**Receiver**: It specifies the callback function which connected to the signal.
**Sender**: It specifies a particular sender from where a signal is received.

Two key elements the Senders and the receivers are in the signals machinery. The sender is responsible to dispatch a signal, and the receiver is the one who receives this signal and then performs something.

# What Are Django Signals?

**List of built-in signals in the models:**

| Signals | Description |
|---|---|
| django.db.models.pre_init & django.db.models.post_init | Sent before or after a models's __init__() method is called |
| django.db.models.signals.pre_save & django.db.models.signals.post_save | Sent before or after a model's save() method is called |
| django.db.models.signals.pre_delete & django.db.models.signals.post_delete | Sent before or after a models' delete() method or queryset delete() method is called |
| django.db.models.signals.m2m_changed | Sent when a ManyToManyField is changed |
| django.core.signals.request_started & django.core.signals.request_finished | Sent when an HTTP request is started or finished |

# What Is Context In Django?

- A context in Django is a dictionary, in which keys represent variable names and values represent their values. This dictionary (context) is passed to the template which then uses the variables to output the dynamic content.
- A context is a variable name -> variable value mapping that is passed to a template.
- Context processors let you specify a number of variables that get set in each context automatically – without you having to specify the variables in each render() call.

# What Are Django Exceptions?

**Django Exceptions:** An exception is an abnormal event that leads to program failure. Django uses its exception classes and python exceptions as well to deal with such situations.

**1.) Django Exception classes:** We define Django core exceptions in "Django.core.exceptions".

**AppRegistryNotReady:** This class raises for using models before loading the app process.

**ObjectDoesNotExist:** It's a base class for DoesNotExist exceptions.

**EmptyResultSet:** This exception arises when the query fails to return results.

**FieldDoesNotExist:** When the requested file does not exist, this exception arises.

**MultipleObjectsReturned:** It raises by the query multiple objects returned when we expect only one object.

**SuspiciousOperation**: It raises when the user has performed some operation, which is considered suspicious from a security perspective.

**PermissionDenied:** It arises when a user does not have permission to execute a specific action requested.

**ViewDoesNotExist:** When the requested view does not exist, this exception raises.

**MiddlewareNotUsed:** When there is no middleware in server configuration, this exception arises.

**ImproperlyConfigured:** When Django configuration is improper, this exception arises.

**FieldError:** When there is a problem with the model field, this exception arises.

**ValidationError:** It raises when data validation fails.

*Source: https://www.javatpoint.com/django-exceptions*

# What Are Django Exceptions?

**2) Django URL Resolver Exceptions:** These exceptions are defined in django.urls module.
**Resolver404**: This exception raised when the path passed to resolve() function does not map to a view.
**NoReverseMatch**: It is raised when a matching URL in your URLconf cannot be identified based on the parameters supplied.

**3) Django Database Exceptions:** The following exceptions are defined in django.db module.
**DatabaseError**: It occurs when the database is not available.
**IntegrityError**: It occurs when an insertion query executes.
**DataError**: It raises when data related issues come into the database.

**4) Django Http Exceptions:** The following exceptions are defined in django.http module.
**UnreadablePostError**: It is raised when a user cancels an upload.

**5) Django Transaction Exceptions:** The transaction exceptions are defined in django.db.transaction.
**TransactionManagementError**: It is raised for any and all problems related to database transactions.

*Source: https://www.javatpoint.com/django-exceptions*

# What Happens If MyObject.objects.get() Is Called With Parameters That Do Not Match An Existing Item In The Database?

```
MyObject.objects.get(id="nitin")
```

- Basically use get() when you want to get **a single unique object**, and get() throws an error if there's no object matching the query.
- If there are no results that match the query, get() will raise a **DoesNotExist exception**.
- If more than one item matches the given get() query then it'll raise **MultipleObjectsReturned**, which is also an attribute of the model class itself.

# How To Make A Variable Available To All The Templates?

What you want is a context processor, and it's very easy to create one. Assuming you have an app named custom_app, follow the next steps:

- Add custom_app to INSTALLED_APPS in settings.py (you've done it already, right?);
- Create a context_processors.py into custom_app folder;
- Add the following code to that new file:
  def categories_processor(request): categories = Category.objects.all() return {'categories': categories}
- Add context_processors.py to TEMPLATE_CONTEXT_PROCESSORS in settings.py
  TEMPLATE_CONTEXT_PROCESSORS +=
  ("custom_app.context_processors.categories_processor", )

And now you can use {{categories}} in all the templates

**As of Django 1.8**

To add a TEMPLATE_CONTEXT_PROCESSORS, in the settings you must add the next code:

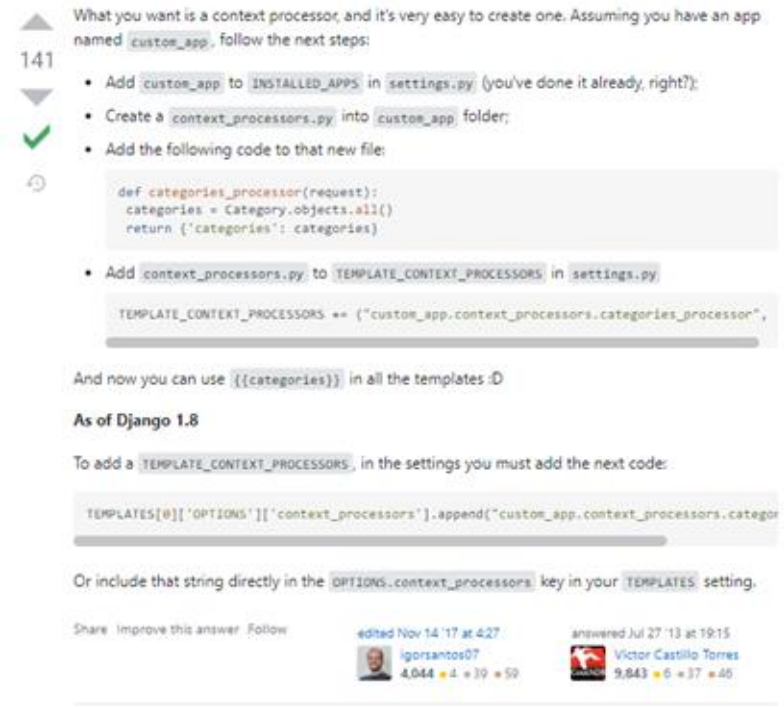
```
TEMPLATES[0]['OPTIONS']['context_processors'].append("custom_app.context_processors.categories_processor")
```

Or include that string directly in the OPTIONS.context_processors key in your TEMPLATES setting.



*Source:- https://stackoverflow.com/questions/17901341/django-how-to-make-a-variable-available-to-all-templates*

# How To Make A Variable Available To All The Templates?



What you want is a context processor, and it's very easy to create one. Assuming you have an app named `custom_app`, follow the next steps:

- Add `custom_app` to `INSTALLED_APPS` in `settings.py` (you've done it already, right?);
- Create a `context_processors.py` into `custom_app` folder;
- Add the following code to that new file:

```
def categories_processor(request):
 categories = Category.objects.all()
 return {'categories': categories}
```

- Add `context_processors.py` to `TEMPLATE_CONTEXT_PROCESSORS` in `settings.py`

```
TEMPLATE_CONTEXT_PROCESSORS += ("custom_app.context_processors.categories_processor",
```

And now you can use `{{categories}}` in all the templates :D

**As of Django 1.8**

To add a `TEMPLATE_CONTEXT_PROCESSORS`, in the settings you must add the next code:

```
TEMPLATES[0]['OPTIONS']['context_processors'].append("custom_app.context_processors.categor
```

Or include that string directly in the `OPTIONS.context_processors` key in your `TEMPLATES` setting.

Share Improve this answer Follow

edited Nov 14 '17 at 4:27 — igorsantos07 — 4,044 • 4 • 39 • 59

answered Jul 27 '13 at 19:15 — Victor Castillo Torres — 9,843 • 6 • 37 • 46

What you want is a context processor, and it's very easy to create one. Assuming you have an app named custom_app, follow the next steps:

- Add custom_app to INSTALLED_APPS in settings.py (you've done it already, right?);
- Create a context_processors.py into custom_app folder;
- Add the following code to that new file:
  def categories_processor(request): categories = Category.objects.all() return {'categories': categories}
- Add context_processors.py to TEMPLATE_CONTEXT_PROCESSORS in settings.py
  TEMPLATE_CONTEXT_PROCESSORS += ("custom_app.context_processors.categories_processor", )

And now you can use {{categories}} in all the templates

**As of Django 1.8**

To add a TEMPLATE_CONTEXT_PROCESSORS, in the settings you must add the next code:

```
TEMPLATES[0]['OPTIONS']['context_processors'].append("custom_app.context_processors.categories_processor")
```

Or include that string directly in the OPTIONS.context_processors key in your TEMPLATES setting.



*Source:- https://stackoverflow.com/questions/17901341/django-how-to-make-a-variable-available-to-all-templates*

# Why Does Django Use Regular Expressions To Define URLs? Is It Necessary To Use Them?

- Django uses a very powerful format for storing URLs, that is regular expressions.
- RegEx or regular expression is the format for sophisticated string searching algorithms.
- It makes the searching process faster. Although it's not necessary to use RegEx when defining URLs.
- They can be defined as normal string also, Django server should still be able to match them, but when you need to pass some data from the user via URL, then RegEx is used.
- The RegEx also makes much cleaner URLs then other formats.

# Why Does Django Use Regular Expressions To Define URLs? Is It Necessary To Use Them?

Example:

```
urlpatterns = [
  path('articles/2003/', views.special_case_2003), #1
  path('articles/<int:year>/', views.year_archive),#2
  path('articles/<int:year>/<int:month>/', views.month_archive), #3
  path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),#4
]
```

***Links:***
***https://raturi.in/blog/designing-django-urls-best-practices/***

# What Is The Difference Between Django OneToOneField & ForeignKey Field?

**ForeignKey Field:** A many-to-one relationship. Requires two positional arguments: the class to which the model is related and the on_delete option.

**OneToOneField**: A one-to-one relationship. Conceptually, this is similar to a ForeignKey with unique=True, but the "reverse" side of the relation will directly return a single object.

***Links:***
***https://stackoverflow.com/questions/5870537/whats-the-difference-between-django-onetoonefield-and-foreignkey***

# Explain Django Field Class And Its Types.

- **'Field'** is basically an abstract class that actually represents a column in the database table.
- The Field class, is in turn, a subclass of RegisterLookupMixin.
- In Django, these fields are used to create database tables (db_type()) which are used to map Python types to the database using get_prep_value() and vice versa using from_db_value() method.
- Therefore, fields are fundamental pieces in different Django APIs such as models and querysets.

**The Django field class types specify:**

- The column type, which tells the database what kind of data to store (e.g. INTEGER, VARCHAR, TEXT).
- The default HTML widget to use when rendering a form field (e.g. <input type="text">, <select>).
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

# Explain How You Can Use File Based Sessions?

To use the file-based sessions, you need to set the SESSION_ENGINE settings to **"django.contrib.sessions.backends.file"**

# What Is Jinja Templating?

Jinja Templating is a very popular templating engine for Python, the latest version in the market is Jinja 2.

There are some features of Jinja templating that make it a better option than the default template system in Django.

- Sandbox Execution – This is like a sandbox or a protected framework for automating the testing process.
- HTML Escaping – Jinja 2 provides automatic HTML Escaping, as <, >, & characters have special values in templates and if used as regular text, these symbols can lead to XSS Attacks which Jinja deals with automatically.
- Template Inheritance
- Generates HTML templates much faster than default engine
- Easier to debug, compared to default engine.

# What Is Serialization In Django?

**Serialization** is the process of converting Django models into other formats such as XML, JSON, etc.

***Links:***
***https://docs.djangoproject.com/en/4.0/topics/serialization/***
***https://opensource.com/article/20/11/django-rest-framework-serializers***

# What Are Generic Views?

- When building a web application there are certain kind of views that we build again and again, such as a view that displays all records in the database (e.g., displaying all books in the books table), etc.
- These kinds of views perform the same functions and lead to repeated code.
- To solve this issue, Django uses class-based generic views.
- When using generic views, all we have to do is inherit the desired class from django.views.generic module and provide some information like model, context_object_name, etc

*Source: https://www.tutorialspoint.com/django/django_generic_views.htm*

# What Is Mixin?

- **Mixin** is a type of multiple inheritance wherein you can combine behaviors and attributes of more than one parent class.
- Mixins provide an excellent way to reuse code from multiple classes.
- There are two main situations where mixins are used: to provide a lot of optional features for a class and to use one particular feature in a lot of different classes

# What Is Mixin?

**For example,** generic class-based views consist of a mixin called TemplateResponseMixin whose purpose is to define render_to_response() method.
When this is combined with a class present in the View, the result will be a TemplateView class.

One **drawback** of using these mixins is that it becomes difficult to analyze what a child class is doing and which methods to override in case of its code being too scattered between multiple classes.

**Django provides a number of mixins that provide more discrete functionality.**
Different type of mixins are -
**ContextMixin** - A dictionary to include in the context and is a convenient way of specifying the simple context in as_view().
**TemplateResponseMixin** - Given a suitable context, TemplateResponseMixin provides a mechanism to construct a TemplateRespons and the template to use is configurable and can be further customized by a subclass.
**SingleObjectMixin** - SingleObjetMixin provides a mechanism for looking up an object associated with the current HTTP request.
**SingleObjectTemplateMixin** - SingleObjetTemplateMixin performs template base response rendering for view that operate upon a single object instance.
**MutlipleObjectMixin** - MultipleObjectMixin used to display list of objects

# Explain The Caching Strategies In The Django?

- Caching refers to the technique of storing the output results when they are processed initially so that next time when the same results are fetched again, instead of processing again those already stored results can be used, which leads to faster accessing as well us less resource utilization.
- Django provides us with a robust cache system that is able to store dynamic web pages so that these pages don't need to be evaluated again for each request.

# Explain The Caching Strategies In The Django?

**Few caching strategies that are available in Django are as follows:**

**Memcached Caching:** The gold standard for caching. An in-memory service that can return keys at a very fast rate. Not a good choice if your keys are very large in size
**Redis**: A good alternative to Memcached when you want to cache very large keys (for example, large chunks of rendered JSON for an API)
**Dynamodb Caching:** Another good alternative to Memcached when you want to cache very large keys. Also scales very well with little IT overhead.
**Localmem Caching:** Only use for local testing; don't go into production with this cache type.
**Database Caching:** It's rare that you'll find a use case where the database caching makes sense. It may be useful for local testing, but otherwise, avoid it.
**File system Caching:** Can be a trap. Although reading and writing files can be faster than making SQL queries, it has some pitfalls. Each cache is local to the application server (not shared), and if you have a lot of cache keys, you can theoretically hit the file system limit for the number of files allowed.
**Dummy Caching:** A great backend to use for local testing when you want your data changes to be made immediately without caching. Be warned: permanently using dummy caching locally can hide bugs from you until they hit an environment where caching is enabled.
Content delivery networks