PROFESSIONAL DJANGO

CODE OF CONDUCT

PREPARED FOR

FSD X1

PREPARED BY
Durjoiy Mistry
Data Science and ML Enigneer, IARE

Table of Content

Coding Stucture

Django is a popular web development framework that offers a range of tools and features for building high-quality web applications. When working on a Django project, it's important to follow professional coding standards to ensure that your code is readable, maintainable, and consistent with industry best practices. In this document, we'll outline some of the key professional coding standards for Django development.

1. Project Structure:

When setting up your Django project, it's important to follow a consistent and organized project structure. This will make it easier to navigate and understand your code, as well as help other developers who may need to work on the project in the future.

- Use a modular approach: Organize your code into separate modules that handle specific functionality, such as models, views, templates, and forms.
- Follow the Django project layout convention: Use the recommended project structure as outlined in the official Django documentation, which includes directories for apps, static files, templates, and media files.
- Use descriptive names: Give your modules, directories, and files clear and descriptive names that accurately reflect their purpose and content.

- Avoid hard-coding paths: Use Django's built-in path-related functions, such as `os.path` and `os.path.join()`, to avoid hard-coding file and directory paths in your code.

2. Code Style and Formatting:

Consistent code style and formatting is essential for making your code readable, maintainable, and easy to understand. In Django, you should follow the PEP 8 guidelines for Python code, as well as some additional Django-specific conventions:

- Use consistent indentation: Use four spaces for indentation, and avoid using tabs.
- Follow naming conventions: Use descriptive and meaningful names for your variables, functions, classes, and modules, and follow the conventions outlined in the official Django documentation.
- Use docstrings: Include docstrings for all classes, functions, and methods, following the conventions outlined in PEP 257.
- Use consistent formatting: Use a consistent formatting style for your code, including spacing, line breaks, and parentheses placement.
- Use comments sparingly: Use comments only when necessary to explain complex or unusual code, and avoid using comments to explain obvious code.

3. Database and Models:

In Django, models define the structure and behavior of your application's data. To ensure that your database and models are well-designed and efficient, you should follow some best practices:

- Use the ORM: Use Django's built-in Object-Relational Mapping (ORM) system to interact with the database, rather than writing raw SQL queries.
- Use migrations: Use Django's built-in migration system to manage changes to your database schema over time.
- Keep models simple: Keep your models as simple and focused as possible, and avoid adding unnecessary fields or methods.
- Follow naming conventions: Use descriptive and consistent names for your models and fields, and follow the conventions outlined in the official Django documentation.
- Use database indexes: Use database indexes to improve query performance, especially for frequently queried fields.

4. Views and Templates:

Views and templates are the backbone of the user interface in Django. To ensure that your views and templates are well-designed and user-friendly, you should follow some best practices:

- Keep views simple: Keep your views as simple and focused as possible, and avoid adding unnecessary logic or functionality.

- Use class-based views: Use Django's built-in class-based views to simplify your code and reduce duplication.
- Use context data: Use context data to pass data from your views to your templates, and avoid adding complex logic to your templates.
- Use template tags and filters: Use Django's built-in template tags and filters to add complex logic and functionality to your templates.
- Follow naming conventions: Use descriptive and consistent names for your views, templates, and template tags, and follow the conventions

Naming Conventions

Project:

- Rules:
    - No Space
    - All lower case
- Example:
    - mysite

Desctiption:
In Django, project names should be all lowercase, with no spaces or special characters, and separated by underscores. For example, if you are building a project for a blog website, you might call your project myblogsite.

It's important to choose a name that is unique and descriptive of the project's purpose. Avoid using generic names like project or django, as these names could conflict with other projects and make it difficult to identify the purpose of your project.

Additionally, it's a good practice to include the word "project" in your project name to make it clear that it is a Django project. For example, myblogsite_project or blog_project are both good choices.

Remember to also choose a name that is easy to type and remember, as you will be using it frequently throughout the development process.

Apps:

- Rules:
    - All lower cased
    - Plural most of the time
    - Underscores can be used if it improves readability.
- Example:
    - products
    - articles
    - users
    - orders
    - oauth2_provider
    - rest_framework
    - polls

Description:

In Django, app names should be lowercase and descriptive, using underscores to separate words. For example, if you are building a blog application, you might call your app blog.
It's important to choose a name that is unique and descriptive of the functionality of the app. Avoid using generic names like common or utils, as these names could conflict with other apps and make it difficult to identify the purpose of your app.

Additionally, Django recommends using a pluralized name for apps that contain multiple models or views. For example, if your blog app includes models for posts, comments, and categories, you might call it blogs instead of blog.
Remember to also include your app in the INSTALLED_APPS list in your project's settings.py file to register it with Django.

Models:

- Rules:
    - CamelCase.
    - Singular number
    - Underscores can be used if it improves readability.
- Example:
    - Product
    - BlogPost

Description:

In Django, model names should be singular and in CamelCase, with the first letter capitalized. For example, a model for a blog post would be named BlogPost.
Additionally, Django recommends using verbose model names that describe the object being represented. For example, instead of naming a model Post, you could use BlogPost or NewsArticle.
It's also important to name your models in a way that is intuitive and easy to understand. For example, if you have a model that represents a user profile, you could name it UserProfile or UserInformation.
Finally, it's a good practice to use pluralized names for related models that represent a collection of objects. For example, if you have a model for comments, you might call it Comment, and the related model that represents a collection of comments might be called Comments.

Forms:

- Rules:
    - CamelCase.
    - Model Name+Action+Form
- Example:
    - ProductCreateForm
    - BlogPostUpdteForm

Description:

In Django, form names should be singular and in CamelCase, with the first letter capitalized. For example, a form for creating a blog post would be named BlogPostForm.

It's also important to name your forms in a way that is intuitive and easy to understand. For example, if you have a form that allows users to submit contact information, you could name it ContactForm.

Additionally, if you have multiple forms for a single model, you can use a descriptive name for the form that specifies its purpose. For example, if you have a form for editing a blog post and another form for creating a new blog post, you might call them BlogPostEditForm and BlogPostCreateForm, respectively.

Remember to also use the ModelForm class when creating forms for models, as it provides a lot of functionality out of the box and can help reduce boilerplate code.

Functions:

- Rules:
    - All lower case.
    - Separated by underscore
    - Must contain desctiopton
- Example:
    - product_create(request):
      “””

      This function is used to create new product.

      ”””

    - blog_update(request)

      “””

      This function is used to update blog.

      ”””

Description:

In Django, function names should be lowercase, with words separated by underscores. Function names should be descriptive and indicate the purpose of the function. For example, if you have a function that retrieves a list of blog posts, you might name it get_blog_posts.

It's important to choose function names that are clear and easy to understand. Avoid using abbreviations or acronyms

unless they are widely understood in the context of your application.

If you have a function that is related to a specific model or view, you can include the name of the model or view in the function name. For example, if you have a view that displays a list of blog posts, you might name the function blog_post_list.

Finally, it's a good practice to follow PEP 8 guidelines for Python code, which recommend using lowercase with underscores for function names. This convention helps make your code more readable and consistent with other Python code.

PEP 8 guidelines:

PEP 8 is the official style guide for Python code, and it provides guidelines for writing clean, consistent, and easy-to-read code. Here are some of the key guidelines from PEP 8:

1. Use 4 spaces for indentation. Do not use tabs.
2. Limit lines to a maximum of 79 characters.
3. Use lowercase for function names and variable names, and separate words with underscores.
4. Use CamelCase for class names.
5. Use uppercase for constants.
6. Use a single space around operators and after commas, but no space directly inside parentheses or brackets.
7. Use a space before and after the # character in comments.
8. Use docstrings to document functions, modules, and classes.
9. Use blank lines to separate functions, classes, and logical sections of code.

10. Use consistent naming conventions for related items, such as functions that perform similar tasks or variables that are related.

Remember that PEP 8 is a guideline, not a strict set of rules. While it's important to follow these guidelines for consistency and readability, it's also important to use your own judgment and common sense when writing code.

Variables:

- Rules:
  - All lower case.
  - Name must be meaningful
- Example:
  - products_list = [] # Contains the list of products
  - customer # stores the current customer object

Description:

In Django, variable names should be lowercase, with words separated by underscores. Variable names should be descriptive and indicate the purpose of the variable. For example, if you have a variable that stores a list of blog posts, you might name it blog_posts.

It's important to choose variable names that are clear and easy to understand. Avoid using abbreviations or acronyms unless they are widely understood in the context of your application.

If you have a variable that is related to a specific model or view, you can include the name of the model or view in the variable name. For example, if you have a view that displays a list of blog posts, you might name the variable blog_posts_list.

Additionally, when defining constants, you should use uppercase letters and underscores to separate words. For

example, if you have a constant that represents the number of posts per page, you might name it POSTS_PER_PAGE.
Finally, it's a good practice to follow PEP 8 guidelines for Python code, which recommend using lowercase with underscores for variable names. This convention helps make your code more readable and consistent with other Python code.

URLs:

- Rules:
  - All lower case.
  - Separated by hyphen (-) not underscore (_).
  - Must contain a name and should be called by that name from view
- Example:
  - path('detail-view/<str:pk>', views.TaskDetail.as_view(), name="task"),

Descrtipon:

In Django, URLs should follow a specific convention to make them easy to read, understand, and maintain. Here are some conventions to follow when writing URLs in Django:

1. Use lowercase letters and hyphens to separate words in the URL path. For example, if you have a page that displays a list of blog posts, you might use the URL /blog/posts/.
2. Use variable placeholders to capture dynamic parts of the URL path. For example, if you have a page that displays a specific blog post, you might use the URL /blog/posts/<int:pk>/, where <int:pk> captures the primary key of the blog post.
3. Use the name attribute to give each URL a unique name. This makes it easier to refer to the URL in your code and

templates. For example, you might name the URL /blog/posts/ as blog_posts.

4. Use the path() function for simple URL patterns and the re_path() function for more complex patterns that require regular expressions.
5. Use namespaces to group related URLs together. This helps prevent naming conflicts and makes it easier to organize your code. For example, if you have a blog app, you might namespace all of the URLs in the app as blog:.
6. Use the reverse() function to generate URLs dynamically based on their name and any required parameters. This helps make your code more maintainable and reduces the risk of errors.

Remember that good URL design can help make your Django application more readable, understandable, and maintainable. By following these conventions, you can create URLs that are easy to work with and help ensure the long-term success of your application.

In Django, URLs should use hyphens to separate words in the URL path, rather than underscores. This convention follows the general practice in web development of using hyphens in URLs, which is also recommended by search engines for SEO purposes.
Using hyphens in URLs makes them more readable and user-friendly, as well as easier to type and share. Additionally, using hyphens can help prevent confusion and errors when working with URLs in other contexts, such as email addresses or command-line interfaces.
While underscores are technically valid characters in URLs, they are less commonly used and can cause issues with some web servers and applications. Therefore, it's generally best to stick with hyphens when creating URLs in Django.

Imports:

- Rules:
  - All imports on top of the file.
  - Should be grouped by types.
- Example:
  - ```
    # future
    from __future__ import unicode_literals

    # standard library
    import json
    from itertools import chain

    # third-party
    import bcrypt

    # Django
    from django.http import Http404
    from django.http.response import (
            Http404,  HttpResponse,  HttpResponseNotAllowed,
    StreamingHttpResponse,
        cookie,
    )

    # local Django
    from .models import LogEntry

    # try/except
    try:
        import yaml
    except ImportError:
        yaml = None

    CONSTANT = 'foo'
    ```
    ```

In Django, import statements should be at the top of your file, below any module-level docstrings and above any other code. Here are some conventions to follow when writing import statements in Django:

1. Group import statements by type, and separate them with a blank line. For example, you might group all of your standard library imports together, followed by all of your third-party library imports, and finally your local project imports.
2. Use absolute imports, which means importing modules using the full package path rather than a relative path. For example, if you have a module called views in a package called blog, you should import it using from blog.views import MyView.
3. Avoid using wildcard imports, which import all of the names from a module. Instead, import only the specific names you need from a module.
4. Use descriptive names for imported modules and objects, and avoid using abbreviations or acronyms unless they are widely understood.
5. Use aliases for imported modules or objects when the name is too long or there is a naming conflict with another module or object. For example, you might import models from the blog package as bp_models to avoid a naming conflict with the built-in models module.
6. Use relative imports when importing modules from within the same package. This helps make your code more modular and easier to reuse.

Remember that good import practices can help make your code more readable, maintainable, and consistent with other Django code.