

# **CSE300 Software Engineering**

# **Learning Management System**

**Group 8**

# **Coding Standards Document**

[**Coding Style**](#_1sr6otp3ou09) **3**

[App Structure](#_hxaz5w8zk7am) 3

[View Style](#_py8t87563lkv) 3

[Model Style](#_h2jgq0pd3znf) 4

[Template Style](#_3vwx98yhmfwc) 4

[**Indentation**](#_ee5h5adoxesb) **5**

[Spaces, Tabs and Indentation](#_ei9x76wer3s) 5

[Formatting of Lines for Source File and Modules](#_t47smrpz871l) 5

[**Comments**](#_vol69v9no3zx) **6**

[Block Comments](#_mv17yg63ya5r) 6

[Inline Comments](#_ux609n9li9r) 6

[**Naming Conventions**](#_upiuthxgv7r) **7**

[Global Variables and Identifiers](#_g3ae39z0dfpz) 7

[Constants and Strings](#_n4ekfkq2hj10) 7

[Functions, Methods, Modules, and Class Names](#_hrhqtsefi7oy) 8

[Other Annotations](#_8ky29dbzc4j4) 8

[**Imports and Configuration**](#_ug5jhcmwuwv) **9**

[Standard Library](#_4r6nujluwcdh) 9

[Third Party Imports](#_tfor1iosms6o) 9

[Configuration and Authentication](#_inhsa583cucr) 10

[Error handling](#_6jvdwe4kemny) 10

[**HTML and CSS(Django Template)**](#_1xdfv32p11l1) **12**

[Style Rules](#_mkd0psa30r41) 12

[Formatting Rules](#_qov99jbnn28s) 13

[Convention](#_8gq12p1yuuq) 13

[Commenting](#_e63bxpuo73ux) 15

[Control Structures](#_d74zu1hzl1j1) 15

**Django**

# Coding Style

## App Structure

* + - Coding Standards are used to maintain uniformity in huge code bases with well-defined and standard style of coding.It reduces complexity and improves readability, reusability of the code. Due to this, it increases efficiency of the team members and to detect error easily.
    - The structure of the website was divided into individual page components, views, and model files and folders. It helped in differentiating the frontend and backend components having specific folders(templates, views, models, css, and static files).
    - This clean separation consists of minimal self-contained files which makes it easy to make changes and test the system. Different teams can handle their own set of folders to work on. The complex code makes debugging and maintenance expensive and difficult.

## View Style

* + - All the views are stored in a single file with comments and indentation as well as easily recognizable.
    - In Django views, we have consistently used the first parameter of every view function named as **request**.
    - View functions are named distinctly to avoid overwriting and redundancy.
    - They are designed in such a way that maximizes the reusability of the code.

## Model Style

* + - Model classes are in a proper order and are stored in a single file to manage easily. All the field names used are in lowercase, using underscores instead of camelCase.
    - All the field names are defined initially and followed by the class Meta with a blank line separating the class and field definition.
    - Each choice is defined as a list of tuples, with an all-uppercase name as a class attribute on the model for a given field of model. These are the best practice to design the model and reduce its complexity.

## Template Style

* + - The template contains all the frontend files of HTML/XML format with variables, which get replaced with values when the template is connected to the backend, and tags, which control the logic of the template.
    - Here, we have a Django template system consisting of tags which function similarly to some programming constructs but these are not simply executed as the corresponding Python code. They are limited but we can add our own extensions to the template language as needed.
    - Also including the template inheritance is a powerful way of cutting down on boilerplate in regular templates. It helps in building the base design for all the templates containing all the common elements. It defines blocks that child templates can override.

# Indentation

## Spaces, Tabs and Indentation

* + - Proper indentation is very important to increase the readability of the code. For making the code readable, coders should use proper indentation and White spaces with some spacing conventions.
    - There must be a space between the curly brackets and the tag contents, and also after giving a comma between two function arguments or variable passing or in tuples.
    - Each nested block must be properly spaced.
    - All braces should be on the same line and the code following the end of braces also starts from a new line.
    - This will help in identifying the sections way more easily and quickly in a huge codebase.

## Formatting of Lines for Source File and Modules

* + - Break long lines and wrap it up in sections using parentheses and indent continuation lines by a tab of 4 spaces. Include a trailing comma after the last import and put the closing parenthesis on its own line.
    - Use a single blank line between all the sections of the code, like the imports, functions, views, and any module level code, and use two blank lines above the first function or class.
    - Proper Indentation should be there for each nested block as well as at the beginning and at the end of each block in the program.

# Comments

## Block Comments

* + - Comments for all the code blocks and the statements increase the understandability of the code for other team members. It helps in synchronization and making changes in code easy for the team.
    - Block comments generally apply to code that follows them, and are indented to the same level as that code. Each line of a block comment should start with a # and a single space with the indented text inside it and the paragraphs inside a block comment are separated by a line containing a single #. This increases the clarity.

## Inline Comments

* + - Inline comments should be used sparingly and it should explain the code in the same line. It should always be separated by at least two spaces from the statement. They should always start with a # followed by a single space.
    - You should have a comment describing what the method actually does. This comment should appear after the def line.It is better to avoid the inline comments being unnecessary and distracting when it is stating the obvious things.

# 

# Naming Conventions

## Global Variables and Identifiers

* + - Avoid Global variables as much as possible. Overriding principle should be followed by using apt naming conventions of the names visible to the user which reflects usage rather than implementation.
    - Always avoid using single characters as variable/identifier names. Meaningful, descriptive and understandable variable names should be used which enhances the understandability of the reason for using it.
    - Local Variables and identifiers should always be named using camelCase lettering starting with small letters.
    - If an identifier is used for multiple purposes then it can lead to confusion to the reader. Also, it leads towards more difficulty during future enhancements of the project.

## Constants and Strings

* + - The constants are always defined on a module level at the beginning of code and they should be following convention of all capital letters with underscores separating words.
    - The string variables should always be initiated empty or with some value at the time of declaration itself.
    - Single line strings should be embedded by opening and closing double quotes on the same line. Similarly for multi liners, the "" that ends a multi line docstring should be on the next line.

## Functions, Methods, Modules, and Class Names

* + - Function names should follow lowercase convention, with words separated by underscores as it's necessary to improve the readability. They should be descriptive but short and must give the hint of what that function or method does.
    - Similarly, modules should have short and all-lowercase names whereas Class names should normally follow the CapWords convention.
    - Always use one leading underscore for instance variables and non-public methods and two leading underscores to invoke Python’s mangling rules. This will help in avoiding name clashes with the subclasses.
    - In the instance methods, one must use self as the first argument whereas for class methods, cls should be used for its first argument.

## Other Annotations

* + - The class and instance variables, module level variables and the local variables should follow a proper indentation having a single space after the colon. Before the colon, there should be no space.
    - Type annotations can be included in the form of comments in the particular code that needs to be backwards compatible.

# 

# 

# Imports and Configuration

## Standard Library

* + - It helps with existing inbuilt functions which reduce the development time and the complexity of the code.
    - Always avoid complex package layouts in Standard library code and maximize the use of absolute imports.
    - Also, avoid wildcard imports (from <module> import \*), as they make it unclear which names are present in the namespace, creating confusion for both readers and many automated tools as well as overwriting functions.
    - There must be a single blank line between each group of imports.

## Third Party Imports

* + - Imports should be on separate lines and not clustered altogether. They are always put at the top of the file, just following the module comments and docstrings, and it should be before declaration of module globals and constants.
    - It is always recommended to use absolute imports, as they are more readable and tend to be better behaved. If the import system is incorrectly configured, they gave a better error message compared to relative imports.

## Configuration and Authentication

* + - Modules should never use settings in general, stored in django.conf.settings at the top level. They are first imported and evaluated and then we can access it.
    - If any setting is accessed before the settings.configure line, this will not work. So it is best practice to import all the files at the top of the code, before accessing it. Good part is that settings configures itself automatically when they are accessed if it has not already been configured.
    - This module when imported, will cause the settings object to be configured. That means that the ability for third parties to import the module at the top level is incompatible with the ability to configure the settings object manually, or makes it very difficult in some circumstances.
    - We have used django.auth and django-session-timeout for authentication for a given period of time and then the session is timeout automatically.

## Error handling

* 1. In this, all the functions are designed in such a way that when they encounter an error condition, they always return a boolean response of 0 or 1 for simplifying the debugging.
  2. This helps in detecting errors on a previous basis, so it reduces the extra cost required by the software project.
  3. Many throw errors as a string – this entangles the error handling logic and the interoperability between modules. Whether you throw an exception or reject a promise – using only the built-in Error object will prevent loss of information and increase uniformity.
  4. Error handling logic should be encapsulated in a centralized and dedicated object that all endpoints call when an error comes in such as mail to admin and logging.
  5. The errors of known cases which are fully understood and can be handled thoughtfully called operational errors. We can operate them in the early process. Similarly, programmer error such as reading an undefined variable refers to unknown code failures which can be recovered by gracefully restarting the application.
  6. Using the serializers(status codes) to throw a 400/500/200 codes as per the error. It helps at the time of debugging.
  7. It's a good practice to add a custom exception handler for catching all general exceptions and returning a specific error message, we can also use a try except.

# HTML and CSS(Django Template)

## Style Rules

* + When the template detects a variable, like {{ variable }} which should consist of any alphanumeric characters and the underscore, but it should never start with an underscore. Never use spaces or punctuation marks in variable names.
  + The use of dot (.) should be done to access attributes of a variable. Although, it has a special meaning when appearing in variable sections.
  + The variable attributes beginning with an underscore are generally considered private variables.
  + It’s a good practice to modify variables for display by using filters like this: {{ name|lower }}. This displays the value of the variable after being filtered, which converts text to lowercase and enhances the use of variable and reduces additional methods.
  + Importantly, always use tags in your Django HTML template (not the HTML <tags>). They are more complex compared to the variables and are in {% tag %} format. Tags are very much useful to create text in the output or control the flow by performing loops or if-else logic statements in your HTML file. They also help in loading some external information into the template to be used by later variables or extending one file into another.
  + Django contains plenty of built-in template tags where some tags require beginning and closing tags as well. Use of tags in your template makes it easier for the programmer to execute some methods in the HTML file itself.

## Formatting Rules

* + Use valid CSS style rules and classes whenever possible. Using proper CSS code will help to find out the code which doesn't have any effect and we will be able to remove them which helps in simplifying the code and reducing the final payload size.
  + Writing style for related or common components in the same location makes it easier for developers to understand and work on the code. All the CSS files are stored at the same place and all the templates are stored altogether in another folder, which helps in bifurcation and accessing of files.
  + Using meaningful generic ID, class names and file names helps a lot in understanding the functioning of that particular class without in the detail. Instead of using any quickfix names for classes we should use meaningful names which represent the use of that particular component. Generic names can be used to define elements that have no different meaning and will be used as helpers.
  + For example, “btn” class for styling any button and then a perfectly named class as “btn-green” for giving it a specific color.

## Convention

* + It is a best practice to have template inheritance for reducing redundancy and unnecessary codes. Always use the [{% extends %}](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-extends) tag first in a template or else the template inheritance won’t work.
  + More use of the [{% block %}](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-block) tags in your base templates are better. But, it is not necessary to define all parent blocks in the child templates itself. So you can have reasonable defaults in blocks, then only define the ones you need for later use. It’s better to have more hooks than fewer hooks.
  + Never duplicate content in a number of templates, it is preferable to add that content to a {% block %} in a parent template to neglect repetition.
  + The {{ block.super }} variable can be used to get the content of the block from the parent template which is useful instead of completely overriding it.
  + By using the same template name as you are inheriting from, [{% extends %}](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-extends) and combined with {{ block.super }}, this can be a powerful way to make small customizations.
  + In larger templates, variables created outside of a [{% block %}](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-block) can’t be used inside the block, this technique helps you see which {% block %} tags are being closed.
  + Never define multiple [block](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-block) tags with the same name in the same template. template’s parent wouldn’t know which one of the blocks’ content to use.
  + For extra readability, we can give a *name* to the {% endblock %} tag as well.

## Commenting

* + Using valid comments in code is also important as that makes the code more readable. And as more than 1 person works on the same code the easier it is to find which style comes from where the faster we can collaborate.
  + The comment syntax: {# #} should be used to comment-out part of a line in a template. It helps in containing any part of template code, invalid or not.
  + It's a good practice to use this syntax for single-line comments as no newlines are allowed between the {# and #} delimiters. And use /\* \*/ to comment out a multi line portion of the template.

## Control Structures

* + The best practice, most powerful but most complex part of Django’s template engine, is integrating template inheritance in your code as much as possible.
  + This template, which we have called the base.html, defines an HTML skeleton document and its empty blocks are filled by the other child templates.
  + All the [block](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-block) tag does is to tell the template engine that a child template may override those portions of the template.
  + In our base file, one child component can fill the block. We have added the theme of the project, navigation, and sidebar in the base template. The [extends](https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#std:templatetag-extends) tag is the key which is used to extend another template in it. When the template system evaluates, first it locates the parent – in our case, “base.html”.
  + This helps in building a base “skeleton” template that defines blocks that child templates can override. It helps in controlling the flow of the template and reduces the code and enhances the reusability as well as understanding.