Coding Standards Documentation

Coding Standards for
Domain Pulse

Ctrl Alt Defeat

Contents

1 File Structure

Our system (Domain Pulse) has multiple different services and components that are often times independant from one another, as a result of our service oriented architecture. Furthermore, our system makes use of a Angular frontend and Django backend. As a result of these factors, the file structure of our system is designed to with modularity and simplicity as focuses, and we have achieved such. These qualities have been achieved with there being three main folder paths of note within our system: 'backend', 'frontend' and 'documentation' (of which will not be covered as it is not 'Coding'). Each folder is self-explanitory in its purpose and all code relating to said folder is contained within it, with little to no overlap.

1.1 Frontend

- The frontend folder consists of a 'src' folder that contains the developed resources.
- The 'assets' folder conveniently stores all assets (icons, backgrounds, etc.) required for UI development.
- The 'app' folder houses all the developed Angular UI pages, neatly organized in their respective component folders (e.g., login-page, register-page, etc.).

1.2 Backend

- Within the backend folder, each separate service of the system is stored in relevant folders, following a clearly defined and separated software-oriented architecture.
- This approach ensures modularity and flexibility within our system by clearly defining individual services.
- Each service folder contains one folder dedicated to storing the necessary Django settings, while other folders contain the developed code required for the proper execution of the respective service.

2 Naming Conventions

2.1 Frontend

When it comes to naming conventions for frontend development, follow these guidelines:

1. **Use meaningful and descriptive names**: Choose names that accurately describe the purpose or functionality of the component, variable, function, or class. Avoid using vague or ambiguous names.

2. **Follow camelCase for angular code**: Use camelCase notation for naming variables, functions, and object properties. Start with a lowercase letter and capitalize the first letter of each subsequent concatenated word. For example: `myVariable`, `getUserData()`, `myObject.property`.

3. **Use kebab-case for HTML and CSS**: Use kebab-case notation for naming HTML elements and CSS classes. Start each word with a lowercase letter and separate words with a hyphen. For example: `<my-component></my-component>`, `.my-class`.

4. **Use PascalCase for class and component names**: Class names and component names should follow PascalCase notation. Start each word with an uppercase letter. For example: `MyClass`, `MyComponent`.

5. **Use lowercase for file names**: File names should be in lowercase and use hyphens to separate words. For example: `my-file.js`, `my-component.html`, `styles.css`.

6. **Use self-explanatory names for CSS classes**: Choose CSS class names that are descriptive and convey their purpose. Avoid using generic names or abbreviations. Use hyphens to separate words in class names. For example: `.main-container`, `.btn-primary`, `.error-message`.

7. **Prefixes and suffixes**: Consider using consistent prefixes or suffixes for variables or functions to indicate their purpose or type. For example, prefixing boolean variables with ”is” or ”has” (`isActive`, `hasError`) or suffixing functions with verbs (`calculateTotal`, `validateForm`).

8. **Avoid excessive abbreviation**: While it's important to keep names concise, avoid excessive abbreviation that may make the code less readable. Aim for a balance between brevity and clarity.

2.2 Backend

- Snake case naming conventions (e.g. test_function) are followed within the backend to ensure readable and understandable code.

- Endpoints follow a well-defined structure, where the endpoint function has a descriptive name and the URL matches the endpoint name, and follow a format of 'service/endpoint' such as 'domains/create_domain'.

- Auxiliary files are created to store the logic of backend functions, organized in folders such as ’util’ or ’preprocess’ to indicate their purpose.
- Functions within auxiliary files include those with the same name as the endpoint (indicating they are called in the endpoint) and helper functions like `remove_whitespace`, promoting an orderly structure within main functions.
- Clear distinction between main and auxiliary functions enhances code readability and flexibility, making it easier to track down and fix bugs.
- Variable names are chosen to clearly describe their purpose within the codebase.

3 Formatting

3.1 Frontend

- Use tabs for indentation.
- Keep lines of code within a reasonable length, so it doesn't go off the screen.
- Align related code vertically so it looks organized.
- Put braces and parentheses where they look nice, either on the same line or the next line.
- Leave comments to explain what the code does, but not too many.
- Try to be consistent with how the code looks, so it doesn't confuse anyone.
- Use vscode auto-formatting to ensure consistency.

3.2 Backend

- The backend code of our system is developed using Django, which utilizes Python as its programming language.
- Python's inherent structure enforces a strict and ordered format due to its reliance on indentation.
- We follow the *Black* formatter guidelines to ensure code readability and maintain consistency within our development process.

4 Commit Structure

All commits made to the GitHub Repository have a specific commit message structure to be followed. The structure is as follows: [Emoji](frontend/backend/docs)'Commit Message'.

- The emoji, accessed from the 'Gitmoji' VS Code extension, represents the purpose of the commit at a glane, e.g. a bug fix being represented by an emoji of a bug.
- (frontend/backend/docs) is used to indicate which part of the system the commit is related to.
- The commit message is a descriptive message of the what work was done within the commit.

5 Pull Request Restrictions

Any branch being merged via pull request into the development branch (dev) needs to have the coverage of changes to the codebase to match or better the coverage of the development branch. Furthermore, the coverage of the newly commited code (ie: patch) must match or exceed the coverage percentage of the project. This ensures that the coverage of the codebase is never decreasing, and that sufficient testing is being done on the codebase. Furthermore any time the development branch is merged into the master branch, the coverage of the development branch must match or be higher than that of the master branch, ensuring an increasing coverage and sufficient testing.