TerraByte

Coding Standards

A project for EPI-USE® by GeekGurusUnion

Demo 4

30 September 2024

This document outlines the coding standards for the AI Crop Prediction project. These standards aim to improve code readability, maintainability, and consistency across the codebase. They cover aspects of coding style, linting, formatting, and general coding guidelines for both frontend and backend development.

Contents

Our Technology Stack:

- **Frontend:** Nuxt.js, Vue.js
- **Backend:** Python
- **Linting:** ESLint
- **Formatting:** Prettier

1. General Coding Style

Indentation:

- Use 4 tabs for indentation.
- There must be a space after giving a comma between two function arguments.
- Each nested block should be properly indented and spaced.
- Insert a space after commas, colons, and semicolons.
- All braces should start from a new line and the code following the end of braces also starts from a new line.

Naming Conventions:

- Use descriptive and meaningful names that reflect the purpose of the variable, function, or component.
- Use camelCase for variable and function names, starting with small letters (e.g., userName, calculateTotal).

- Local variables should be named using camel case lettering starting with small letters (e.g. localData) whereas Global variables names should start with a capital letter (e.g. GlobalData).
- It is better to avoid the use of digits in variable names.
- The name of the function must describe the reason for using the function clearly and briefly.

2. Frontend (Vue.js & Nuxt.js)

Linting:

For our linting purposes, we have chosen ESLint due to its robust features and seamless integration with various development tools. Below are the key reasons and setup instructions:

Why ESLint?

- **Static Analysis**: ESLint statically analyzes your code to quickly identify potential problems and enforce coding standards.
- **Editor Integration**: It is built into most text editors, providing instant feedback as you code.
- **CI/CD Integration**: ESLint can be easily integrated into your continuous integration pipeline to ensure code quality across the development lifecycle.

We are using the @nuxt/eslint module in our package.json as it seamlessly integrates with our current setup.

- **Module Documentation**: For detailed setup instructions, refer to the nuxt/ESLint Module documentation
- **ESLint Rules**: You can find a list of available ESLint rules for Vue.js in the User Guide | eslint-plugin-vue.

Formatting:

To maintain consistent code formatting, we use Prettier:

- **Why Prettier?**
    - **Consistency**: Prettier enforces a consistent style across the codebase, reducing the amount of time spent on code reviews and formatting debates.
    - **Automation**: It automatically formats your code, allowing developers to focus on writing code rather than formatting it.
- **Configuration**:
    - Configure Prettier according to the project's preferences. Some common preferences we decided to include:

        - **Single Quotes**: Prefer single quotes over double quotes.
        - **Semicolons**: Determine whether to use semicolons at the end of statements.
        - **Tab Width:** Set how many tabs to indent by.
        - **Print Width:** Set the line length
- **Integration**:
    - **IDE Integration**: Ensure Prettier is integrated into your IDE to automatically format code on save.
    - **Pre-Commit Hooks**: Use Husky pre-commit hooks to run Prettier before committing code to ensure all committed code is properly formatted.
    - **CI/CD Pipeline**: Integrate Prettier into the CI/CD pipeline to enforce consistent formatting across all code that is merged or deployed.

By following these guidelines, we ensure that our codebase remains clean, readable, and maintainable.

3. Backend (Python)

Python is a dynamically typed language, which allows for flexibility but can introduce challenges when building robust and maintainable backend systems. Type-related errors may only surface at runtime, leading to potential bugs and reduced reliability. **Pydantic** offers a solution by introducing **data validation** and **type checking** at runtime, enhancing the robustness of Python applications.

Class Definition:

- Use Pydantic models to define the shape and structure of your data. This enforces clear type definitions, making it easier to catch errors early.
- **Example:**

```
from pydantic import BaseModel

class User(BaseModel):
      id: int
      name: str
      email: str
```

Type Safety:

- Even though Python is dynamically typed, using Pydantic helps enforce type constraints, improving code clarity and safety.

- Encourage team members to specify types in function signatures and utilize Pydantic to enforce these types.
- **Example:**

```python
def create_user(user: User) -> None:
      # Business logic here
      pass
```

4. Coding Guidelines

Coding guidelines provide general suggestions regarding the coding style that should be followed to enhance readability, writability, and reliability of the code. Here are some key guidelines:

- **Avoid Complex Coding Styles**: Code should be easily understandable. Complex code makes maintenance and debugging difficult and expensive.
- **Use Descriptive Identifiers**: Each variable should be given a descriptive and meaningful name indicating its purpose. Avoid using an identifier for multiple purposes, as this can lead to confusion and difficulties during future enhancements.
- **Document Your Code**: Ensure the code is well-documented. Proper comments increase the ability to understand the code. Use comments to explain the "why" behind your code, not just the "what."
- **Keep Functions Small**: Functions should be small and focused on a single task. Lengthy functions are difficult to understand and maintain. Break down lengthy functions into smaller, more manageable ones.
- **File Length and Reusability**: Keep files at a consistent and maintainable length. Aim to minimize code repetition by creating reusable components. This promotes DRY (Don't Repeat Yourself) principles and makes the codebase more maintainable.
- **Consistent Formatting**: Adhere to the project's formatting standards to ensure a consistent look and feel across the codebase. Use tools like Prettier to automate formatting.

Reusable Components

- **Component Design**: Design components to be reusable. This means they should be modular, self-contained, and easily integrated into different parts of the application.
- **Avoid Code Duplication**: Identify common functionalities and abstract them into reusable components. This reduces code duplication and enhances maintainability.
- **Parameterization**: Make components flexible by parameterizing them where appropriate. This allows the same component to be used in different contexts with different data or settings.

- **Documentation**: Document the purpose and usage of reusable components clearly. This helps other developers understand how to use them effectively.

By following these guidelines, we can maintain a high standard of code quality, improve collaboration, and ensure the longevity and scalability of our project.

5. Enforcement

To ensure adherence to these coding standards, we have implemented the following enforcement mechanisms:

- **IDE Integration**:
    - **Formatting on Save**: Integrate Prettier into your IDE to automatically format your code on save, ensuring consistent code style across the project.
- **Pre-Commit Hooks**:
    - **Husky**: We have configured Husky pre-commit hooks to run linters and formatters before committing code. This ensures that only code adhering to our standards is committed to the repository.
- **Continuous Integration**:
    - **CI/CD Pipeline**: Our CI/CD pipeline is set up to run linters and formatting tools as part of the build process. This acts as a final check to ensure that all code meets our standards before it is merged or deployed.
- **Regular Code Reviews**:
    - Conduct regular code reviews to ensure ongoing adherence to these standards. Code reviews help catch issues that automated tools might miss and promote knowledge sharing and best practices among team members.

By integrating these tools and practices into our development workflow, we can maintain high code quality and consistency across the project.

6. Conclusion

Adhering to these coding standards will promote code quality, maintainability, and a collaborative development environment. Remember, this document is a living document, so feel free to suggest improvements as the project evolves.