# Coding Standards

Team : CacheME

Project : Secure File Sharing Platform

Version 4

# Table of Contents

# 1. Overview

This document defines the coding standards, Git workflow (GitFlow), and language specific conventions to be followed by all developers in this project. The goal is to maintain readability, consistency, and collaborative efficiency.

# 2. Repository Structure

### *Mono-Repo Structure*

This project uses a Mono-Repo approach, where both the backend (sfsp-api),  and frontend(sfsp-ui and sfsp-admin) are contained in a single repository.


### *Git flow*

We follow the **GitFlow** branching models.

### *Branch Types*

| Branch | Purpose | Naming Convention |
|---|---|---|
| main | Production-ready code | main |
| develop | Integration of features for next release | dev |
| feature/* | New features | feature/ui/login |
| bugfix/* | Non-critical bug fixes | bugfix/null-reference |
| hotfix/* | Critical production fixes | hotfix/fix-crash |

### *Rules*

- Never commit directly to main or develop.
- Always create Pull Requests (PRs) to merge into develop.
- PRs must be reviewed by at least one team member.
- PRs into dev must be reviewed by at least 2 members.
- PRs into main must be reviewed by the remaining 4 or 3 members.
- Use clear, concise commit messages.

### *Management*

- We use GitHub Issues and Projects for tracking tasks and bugs.
- We use code reviews for PRs.

# 3. Testing

- Every feature and bugfix must include appropriate unit and/or integration tests.
- Use mock data instead of live services for integration tests.

## *3.1 Types of Tests*

### Unit Testing

- **Using** : Jest , Cypress , Pytest, Testify
- UI specific : Each test has to have the same name as the component being tested.
    - Example : a component named sidebar.js will have a corresponding test called sidebar.cy.js

### Integration Testing

- **Using** : Cypress, Jest , Pytest, Testify

### End-to-End Testing

- **Using** : Cypress, Jest, Pytest, Testify

## *3.2 Code Coverage*

- 70% or higher coverage for critical system components

# 4. General Guidelines

---

- Follow DRY (Don't Repeat Yourself).
- Use meaningful variable and function names.
- Avoid magic numbers; define constants.
- Use environment variables for config, never hardcode secrets.
- Write docstrings/comments where the logic is non-trivial.
- Use camelCase for lengthy variables.

# 5. Language Specific Guidelines

---

## *Javascript*

- Use ESLint with Airbnb style guide.
- Use PascalCase for classes.
- Prefer const and let over var.
- Use arrow functions unless a method requires a context bound this.

**Example:**

Axios Request

```
import axios from 'axios';

// Good
async function fetchData() {
    try {
        const response = await axios.get('https://api.example.com/data');
        console.log(response.data);
    } catch (error) {
        console.error('Error fetching data:', error);
    }
}
```

## *Python*

- Follow Flask8.
- Use snake_case for functions and variables, PascalCase for classes.
- Use type hints where applicable.
- Use black for autoformatting.

**Example:**

Python Good Practices

```python
from typing import List

class DataProcessor:
    def __init__(self, data_source: str):
        self.data_source = data_source

    def load_data(self) -> List[str]:
        """Load data from the data source."""
        with open(self.data_source, 'r') as file:
            return [line.strip() for line in file]

def process_items(items: List[str]) -> None:
    for item in items:
        print(f"Processing: {item}")

if __name__ == "__main__":
    processor = DataProcessor("data.txt")
    items = processor.load_data()
    process_items(items)
```

## *Go*

---

- Use gofmt for formatting.
- Use golangci-lint for linting.
- Keep functions small and focused.
- Error handling must be explicit.

**Example:**

```go
// Good
func FetchUser(id string) (*User, error) {
    res, err := http.Get(fmt.Sprintf("/api/users/%s", id))
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()

    var user User
    if err := json.NewDecoder(res.Body).Decode(&user); err != nil {
        return nil, err
    }
    return &user, nil
}
```

# 6. Error Handling

- Raise errors as early as possible.
- Handle potential failures gracefully.
- Make use of meaningful error messages.

# 7. Linting

To maintain a consistent and high-quality codebase, all code must pass the configured linting and formatting checks.

**Guidelines:**

- All code must be linted and formatted using the appropriate tool.
- Developers should run the relevant linting/formatting commands locally.

# 8. Secrets Management

- Never commit .env files or API keys.
- Use .gitignore to exclude local config files.
- Use environment variables or secret managers (e.g. Vault, AWS Secrets Manager).

# 9. Documentation

- Update the README.md with every major change.
- Document API endpoints using markdown files for each api service where applicable.

# 10. Code Review Checklist

---

Before merging any PR:

- ☐ Follows GitFlow
- ☐ Lint passes
- ☐ No secrets committed
- ☐ Follows language specific standards
- ☐ Code is readable and well documented

# 11. Tooling

---

| **Tool** | **Purpose** |
|---|---|
| ESLint | JavaScript linting |
| Flask8 | Python formatting |
| GolangCI-Lint | Go linting |
| Prettier | Code formatting |
| Jest/PyTest/Cypress/Testify | Testing framework |
| Markdown | API documentation |