GITGOOD

Help gitGood, gitBetter

CODING STANDARDS

DEMO 4

WEATHER TO WEAR

EPI-USE®

KYLE LIEBENBERG
DIYA BUDHIA
ALISHA PERUMAL

IBRAHIM SAID
TAYLOR SERGEL

INTRODUCTION

We, the team of Weather to Wear, have decided on a preliminary standard for how we will approach coding this project. This document outlines the coding standards and conventions for the Weather To Wear project to ensure uniformity, clarity, flexibility, reliability, and efficiency in our codebase.

NAMING CONVENTIONS

1. We will be using meaningful names for variables, functions, folders and files.
2. Use appropriate whitespace to make code more readable.

SPECIFIC CONVENTIONS

1. Methods and variables:
    a. For methods and variables, we will be using camel case.
    b. This is where the name starts with a lowercase letter and ,if there are multiple words in the method or variable name, then the later words begin with capital letters.
2. Folders
    a. For folders in the system, we have decided to use kebab case.
    b. This is where words in the name of the folder are separated using hyphens. For example, app-backend, app-frontend, etc.
3. Classes:
    a. For classes, we will use pascal case.
    b. This is where words are differentiated from each other using the capital of it. For example, AppPage, HomePage, etc.
4. Constants:
    a. For constants, we will use snake case.
    b. This is where the name starts with a lowercase letter and ,if there are multiple words in the method or variable name, then the later words begin with capital letters.

CODE FORMATTING

1. Indentation and spacing
   a. Uses 2 spaces for indentation
   b. Add blank lines to separate logical code and functions.
   c. Use consistent spacing around operators
2. Braces and Brackets

```
// Correct - Opening brace on same line
if (condition) {
  doSomething();
} else {
  doSomethingElse();
}

// Functions
function fetchWeatherData() {
  // implementation
}
```

3. Semicolons
   a. Always use semicolons to terminate statements.

COMMENTING STANDARDS

1. **Single-line comments:** Use // for inline or single-line comments.

```
// example of inline comment
```

2. **Block comments:** Use /* */ for multi-line or block comments.

```
/*
* example of block comments
*/
```

FILE STRUCTURE

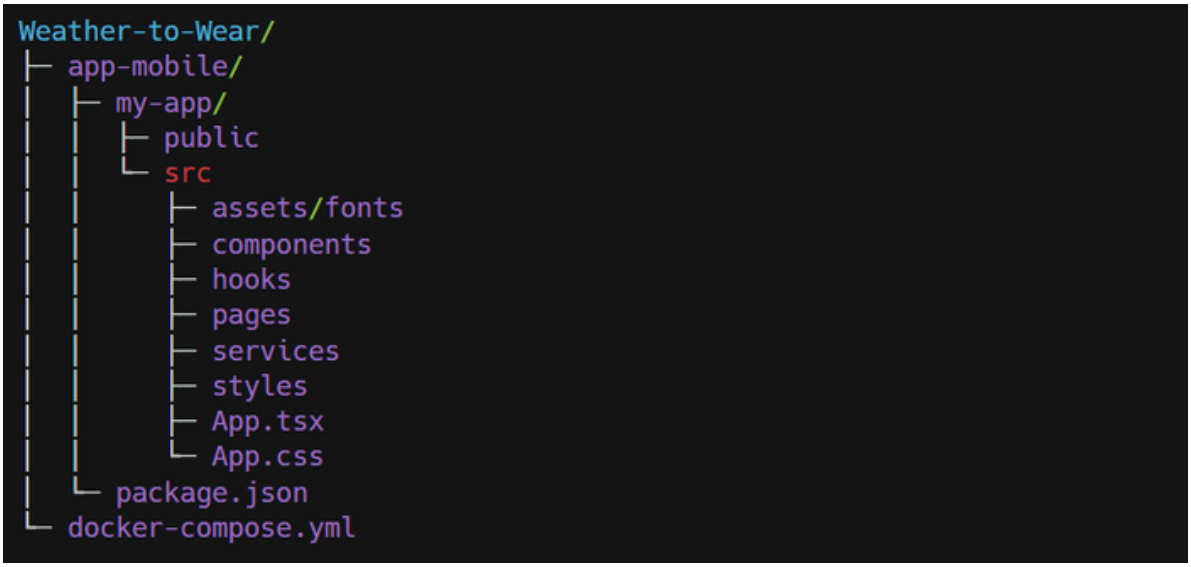
1. The Weather to Wear team has decided to have a file for each subsystem, such as the backend, frontend, and documentation.
2. Below is an outline of the main file structure:

```
Weather-to-Wear/
├─ .github/workflows
├─ app-backend
├─ app-mobile
├─ docs
├─ infra
├─ .gitignore
├─ README.md
├─ docker-compose.yml
└─ package.json
```

3. Below outlines the main file structure for the backend. The backend contains the server-side logic, API routes and database interactions.

```
Weather-to-Wear/
├─ app-backend/
│  ├─ uploads/
│  ├─ prisma/
│  │  └─ schema.prisma
│  ├─ src/
│  │  ├─ middleware/
│  │  ├─ modules/
│  │  │  ├─ auth/
│  │  │  ├─ weather/
│  │  │  └─ closet/
│  │  └─ app.ts
│  ├─ .env
│  └─ package.json
├─ docker-compose.yml
└─ uploads/
```

4. Below outlines the main file structure for the frontend. This includes client-side code, UI components and API integrations.



FILE NAMING

1. Use descriptive names that indicate the files purpose.
2. Group related files in appropriate directories.
3. Keep file names concise and meaningful.

TESTING AND DEBUGGUNG

1. Unit Testing
    a. Using: Jest
    b. Test individual components and functions in isolation to ensure that they behave as expected.
    c. Each test has to have the same name as the expected component being tested.
2. Integration Testing
    a. Using: Jest
    b. Test the interactions between different components work correctly to verify that they work in cohesion with each other.
3. End-to-end Testing
    a. Using: Cypress
    b. Test the entire application from the users perspective to ensure that all components work together from the frontend to the backend.
4. Acceptance Testing
    a. Using: Third party
    b. Users use the application and report back to ensure the application functionalities work as should be expected.

TESTING REQUIREMENTS

1. Minimum 80% code coverage measured by Jest.
2. All new features must include corresponding tests.
3. Tests must have descriptive names matching the components being tested.
4. Critical paths require both unit and integration tests.

CODE COVERAGE

1. Code coverage is 80% or higher
2. Measured using Jest

GITHUB STRUCTURE AND BRANCHING STRATEGY

OVERVIEW

The Weather to Wear project follows a monorepo structure, consolidating all components into a single GitHub repository for easier collaboration, integration, and deployment. The repository contains two main applications:

- **app-backend/** - Node.js backend with PostgreSQL, Dockerized and tested using Jest.
- **app-mobile/** - Frontend/mobile interface (under development).
- **.github/** - GitHub Actions workflows for CI/CD.
- **docs/** - Project documentation and planning assets.
- **infra/** - Infrastructure and deployment configurations.

We use a standardized Git branching model to support team collaboration and feature isolation:

- **main** - Stable production-ready branch.
- **dev -** Integration branch for completed features and testing.
- **feature/<name>** - Individual feature branches created from dev.
- **hotfix/<name>** - Emergency fixes branched from main.

BRANCHING STRATEGY AND NAMING

1. Each subsystem has its own branch for feature development.
2. Branch features should describe the feature in development.
3. Naming of branches should be meaningful.

COMMIT MESSAGES

1. Commit messages should describe the commit being made.
2. Explain meaningful changes to the current code.
3. Commit messages should be done after approximately 100 lines of code.

CODE REVIEW PROCESS

1. All code changes require peer review before merging.
2. Reviews must check for adherence to these standards.
3. Address all review comments before approval.
4. Use descriptive pull request titles and descriptions.

QUALITY TOOLS

1. ESLint - JavaScript Linting
2. Jest - Code coverage reporting

SECURITY GUIDELINES

1. Never commit sensitive data (API keys, passwords, tokens).
2. Use environment variables for configuration.
3. Validate and sanitize all user inputs.
4. Regularly update dependencies.

ENVIRONMENT VARIABLES

Our environment variables make use of the primary and secondary weather APIs, our database url, and the port we will operate on. The environment variables have the following structure:

```
const config = {
  FREE_WEATHER_API_KEY: process.env.WEATHER_API_KEY,
  OPENWEATHER_API_KEY: process.env.WEATHER_API_KEY,
  DATABASE_URL: process.env.DATABASE_URL,
  PORT: process.env.PORT || 5000
};
```

PERFORMANCE GUIDELINES

1. Avoid premature optimization.
2. Use appropriate data structures and algorithms.
3. Implement proper error handling and logging.
4. Consider caching for frequently accessed data.
5. Monitor and profile application performance.

DOCUMENTATION GUIDELINES

1. API endpoints must be documented.
2. Complex algorithms require explanation comments.
3. Setup and deployment instructions in README.
4. Database schema documentation should be concise and easy to follow.
5. Architecture decision records for major considerations and major changes to the architecture.

README STANDARDS

Each modules README should include:

1. Purpose and functionality
2. Installation instructions
3. Usage examples
4. API documentation
5. Contributing guidelines

TEAM RESPONSIBILITIES

1. All team members are responsible for following these standards.
2. Regular team reviews of standards and updates as needed.
3. Mentoring and knowledge sharing for consistent implementation.