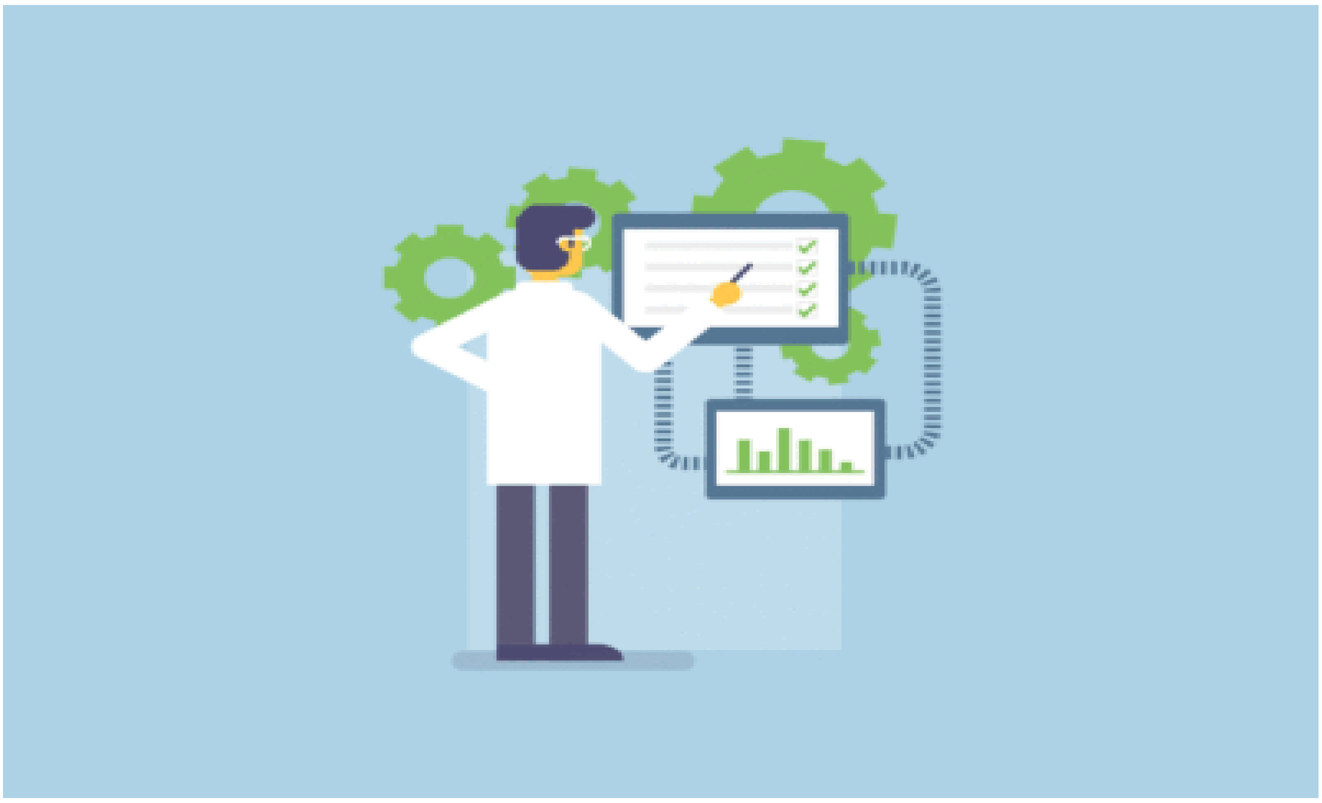

# Coding Standards for The Republic Project

Maintaining high coding standards is crucial for the success of The Republic project. This document provides comprehensive guidelines and best practices to ensure a clean, readable, and maintainable codebase.

## Table of Contents

1. General Guidelines
2. Naming Conventions
3. Code Structure
4. Error Handling
5. Testing Standards
6. Version Control Practices
7. Linter Rules

PROF

## General Guidelines

### Indentation and Formatting

- Use 4 spaces for indentation to enhance code structure and readability.
- Consistently apply indentation throughout the codebase.

Example:

```
function exampleFunction() {
    if (condition) {
        // This line is indented with 4 spaces
        doSomething();
    }
}
```

### Single Statement per Line

- Each statement should be on a separate line for clarity and readability.
- Avoid combining multiple statements on a single line, even if they are short.

Example:

```
// Good
let x = 5;
let y = 10;
let z = x + y;

// Avoid
let x = 5; let y = 10; let z = x + y;
```

## Naming Conventions

### Variables:

- Use camel case for variable names.
  - Start with a lowercase letter and capitalize subsequent words.
- Use snake case for constants.
  - Use all uppercase letters with underscores separating words.

PROF

Examples:

```
// Variables
let firstName: string = "John";
let lastName: string = "Doe";
let fullName: string = `${firstName} ${lastName}`;

// Constants
const MAX_RETRY_ATTEMPTS: number = 3;
const API_BASE_URL: string = "https://api.example.com";
```

### Functions:

- Use camel case for function names.

    - Start with a lowercase letter and capitalize subsequent words.
- Functions should have descriptive names that clearly indicate their purpose.

Examples:

```
function getUserName(userId: number): string {
    // Function logic
    return "username";
}

function calculateTotalPrice(items: Item[], discountPercentage: number):
number {
    // Function logic
    return totalPrice;
}
```

### Files:

- Use snake case or Pascal case for file names.
    - Snake case: start with a lowercase letter and use underscores.
    - Pascal case: start with a capital letter and capitalize all words.

Examples:

```
// Pascal case file names
import { UserProfile } from "./UserProfile";
import { AuthenticationService } from "./AuthenticationService";

// Snake case file names
import { database_connection } from "./database_connection";
import { error_handlers } from "./error_handlers";
```

PROF

## Code Structure

### Comments

- Add comments for function descriptions and complex code explanations.
- Use `//` for single-line comments and `/* */` for multi-line comments.
- Write clear and concise comments that add value to the code.

Example:

```
/**
 * Calculates the total price of items after applying a discount.
 * @param items An array of Item objects to calculate the total price
for.
```

```
 * @param discountPercentage The percentage of discount to apply (0-
100).
 * @returns The total price after applying the discount.
 */
function calculateTotalPrice(items: Item[], discountPercentage: number):
number {
    // Calculate the sum of all item prices
    const subtotal = items.reduce((total, item) => total + item.price,
0);

    // Apply the discount
    const discountAmount = subtotal * (discountPercentage / 100);
    const totalPrice = subtotal - discountAmount;

    return totalPrice;
}
```

## Error Handling

- Raise errors early in the code to prevent cascading issues.
- Restore state and resources after handling errors to maintain system integrity.
- Provide meaningful error messages for better understanding and debugging.

Example:

```
function transferFunds(fromAccount: Account, toAccount: Account, amount:
number): void {
    if (amount <= 0) {
        throw new Error("Transfer amount must be greater than zero");
    }

    if (fromAccount.balance < amount) {
        throw new Error("Insufficient funds for transfer");
    }

    try {
        fromAccount.withdraw(amount);
        toAccount.deposit(amount);
    } catch (error) {
        // Restore the original state if an error occurs during transfer
        fromAccount.deposit(amount);
        console.error("Fund transfer failed:", error.message);
        throw new Error("Fund transfer failed. Please try again
later.");
    }
}
```

PROF

## Testing Standards

## Types of Tests

**Unit Testing:**

- Test individual components, functions, and utilities in isolation.
- Use Jest for unit testing.
- Each test file should have the same name as the component being tested, with a `.test.ts` or `.test.tsx` extension.

Example:

```
// File: UserProfile.test.ts
import { UserProfile } from "./UserProfile";

describe("UserProfile", () => {
    it("should return the full name", () => {
        const profile = new UserProfile("John", "Doe");
        expect(profile.getFullName()).toBe("John Doe");
    });

    it("should update the email address", () => {
        const profile = new UserProfile("John", "Doe");
        profile.setEmail("john.doe@example.com");
        expect(profile.getEmail()).toBe("john.doe@example.com");
    });
});
```

**Integration Testing:**

- Test interactions between different components or modules.
- Use Cypress for integration testing.
- Focus on testing the flow of data and control between components.

PROF

Example:

```
// File: authentication.spec.ts
describe("Authentication Flow", () => {
    it("should log in and redirect to dashboard", () => {
        cy.visit("/login");
        cy.get("#username").type("testuser");
        cy.get("#password").type("password123");
        cy.get("#login-button").click();
        cy.url().should("include", "/dashboard");
        cy.get("#welcome-message").should("contain", "Welcome, Test
User");
    });
});
```

**End-to-End Testing:**

- Test the entire application from the user's perspective.
- Use Cypress for end-to-end testing.
- Simulate real user scenarios and test critical user journeys.

Example:

```
// File: checkout.spec.ts
describe("Checkout Process", () => {
    it("should complete a purchase", () => {
        cy.login("testuser", "password123");
        cy.visit("/products");
        cy.get(".product-item").first().click();
        cy.get("#add-to-cart").click();
        cy.get("#cart").click();
        cy.get("#checkout-button").click();
        cy.get("#shipping-address").type("123 Test St, Test City,
12345");
        cy.get("#payment-method").select("Credit Card");
        cy.get("#card-number").type("4111111111111111");
        cy.get("#card-expiry").type("12/25");
        cy.get("#card-cvv").type("123");
        cy.get("#place-order").click();
        cy.get("#order-confirmation").should("be.visible");
        cy.get("#order-number").should("not.be.empty");
    });
});
```

### Code Coverage

- Aim for 80% or higher code coverage for critical components.
- Measure code coverage using Jest.
- Regularly review and improve test coverage.

PROF

Example of running Jest with coverage:

```
jest --coverage
```

## Version Control Practices

### Git Flow

- Use the Git Flow branching strategy for parallel development.
- Branches: Main, Development, Feature, Documentation.

Example of creating a feature branch:

```
git checkout develop
git checkout -b feature/new-user-registration
```

### Git Branch Naming Conventions

- Descriptive, concise, and reflective of the work in the branch.
- Lowercase and hyphen-separated.
- Alphanumeric characters with no continuous hyphens.

Examples:

```
main
develop
feature/user-authentication
feature/payment-integration
hotfix/login-bug
release/v1.2.0
```

### Review Process

- Base features on the development branch.
- Implement automated checks for linting and unit tests.
- Conduct manual review by testers.

Example of creating a pull request:

```
git push origin feature/new-user-registration
# Then create a pull request on GitHub/GitLab from feature/new-user-
registration to develop
```

PROF

### Commits

- Make one commit per feature or each aspect of a feature.
- Write descriptive and concise commit messages.
- Use separate commits for code changes, tests, and documentation.

Example of good commit messages:

```
Add user registration form
Implement password strength validation
Update user registration documentation
Add unit tests for user registration process
```

### CI/CD

- Use ESLint for code quality and consistency.
- Implement custom rules aligned with coding standards.
- Set up automated checks and manual review for compliance.

## Linter Rules

### Extends

We use predefined sets of rules from various sources to maintain code quality:

- **eslint:recommended**: Basic rules for JavaScript.
- **plugin:@next/next/recommended**: Rules specific to Next.js.
- **plugin:@typescript-eslint/recommended**: Rules for TypeScript.

### Parser

We use a special tool called a parser to understand our TypeScript code:

- **@typescript-eslint/parser**: This parser helps ESLint understand TypeScript.

### Parser Options

These settings tell the parser how to interpret our code:

- **ecmaVersion**: 2020 (Supports modern JavaScript features).
- **sourceType**: module (Allows the use of import/export statements).
- **jsx**: true (Supports JSX, a syntax used in React).

### Settings

These settings help configure specific tools:

- **react version**: detect (Automatically detects the React version).

PROF

### Environment

These settings specify the environments our code is expected to run in:

- **jest/globals**: true (Supports Jest testing framework).
- **browser**: true (Code runs in a web browser).
- **es6**: true (Supports ES6 features).
- **node**: true (Code runs in Node.js).
- **jest**: true (Supports Jest testing framework).

### Plugins

Plugins add extra rules and functionalities:

- **react**: For React-specific linting rules.

- **@next/next**: For Next.js-specific linting rules.
- **@typescript-eslint**: For TypeScript-specific linting rules.
- **jest**: For Jest-specific linting rules.

### Rules

These are specific guidelines our code must follow:

- **semi**: Always use semicolons.
- **@typescript-eslint/no-explicit-any**: Avoid using the `any` type in TypeScript.
- **import/order**: No specific order for import statements.

By following these coding standards, we ensure consistency, readability, and maintainability throughout the development lifecycle of The Republic project.

---

Back to Table of Contents