Save n Bite - Coding Standards Document

Project: Save n Bite
Team: Secure Web & Mobile Guild (SWMG)
Version: 3.0
Date: August 15, 2025

Table of Contents

1. Overview
2. General Coding Principles
3. Repository Structure
4. Backend Standards (Django/Python)
5. Frontend Standards (React/JavaScript)
6. Database Standards
7. API Standards
8. Testing Standards
9. Version Control Standards
10. Environment Configuration
11. Code Quality Tools
12. Security Standards

Overview

This document establishes coding conventions and styles for the Save n Bite project to ensure uniformity, clarity, flexibility, reliability, and efficiency across our codebase. The project is a digital platform aimed at reducing food waste by connecting food providers with individuals and organizations in need.

Technology Stack

- **Backend:** Django + Django REST Framework, PostgreSQL, Redis, Scikit-learn, Pandas
- **Frontend:** React.js with Vite, Tailwind CSS
- **Testing:** Jest (Frontend), Django Test Framework (Backend)
- **Dependency Management:** Poetry (Python), npm/yarn (JavaScript)
- **Version Control:** Git with GitHub
- **Deployment:** AWS (planned)

General Coding Principles

Code Quality Standards

1. **Readability First:** Code should be self-documenting and easily understood
2. **DRY Principle:** Don't Repeat Yourself - extract common functionality
3. **SOLID Principles:** Follow object-oriented design principles
4. **Consistent Naming:** Use descriptive, consistent naming conventions

5. **Modular Design:** Keep components and modules focused and reusable
6. **Error Handling:** Implement comprehensive error handling and logging

Documentation Requirements

- All functions/methods must have docstrings
- Complex logic requires inline comments
- API endpoints must be documented
- Database schema changes require migration documentation
- README files for each major component

Repository Structure

```
save-n-bite/
├── save-n-bite-backend/       # Django backend application
│   ├── backend/               # Main Django project
│   │   ├── __init__.py
│   │   ├── settings.py        # Configuration settings
│   │   ├── urls.py           # Main URL routing
│   │   ├── wsgi.py           # WSGI configuration
│   │   └── asgi.py           # ASGI configuration
│   ├── authentication/        # User authentication app
│   ├── admin_system/          # Administration app
│   ├── food_listings/         # Food listing management
│   ├── interactions/          # User interactions (cart, orders)
│   ├── notifications/         # Notification system
│   ├── analytics/            # AI/ML analytics
│   ├── scheduling/           # Pickup/delivery scheduling
│   ├── reviews/              # Review and rating system
│   ├── static/               # Static files
│   ├── media/                # User-uploaded media
│   ├── logs/                 # System logs
│   ├── requirements.txt      # Environment requirements
│   ├── manage.py             # Django management script
│   ├── blop_storage.py       # BLOB srorage config
│   └── pyproject.toml        # Poetry configuration
├── save-n-bite-frontend/     # React frontend application
│   ├── dist/                 # containing assets
│   ├── public/               # Public assets
│   ├── src/                  # Source code
│   │   ├── components/       # React components
│   │   │   ├── auth/         # Authentication components
│   │   │   ├── common/       # Shared components
│   │   │   └── pages/        # Page components
│   │   ├── hooks/            # Custom React hooks
│   │   ├── services/         # API service layer
│   │   ├── utils/            # Utility functions
│   │   ├── styles/           # CSS/styling files
│   │   ├── __mocks__/        # Test mocks
│   │   └── setupTests.js     # Test configuration
```

```
│   ├── package.json          # Dependencies and scripts
│   ├── vite.config.js        # Vite configuration
│   ├── tailwind.config.js    # Tailwind CSS configuration
│   └── jest.config.js        # Jest testing configuration
├── documentation/            # Project documentation
└── README.md                # Project overview
```

File Organization Rules

- Each Django app should be focused on a single domain
- React components should be organized by feature/page
- Shared utilities go in dedicated directories
- Test files should mirror source structure
- Configuration files at project root

Backend Standards (Django/Python)

Python Code Style

- **PEP 8 Compliance:** Follow Python Enhancement Proposal 8
- **Line Length:** Maximum 88 characters (Black formatter standard)
- **Imports:** Use absolute imports, group by standard/third-party/local
- **Naming Conventions:**
  - Variables and functions: `snake_case`
  - Classes: `PascalCase`
  - Constants: `UPPER_SNAKE_CASE`
  - Private methods: `_leading_underscore`

Django Conventions

Models

```python
class FoodListing(models.Model):
    """Model representing a food listing from providers."""

    title = models.CharField(max_length=200, help_text="Food item title")
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    expiration_date = models.DateTimeField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = 'food_listings'
        ordering = ['-created_at']

    def __str__(self):
        return self.title
```

```
    def is_expired(self):
        """Check if the food item has expired."""
        return timezone.now() > self.expiration_date
```

Views

```
class FoodListingViewSet(viewsets.ModelViewSet):
    """ViewSet for managing food listings."""

    queryset = FoodListing.objects.all()
    serializer_class = FoodListingSerializer
    permission_classes = [IsAuthenticatedOrReadOnly]
    filter_backends = [DjangoFilterBackend, SearchFilter]
    search_fields = ['title', 'description']

    def get_queryset(self):
        """Filter queryset based on user permissions."""
        queryset = super().get_queryset()
        if not self.request.user.is_staff:
            queryset = queryset.filter(is_active=True)
        return queryset
```

Serializers

```
class FoodListingSerializer(serializers.ModelSerializer):
    """Serializer for food listing data."""

    is_expired = serializers.ReadOnlyField()
    provider_name = serializers.CharField(source='provider.name', read_only=True)

    class Meta:
        model = FoodListing
        fields = '__all__'

    def validate_expiration_date(self, value):
        """Ensure expiration date is in the future."""
        if value <= timezone.now():
            raise serializers.ValidationError("Expiration date must be in the
future.")
        return value
```

Error Handling

```
# Use custom exception classes
class FoodListingError(Exception):
```

```
    """Base exception for food listing operations."""
    pass

# Proper logging
import logging
logger = logging.getLogger(__name__)

def create_food_listing(data):
    try:
        listing = FoodListing.objects.create(**data)
        logger.info(f"Created food listing: {listing.id}")
        return listing
    except ValidationError as e:
        logger.error(f"Validation error creating food listing: {e}")
        raise FoodListingError("Invalid food listing data")
```

Frontend Standards (React/JavaScript)

JavaScript/TypeScript Style

- **ESLint Configuration:** Extend React App configuration
- **Prettier Integration:** Automatic code formatting
- **Naming Conventions:**
  - Components: `PascalCase` (e.g., `FoodCard.jsx`)
  - Files: `PascalCase` for components, `camelCase` for utilities
  - Variables/Functions: `camelCase`
  - Constants: `UPPER_SNAKE_CASE`

React Component Structure

```
import React, { useState, useEffect } from 'react';
import PropTypes from 'prop-types';

/**
 * FoodCard component displays individual food listing information
 * @param {Object} props - Component props
 * @param {Object} props.foodItem - Food listing data
 * @param {Function} props.onAddToCart - Callback for adding to cart
 */
const FoodCard = ({ foodItem, onAddToCart }) => {
  const [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    // Component lifecycle logic
  }, []);

  const handleAddToCart = async () => {
    setIsLoading(true);
    try {
```

```
      await onAddToCart(foodItem.id);
    } catch (error) {
      console.error('Failed to add to cart:', error);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="bg-white rounded-lg shadow-md p-4">
      <h3 className="text-lg font-semibold">{foodItem.title}</h3>
      <p className="text-gray-600">{foodItem.description}</p>
      <button
        onClick={handleAddToCart}
        disabled={isLoading}
        className="mt-2 bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600
disabled:opacity-50"
      >
        {isLoading ? 'Adding...' : 'Add to Cart'}
      </button>
    </div>
  );
};

FoodCard.propTypes = {
  foodItem: PropTypes.shape({
    id: PropTypes.number.isRequired,
    title: PropTypes.string.isRequired,
    description: PropTypes.string.isRequired,
  }).isRequired,
  onAddToCart: PropTypes.func.isRequired,
};

export default FoodCard;
```

Custom Hooks

```
// hooks/useFoodListings.js
import { useState, useEffect } from 'react';
import { foodListingService } from '../services/api';

export const useFoodListings = () => {
  const [listings, setListings] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchListings = async () => {
      try {
        const data = await foodListingService.getAll();
        setListings(data);
```

```
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchListings();
  }, []);

  return { listings, loading, error };
};
```

Styling Standards (Tailwind CSS)

- Use utility-first approach with Tailwind CSS
- Extract complex styles into component classes when needed
- Maintain consistent spacing scale (4, 8, 12, 16, 24, 32, etc.)
- Use semantic color names from design system

```
// Good: Utility classes
<div className="bg-white shadow-lg rounded-lg p-6 mb-4">

// Good: Custom component class when complex
<div className="food-card">

// Bad: Inline styles
<div style={{ backgroundColor: '#fff', padding: '24px' }}>
```

Database Standards

PostgreSQL Conventions

- **Table Names:** Lowercase with underscores (e.g., `food_listings`)
- **Column Names:** Lowercase with underscores (e.g., `created_at`)
- **Primary Keys:** Always use `id` as primary key
- **Foreign Keys:** Format as `{table}_id` (e.g., `user_id`)
- **Indexes:** Add indexes for frequently queried columns
- **Constraints:** Use database constraints for data integrity

Migration Standards

```
# migrations/0001_initial.py
from django.db import migrations, models

class Migration(migrations.Migration):
    """Initial migration for food listings app."""
```

```
    initial = True

    dependencies = [
        ('authentication', '0001_initial'),
    ]

    operations = [
        migrations.CreateModel(
            name='FoodListing',
            fields=[
                ('id', models.BigAutoField(primary_key=True)),
                ('title', models.CharField(max_length=200)),
                # ... other fields
            ],
            options={
                'db_table': 'food_listings',
                'ordering': ['-created_at'],
            },
        ),
    ]
```

API Standards

RESTful API Design

- Use HTTP methods appropriately (GET, POST, PUT, PATCH, DELETE)
- Follow REST naming conventions for endpoints
- Use plural nouns for resource names
- Implement proper HTTP status codes
- Include pagination for list endpoints

URL Patterns

```
# urls.py
urlpatterns = [
    path('api/food-listings/', FoodListingListCreateView.as_view()),
    path('api/food-listings/<int:pk>/', FoodListingDetailView.as_view()),
    path('api/users/<int:user_id>/orders/', UserOrderListView.as_view()),
]
```

Response Format Standards

```
{
  "success": true,
  "data": {
    "id": 1,
```

```
    "title": "Fresh Vegetables",
    "description": "Organic vegetables near expiry",
    "price": "5.99",
    "created_at": "2025-06-26T10:00:00Z"
  },
  "message": "Food listing retrieved successfully"
}

// Error Response
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid data provided",
    "details": {
      "price": ["This field is required"]
    }
  }
}
```

Testing Standards

Backend Testing (Django)

```python
# tests.py
from django.test import TestCase
from django.contrib.auth.models import User
from .models import FoodListing

class FoodListingTestCase(TestCase):
    """Test cases for FoodListing model."""

    def setUp(self):
        """Set up test data."""
        self.user = User.objects.create_user(
            username='testuser',
            email='test@example.com',
            password='testpass123'
        )

    def test_create_food_listing(self):
        """Test creating a food listing."""
        listing = FoodListing.objects.create(
            title='Test Food',
            description='Test description',
            price=10.99,
            provider=self.user
        )
        self.assertEqual(listing.title, 'Test Food')
        self.assertEqual(str(listing), 'Test Food')
```

Frontend Testing (Jest/React Testing Library)

```jsx
// FoodCard.test.jsx
import React from 'react';
import { render, fireEvent, screen } from '@testing-library/react';
import FoodCard from './FoodCard';

describe('FoodCard Component', () => {
  const mockFoodItem = {
    id: 1,
    title: 'Test Food',
    description: 'Test description',
    price: '10.99'
  };

  const mockOnAddToCart = jest.fn();

  beforeEach(() => {
    mockOnAddToCart.mockClear();
  });

  test('renders food item information', () => {
    render(<FoodCard foodItem={mockFoodItem} onAddToCart={mockOnAddToCart} />);

    expect(screen.getByText('Test Food')).toBeInTheDocument();
    expect(screen.getByText('Test description')).toBeInTheDocument();
  });

  test('calls onAddToCart when button is clicked', () => {
    render(<FoodCard foodItem={mockFoodItem} onAddToCart={mockOnAddToCart} />);

    fireEvent.click(screen.getByText('Add to Cart'));
    expect(mockOnAddToCart).toHaveBeenCalledWith(1);
  });
});
```

Test Coverage Requirements

- **Minimum Coverage:** 70% for all components and functions
- **Critical Paths:** 90% coverage for authentication and critical flows
- **Unit Tests:** All utility functions and business logic
- **Integration Tests:** API endpoints and database operations
- **End-to-End Tests:** Critical user journeys

Version Control Standards

Git Workflow

- **Main Branch:** `main` - Production-ready code
- **Development Branch:** `develop` - Integration branch
- **Feature Branches:** `feature/feature-name`
- **Bug Fix Branches:** `bugfix/issue-description`
- **Release Branches:** `release/version-number`

Commit Message Format

```
" <meaningful description> "
```

Types:

- `feat`: New feature
- `fix`: Bug fix
- `docs`: Documentation changes
- `style`: Code style changes
- `refactor`: Code refactoring
- `test`: Adding or updating tests
- `chore`: Maintenance tasks

Examples:

```
feat(auth): add JWT token refresh functionality

Implement automatic token refresh to improve user experience
by maintaining session without requiring re-login.

Closes #123
```

Pull Request Guidelines

- Use descriptive titles and descriptions
- Reference related issues
- Include screenshots for UI changes
- Ensure all tests pass
- Require code review approval
- Update documentation when necessary

Environment Configuration

Environment Variables

```
Note the following is not a true reflection on what is actually contained in .env
# .env (not committed to git)
SECRET_KEY=your_secret_key_here
```

```
DEBUG=True
DATABASE_URL=postgresql://user:password@localhost:5432/savenbite_db
REDIS_URL=redis://localhost:6379/0
AWS_ACCESS_KEY_ID=your_aws_key
AWS_SECRET_ACCESS_KEY=your_aws_secret
```

Configuration Management

```python
# settings.py
import os
from decouple import config

SECRET_KEY = config('SECRET_KEY')
DEBUG = config('DEBUG', default=False, cast=bool)

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME'),
        'USER': config('DB_USER'),
        'PASSWORD': config('DB_PASSWORD'),
        'HOST': config('DB_HOST', default='localhost'),
        'PORT': config('DB_PORT', default='5432'),
    }
}
```

Code Quality Tools

Backend Tools

- **Black:** Code formatting
- **isort:** Import sorting
- **flake8:** Linting
- **mypy:** Type checking

Frontend Tools

- **ESLint:** JavaScript linting
- **Prettier:** Code formatting
- **Jest:** Testing framework
- **React Testing Library:** Component testing

Configuration Files

.eslintrc.js

```
module.exports = {
  extends: [
    'react-app',
    'react-app/jest'
  ],
  rules: {
    'no-console': 'warn',
    'prefer-const': 'error',
    'no-unused-vars': 'error'
  }
};
```

pyproject.toml (Poetry)

```
# This is not a full version of the toml

[tool.poetry]
name = "save-n-bite-backend"
version = "0.1.0"
description = "Backend for Save n Bite food waste reduction platform"

[tool.poetry.dependencies]
python = "^3.10"
django = "^5.2"
djangorestframework = "^3.14"
psycopg2-binary = "^2.9"
redis = "^4.5"
scikit-learn = "^1.2"
pandas = "^2.0"

[tool.black]
line-length = 88
target-version = ['py310']

[tool.isort]
profile = "black"
```

Security Standards

Authentication & Authorization

- Use JWT tokens for API authentication
- Implement role-based access control (RBAC)
- Secure password storage with Django's built-in hashing
- Implement rate limiting on API endpoints
- Use HTTPS in production

Data Protection

- Encrypt sensitive data at rest
- Implement input validation and sanitization
- Use parameterized queries to prevent SQL injection
- Validate file uploads and restrict file types
- Implement CSRF protection

Privacy Compliance

- Store minimal personal information
- Implement data retention policies
- Provide user data export/deletion capabilities
- Log access to sensitive data
- Regular security audits

Environment Security

```
# settings.py security settings
SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
X_FRAME_OPTIONS = 'DENY'
SECURE_HSTS_SECONDS = 31536000  # 1 year
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True

# CORS settings
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",  # Development
    "https://savenbite.com",  # Production
]
```

Conclusion

This coding standards document serves as the foundation for maintaining high-quality, consistent code across the Save n Bite project. All team members are expected to follow these guidelines and update this document as the project evolves.

Regular code reviews and automated quality checks ensure adherence to these standards, promoting a maintainable and scalable codebase that supports the project's mission of reducing food waste and connecting communities.

Document Maintainers:

- Sabrina-Gabriel Freeman (Team Lead)
- Marco Geral (Backend Lead)

Last Updated: August 15, 2025