CRISP Coding Standards & Guidelines

Overview

This document outlines the coding standards and conventions used in the CRISP (Cybersecurity Resilience through Information Sharing Platform) project. CRISP is a Django-based threat intelligence platform with a React frontend, implementing STIX/TAXII standards for threat data sharing.

These standards ensure uniform style, clarity, flexibility, reliability, and efficiency across the CRISP platform codebase. They promote maintainable, secure, and scalable code while facilitating team collaboration and code review processes.

Project Structure

Backend (Django)

- **Framework**: Django 4.2.10 with Django REST Framework
- **Database**: PostgreSQL with psycopg2-binary
- **Architecture**: Service-oriented with Repository pattern
- **Authentication**: JWT-based with django-rest-framework-simplejwt
- **Task Queue**: Celery with Redis
- **WebSocket Support**: Django Channels with Redis channel layer

Frontend (React)

- **Framework**: React with Vite build tool
- **Routing**: React Router DOM
- **Styling**: CSS modules and Tailwind CSS
- **State Management**: React hooks and context
- **API Integration**: Fetch API with JWT authentication

Table of Contents

Repository Structure

The CRISP platform follows a modular architecture with clear separation of concerns:

```
Capstone-Unified/
  README.md                        # Project overview and setup instructions
  requirements.txt                 # Python dependencies
  manage.py                       # Django management commands
  docker-compose.yml              # Container orchestration
  Dockerfile                      # Application containerization
  .env.example                    # Environment variables template
  .gitignore                      # Version control exclusions
  CODING_STANDARDS.md             # This document
  TESTING.md                      # Testing guidelines
  DEPLOYMENT_README.md            # Deployment instructions
  SECURITY_TESTING_GUIDE.md       # Security testing procedures

  settings/                       # Django configuration
     __init__.py
     settings.py                 # Main settings file
     urls.py                     # URL routing configuration
     wsgi.py                     # WSGI application
     asgi.py                     # ASGI application (WebSocket support)
     apps.py                     # Custom app configurations

  core/                           # Core application logic
     __init__.py
     admin.py                    # Django admin configuration
     models/                     # Data models
        __init__.py
        models.py               # Main threat intelligence models
        user_behavior_models.py # Behavior analytics models
     api/                        # REST API endpoints
        __init__.py
        auth_api.py             # Authentication endpoints
        threat_feed_views.py    # Threat feed management
        user_api.py             # User management
        reports_api.py          # Reporting endpoints
     services/                   # Business logic layer
        __init__.py
        auth_service.py         # Authentication logic
        threat_service.py       # Threat processing
        user_service.py         # User management
        email_service.py        # Email notifications
     serializers/                # API data serialization
```

```
        __init__.py
        auth_serializer.py
        threat_feed_serializer.py
    middleware/                   # Request/response processing
        __init__.py
        audit_middleware.py       # Request auditing
        trust_middleware.py       # Trust level enforcement
    management/                   # Custom Django commands
        __init__.py
        commands/                 # Management commands
            populate_database.py
            sync_mitre_data.py
            cleanup_feeds.py
    migrations/                   # Database schema changes
    tests/                        # Unit and integration tests
        __init__.py
        test_models.py
        test_services.py
        test_api.py
    patterns/                     # Design patterns implementation
        factory/                  # Factory pattern
        observer/                 # Observer pattern
        strategy/                 # Strategy pattern
        decorator/                # Decorator pattern
    repositories/                 # Data access layer
        __init__.py
        threat_feed_repository.py
        indicator_repository.py

core/user_management/             # User management module
    models/
        user_models.py            # Custom user models
        invitation_models.py     # User invitation system
    services/
        user_service.py
        organization_service.py
    views/
        auth_views.py
    tests/

core/trust_management/            # Trust relationship module
    models/
        trust_models.py
    services/
        trust_service.py
    tests/
```

```
core/alerts/                      # Alert management module
   alerts_views.py
   alerts_urls.py

soc/                              # Security Operations Center
   __init__.py
   api.py                         # SOC-specific API endpoints
   urls.py                        # SOC URL routing
   consumers.py                   # WebSocket consumers
   routing.py                     # WebSocket routing

frontend/                         # React frontend application
   crisp-react/
       package.json               # Node.js dependencies
       vite.config.js             # Build configuration
       index.html                 # Main HTML template
       src/
          main.jsx                # Application entry point
          App.jsx                 # Root component
          api.js                  # API service layer
          components/             # React components
             enhanced/            # Enhanced UI components
             soc/                 # SOC dashboard components
             threat/              # Threat management components
             user/                # User management components
             trust/               # Trust management components
          assets/                 # Static assets and styles
          test/                   # Frontend tests
              unit/
              integration/
              e2e/

scripts/                          # Utility scripts
   setup/                         # Setup and initialization
      setup_clean.sh
      reset_db.sh
   testing/                       # Testing scripts
      run-all-tests.sh
      locustfile.py               # Load testing
   security/                      # Security testing
       run-security-tests.sh

security-testing/                 # Security assessment tools
   configs/                       # Security tool configurations
   reports/                       # Security scan results
```

```
    wordlists/                  # Custom wordlists

docs/                           # Project documentation
    API_DOCUMENTATION.md
    ARCHITECTURE.md
    DEPLOYMENT_GUIDE.md
```

Naming Conventions

Python (Backend)

- **Files**: snake_case (e.g., `auth_service.py`, `threat_feed_views.py`)
- **Classes**: PascalCase (e.g., `AuthenticationService`, `ThreatFeedRepository`)
- **Functions/Methods**: snake_case (e.g., `authenticate_user`, `get_by_stix_id`)
- **Variables**: snake_case (e.g., `user_session`, `auth_result`)
- **Constants**: SCREAMING_SNAKE_CASE (e.g., `MAX_TRUST_LEVEL`)
- **Private methods**: Prefix with underscore (e.g., `_get_client_info`)
- **Test files**: Prefix with `test_` (e.g., `test_auth_service.py`)

JavaScript/React (Frontend)

- **Files**: PascalCase for components (e.g., `UserManagement.jsx`)
- **Components**: PascalCase (e.g., `LoadingSpinner`, `ConfirmationModal`)
- **Functions**: camelCase (e.g., `getUsersList`, `createUser`)
- **Variables**: camelCase (e.g., `currentUser`, `showModal`)
- **Constants**: SCREAMING_SNAKE_CASE (e.g., `API_BASE_URL`)
- **CSS Files**: kebab-case (e.g., `threat-dashboard.css`, `user-profile.css`)

Database

- **Tables**: Plural, snake_case (e.g., `custom_users`, `threat_feeds`)
- **Columns**: snake_case (e.g., `created_at`, `trust_level`)
- **Foreign Keys**: `{model}_id` format (e.g., `organization_id`)
- **Indexes**: `idx_{table}_{columns}` format (e.g., `idx_threat_indicators_type_created`)

Configuration Files

- **Docker**: Descriptive names (e.g., `docker-compose.production.yml`)
- **Environment**: `.env`, `.env.example`, `.env.production`
- **Scripts**: Descriptive kebab-case (e.g., `run-security-tests.sh`)

Directory Organization Principles

1. **Separation of Concerns**: Each directory has a single, well-defined responsibility
2. **Modularity**: Related functionality is grouped together

3. **Scalability**: Structure supports growth without major reorganization
4. **Clarity**: Directory names clearly indicate their purpose
5. **Consistency**: Similar patterns across all modules

General Principles

Code Quality Standards

- **Readability First**: Code is read more than it's written
- **Consistency**: Follow established patterns within the codebase
- **Simplicity**: Prefer simple solutions over complex ones
- **DRY Principle**: Don't Repeat Yourself
- **SOLID Principles**: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion
- **Security by Design**: Consider security implications in every code change

Version Control

- Use meaningful commit messages following conventional commit format
- Keep commits atomic and focused on a single change
- Use feature branches for all development
- Squash commits before merging to main branch
- Never commit sensitive information (keys, passwords, etc.)

Python/Django Standards

Code Style Guidelines

Python Style

- **Line Length**: Maximum 120 characters
- **Imports**: Group in order: standard library, third-party, local imports
- **Docstrings**: Use triple quotes for all functions and classes
- **Type Hints**: Use where applicable, especially in service classes
- **Error Handling**: Use specific exception types with meaningful messages

Example Python Code Style:

```python
"""
Authentication Service - JWT-based authentication with trust integration
Handles user authentication, session management, and security features
"""
from typing import Dict, Optional, Tuple
from django.contrib.auth import authenticate
from rest_framework_simplejwt.tokens import RefreshToken
from core.services.trust_service import TrustService

class AuthenticationService:
    """Enhanced authentication service with trust-aware access control"""

    def __init__(self):
        self.trust_service = TrustService()

    def authenticate_user(self, username: str, password: str,
                         request=None, remember_device: bool = False) -> Dict:
        """Authenticate user with comprehensive security checks"""
        auth_result = {
            'success': False,
            'user': None,
            'message': ''
        }

        try:
            user = authenticate(username=username, password=password)
            if user and user.is_active:
                auth_result.update({
                    'success': True,
                    'user': user,
                    'message': 'Authentication successful'
                })
            return auth_result
        except Exception as e:
            logger.error(f"Authentication failed: {e}")
            auth_result['message'] = 'Authentication failed'
            return auth_result
```

Type Hints

```
from typing import List, Dict, Optional, Union
from django.db.models import QuerySet

# Always use type hints for function parameters and return values
def get_user_alerts(user_id: str, severity: Optional[str] = None) -> QuerySet[Alert]:
    """Retrieve alerts for a specific user with optional severity filter."""
    alerts = Alert.objects.filter(user_id=user_id)
    if severity:
        alerts = alerts.filter(severity=severity)
    return alerts

# Use Union for multiple possible types
def process_indicator_data(data: Union[str, Dict[str, str]]) -> bool:
    """Process indicator data from string or dictionary format."""
    pass
```

Django-Specific Standards

Models

```python
from django.db import models
from django.core.validators import validate_email
import uuid

class ThreatIndicator(models.Model):
    """Threat intelligence indicator model."""

    # Use UUIDs for primary keys in security-sensitive models
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

    # Use descriptive field names
    indicator_type = models.CharField(
        max_length=50,
        choices=IndicatorType.choices,
        help_text="Type of threat indicator"
    )

    # Add validation where appropriate
    confidence_score = models.IntegerField(
        validators=[validators.MinValueValidator(0), validators.MaxValueValidator(100)],
        help_text="Confidence score from 0-100"
    )

    # Always include audit fields
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    created_by = models.ForeignKey('user_management.CustomUser', on_delete=models.PROTECT)

    class Meta:
        db_table = 'threat_indicators'
        indexes = [
            models.Index(fields=['indicator_type', 'created_at']),
            models.Index(fields=['confidence_score']),
        ]
        ordering = ['-created_at']

    def __str__(self) -> str:
        return f"{self.indicator_type}: {self.value}"

    def clean(self):
        """Custom validation logic."""
        super().clean()
        if self.indicator_type == 'email' and self.value:
            validate_email(self.value)
```

Views

```
from django.shortcuts import get_object_or_404
from rest_framework import status
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

@api_view(['GET', 'POST'])
@permission_classes([IsAuthenticated])
def threat_indicators_view(request):
    """Handle threat indicator operations."""

    if request.method == 'GET':
        # Use pagination for list views
        indicators = ThreatIndicator.objects.filter(
            organization=request.user.organization
        )
        serializer = ThreatIndicatorSerializer(indicators, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        # Always validate input data
        serializer = ThreatIndicatorSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save(created_by=request.user)
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Services Layer

```
from django.db import transaction
from core.exceptions import ThreatProcessingError

class ThreatIntelligenceService:
    """Service for threat intelligence operations."""

    @staticmethod
    @transaction.atomic
    def process_threat_feed(feed_data: Dict[str, Any]) -> ProcessingResult:
        """Process threat intelligence feed data."""
        try:
            # Validate input
            if not feed_data.get('indicators'):
                raise ThreatProcessingError("No indicators found in feed")

            # Process indicators
            processed_count = 0
            for indicator_data in feed_data['indicators']:
                if ThreatIntelligenceService._is_valid_indicator(indicator_data):
                    ThreatIndicator.objects.create(**indicator_data)
                    processed_count += 1

            return ProcessingResult(
                success=True,
                processed_count=processed_count,
                message=f"Successfully processed {processed_count} indicators"
            )

        except Exception as e:
            # Log error details
            logger.error(f"Failed to process threat feed: {str(e)}")
            raise ThreatProcessingError(f"Processing failed: {str(e)}")

    @staticmethod
    def _is_valid_indicator(indicator_data: Dict[str, Any]) -> bool:
        """Validate individual indicator data."""
        required_fields = ['type', 'value', 'confidence']
        return all(field in indicator_data for field in required_fields)
```

Error Handling

```
import logging
from django.core.exceptions import ValidationError
from rest_framework.views import exception_handler

logger = logging.getLogger(__name__)

class CRISPException(Exception):
    """Base exception for CRISP platform."""
    pass

class ThreatProcessingError(CRISPException):
    """Exception for threat processing errors."""
    pass

def custom_exception_handler(exc, context):
    """Custom exception handler for API responses."""
    response = exception_handler(exc, context)
    if response is not None:
        # Log the error
        logger.error(f"API Error: {exc}", extra={'context': context})

        # Customize error response
        custom_response_data = {
            'error': {
                'status_code': response.status_code,
                'message': 'An error occurred',
                'details': response.data
            }
        }
        response.data = custom_response_data

    return response
```

Architecture Patterns

Design Patterns Used

1. **Repository Pattern**: Data access abstraction (`IndicatorRepository`, `ThreatFeedRepository`)
2. **Service Pattern**: Business logic encapsulation (`AuthService`, `TrustService`)
3. **Factory Pattern**: Object creation (`StixFactory`, `TaxiiServiceFactory`)
4. **Observer Pattern**: Event handling (`FeedObservers`, `AlertSystemObserver`)
5. **Decorator Pattern**: Feature enhancement (`StixDecorator`)
6. **Strategy Pattern**: Algorithm selection (`AnonymizationStrategies`)

Service Layer Guidelines

- Services handle business logic and orchestration
- Repositories handle data access only
- API views are thin and delegate to services
- Use dependency injection where possible
- Maintain single responsibility per service
- Implement proper error handling and logging

JavaScript/React Standards

Code Style

```
// Use ES6+ features
// Use camelCase for variables and functions
// Use PascalCase for components
// Use UPPER_SNAKE_CASE for constants
// Good
const API_ENDPOINTS = {
  THREAT_INDICATORS: '/api/threat-indicators/',
  USER_ALERTS: '/api/alerts/'
};

const ThreatIndicatorComponent = ({ indicators, onUpdate }) => {
  const [loading, setLoading] = useState(false);

  const handleIndicatorUpdate = async (indicatorId, data) => {
    setLoading(true);
    try {
      await updateThreatIndicator(indicatorId, data);
      onUpdate();
    } catch (error) {
      console.error('Failed to update indicator:', error);
    } finally {
      setLoading(false);
    }
  };

  return (
    <div className="threat-indicator-container">
      {/* Component content */}
    </div>
  );
};

// Bad
const threatIndicatorComponent = function(props) {
  var Loading = false;

  function HandleIndicatorUpdate(indicatorId, data) {
    Loading = true;
    updateThreatIndicator(indicatorId, data).then(() => {
      props.onUpdate();
      Loading = false;
    }).catch(err => {
      console.log(err);
      Loading = false;
    });
  }

  return React.createElement('div', null, 'content');
}
```

JavaScript/React Style

- **Line Length**: Maximum 120 characters
- **Semicolons**: Always use semicolons
- **Quotes**: Use single quotes for strings, double quotes for JSX attributes
- **Arrow Functions**: Prefer arrow functions for functional components
- **Destructuring**: Use object destructuring where appropriate
- **Hooks**: Follow React hooks rules (use at top level)

Example React Code Style:

```
import React, { useState, useEffect } from 'react';
import { getUsersList, createUser } from '../../api.js';
import LoadingSpinner from './LoadingSpinner.jsx';

const UserManagement = ({ active = true, initialSection = null }) => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchUsers = async () => {
      try {
        const response = await getUsersList();
        setUsers(response.data);
      } catch (err) {
        setError('Failed to fetch users');
      } finally {
        setLoading(false);
      }
    };

    if (active) {
      fetchUsers();
    }
  }, [active]);

  if (loading) {
    return <LoadingSpinner message="Loading users..." />;
  }

  if (error) {
    return <div className="error-message">{error}</div>;
  }

  return (
    <div className="user-management">
      <h2>User Management</h2>
      {/* Component content */}
    </div>
  );
};

export default UserManagement;
```

API Integration

```
// Create a centralized API service
class APIService {
  constructor(baseURL = '/api') {
    this.baseURL = baseURL;
    this.token = localStorage.getItem('access_token');
  }

  async request(endpoint, options = {}) {
    const url = `${this.baseURL}${endpoint}`;
    const config = {
      headers: {
        'Content-Type': 'application/json',
        ...(this.token && { Authorization: `Bearer ${this.token}` }),
        ...options.headers
      },
      ...options
    };

    try {
      const response = await fetch(url, config);

      if (response.status === 401) {
        // Handle token expiration
        this.handleTokenExpiration();
        throw new Error('Authentication required');
      }

      if (!response.ok) {
        throw new Error(`HTTP ${response.status}: ${response.statusText}`);
      }

      return await response.json();
    } catch (error) {
      console.error(`API request failed: ${endpoint}`, error);
      throw error;
    }
  }

  async get(endpoint) {
    return this.request(endpoint);
  }

  async post(endpoint, data) {
    return this.request(endpoint, {
      method: 'POST',
      body: JSON.stringify(data)
    });
  }

  handleTokenExpiration() {
    localStorage.removeItem('access_token');
    window.location.href = '/login';
  }
}

export const apiService = new APIService();
```

Database Standards

Schema Design

```
-- Use meaningful table and column names
-- Include audit columns on all tables
-- Use appropriate data types
-- Add indexes for performance

CREATE TABLE threat_indicators (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    indicator_type VARCHAR(50) NOT NULL,
    indicator_value TEXT NOT NULL,
    confidence_score INTEGER CHECK (confidence_score >= 0 AND confidence_score <= 100),
    source_feed_id UUID REFERENCES threat_feeds(id),
    organization_id UUID REFERENCES organizations(id) NOT NULL,

    -- Audit fields
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_by UUID REFERENCES users(id) NOT NULL,

    -- Constraints
    CONSTRAINT unique_indicator_per_org UNIQUE (indicator_value, organization_id)
```

```
);

-- Indexes for performance
CREATE INDEX idx_threat_indicators_type_created ON threat_indicators(indicator_type, createc
CREATE INDEX idx_threat_indicators_org_confidence ON threat_indicators(organization_id, conf
CREATE INDEX idx_threat_indicators_source ON threat_indicators(source_feed_id);
```

Migration Standards

```
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):
    """Add threat indicator confidence scoring."""

    dependencies = [
        ('core', '0015_add_threat_feeds'),
    ]

    operations = [
        # Always provide defaults for new non-nullable fields
        migrations.AddField(
            model_name='threatindicator',
            name='confidence_score',
            field=models.IntegerField(default=50, help_text='Confidence score 0-100'),
        ),

        # Add indexes in separate operations for better performance
        migrations.RunSQL(
            "CREATE INDEX CONCURRENTLY idx_threat_indicators_confidence ON core_threatindica
            reverse_sql="DROP INDEX idx_threat_indicators_confidence;"
        ),

        # Use data migrations for complex data changes
        migrations.RunPython(
            code=update_existing_confidence_scores,
            reverse_code=migrations.RunPython.noop
        ),
    ]

def update_existing_confidence_scores(apps, schema_editor):
    """Update confidence scores for existing indicators."""
    ThreatIndicator = apps.get_model('core', 'ThreatIndicator')

    # Batch update for performance
    indicators = ThreatIndicator.objects.filter(confidence_score__isnull=True)
    for indicator in indicators.iterator(chunk_size=1000):
        indicator.confidence_score = calculate_confidence(indicator)
        indicator.save(update_fields=['confidence_score'])
```

Security Standards

Authentication & Authorization

```
from django.contrib.auth.decorators import login_required
from core.decorators import require_organization_access

@login_required
@require_organization_access
@api_view(['GET'])
def sensitive_data_view(request, organization_id):
    """Access sensitive organizational data."""

    # Verify user has specific permissions
    if not request.user.has_perm('core.view_sensitive_data'):
        return Response(
            {'error': 'Insufficient permissions'},
            status=status.HTTP_403_FORBIDDEN
        )

    # Additional authorization checks
    if not request.user.can_access_organization(organization_id):
        return Response(
            {'error': 'Organization access denied'},
            status=status.HTTP_403_FORBIDDEN
        )

    # Proceed with data retrieval
    data = get_sensitive_data(organization_id)
    return Response(data)
```

Input Validation

```
from django.core.exceptions import ValidationError
from django.utils.html import escape
import re

class ThreatIndicatorValidator:
    """Validator for threat indicator data."""

    @staticmethod
    def validate_ip_address(ip_address: str) -> bool:
        """Validate IP address format."""
        ip_pattern = r'^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$'
        if not re.match(ip_pattern, ip_address):
            raise ValidationError('Invalid IP address format')

        # Additional validation for private/reserved ranges
        octets = [int(x) for x in ip_address.split('.')]
        if octets[0] in [10, 127] or (octets[0] == 172 and 16 <= octets[1] <= 31):
            raise ValidationError('Private IP addresses not allowed')

        return True

    @staticmethod
    def sanitize_input(user_input: str) -> str:
        """Sanitize user input to prevent XSS."""
        if not isinstance(user_input, str):
            raise ValidationError('Input must be a string')

        # Remove potentially dangerous characters
        sanitized = escape(user_input.strip())

        # Additional sanitization rules
        if len(sanitized) > 1000:
            raise ValidationError('Input too long')

        return sanitized
```

Secure Configuration

```
# settings/security.py
import os

# Security Headers
SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
X_FRAME_OPTIONS = 'DENY'
SECURE_REFERRER_POLICY = 'strict-origin-when-cross-origin'

# HTTPS Settings (Production)
if not DEBUG:
    SECURE_SSL_REDIRECT = True
    SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
    SECURE_HSTS_SECONDS = 31536000  # 1 year
    SECURE_HSTS_INCLUDE_SUBDOMAINS = True
    SECURE_HSTS_PRELOAD = True

# Session Security
```

```
SESSION_COOKIE_SECURE = not DEBUG
SESSION_COOKIE_HTTPONLY = True
SESSION_COOKIE_SAMESITE = 'Strict'
SESSION_COOKIE_AGE = 3600  # 1 hour

# CSRF Protection
CSRF_COOKIE_SECURE = not DEBUG
CSRF_COOKIE_HTTPONLY = True
CSRF_COOKIE_SAMESITE = 'Strict'

# Content Security Policy
CSP_DEFAULT_SRC = ("'self'",)
CSP_SCRIPT_SRC = ("'self'", "'unsafe-inline'")  # Minimize unsafe-inline usage
CSP_STYLE_SRC = ("'self'", "'unsafe-inline'")
CSP_IMG_SRC = ("'self'", "data:", "https:")
```

API Standards

REST API Guidelines

- Use standard HTTP methods (GET, POST, PUT, DELETE)
- Use appropriate HTTP status codes
- Implement pagination for list endpoints
- Use consistent URL patterns (`/api/v1/resource/`)
- Include API versioning in URLs
- Implement proper error handling and validation

Response Format

```
{
  "success": true,
  "data": {},
  "message": "Operation successful",
  "pagination": {
    "page": 1,
    "total_pages": 10,
    "total_items": 100
  }
}
```

Error Response Format

```
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input data",
    "details": {
      "field_name": ["This field is required"]
    }
  }
}
```

Testing Standards

Backend Testing

- **Framework**: Django's built-in testing framework
- **Coverage**: Aim for 80%+ test coverage
- **Test Types**: Unit tests, integration tests, end-to-end tests
- **File Naming**: `test_*.py` (e.g., `test_auth_service.py`)
- **Test Methods**: `test_*` prefix (e.g., `test_authenticate_user_success`)

Frontend Testing

- **Framework**: ESLint for linting, Vitest for unit testing
- **Component Testing**: Test user interactions and state changes
- **API Testing**: Mock API calls in component tests
- **File Naming**: `*.test.jsx` or `*.spec.jsx`

Unit Tests

```python
from django.test import TestCase
from django.contrib.auth import get_user_model
from unittest.mock import patch, Mock
from core.models import ThreatIndicator
from core.services import ThreatIntelligenceService

User = get_user_model()

class ThreatIntelligenceServiceTest(TestCase):
    """Test cases for ThreatIntelligenceService."""

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

    def test_process_valid_threat_feed(self):
        """Test processing valid threat feed data."""
        feed_data = {
            'indicators': [
                {
                    'type': 'ip',
                    'value': '192.168.1.100',
                    'confidence': 85,
                    'source': 'test_feed'
                }
            ]
        }

        result = ThreatIntelligenceService.process_threat_feed(feed_data)

        self.assertTrue(result.success)
        self.assertEqual(result.processed_count, 1)
        self.assertEqual(ThreatIndicator.objects.count(), 1)

    def test_process_invalid_threat_feed(self):
        """Test processing invalid threat feed data."""
        feed_data = {'indicators': []}

        with self.assertRaises(ThreatProcessingError):
            ThreatIntelligenceService.process_threat_feed(feed_data)

    @patch('core.services.logger')
    def test_process_threat_feed_logs_errors(self, mock_logger):
        """Test that errors are properly logged."""
        feed_data = None

        with self.assertRaises(ThreatProcessingError):
            ThreatIntelligenceService.process_threat_feed(feed_data)

        mock_logger.error.assert_called_once()
```

Integration Tests

```
from django.test import TransactionTestCase
from django.db import transaction
from rest_framework.test import APITestCase
from rest_framework import status

class ThreatIndicatorAPITest(APITestCase):
    """Integration tests for Threat Indicator API."""

    def setUp(self):
        """Set up test environment."""
        self.user = User.objects.create_user(
            username='apiuser',
            email='api@example.com',
            password='apipass123'
        )
        self.client.force_authenticate(user=self.user)

    def test_create_threat_indicator(self):
        """Test creating a threat indicator via API."""
        data = {
            'type': 'domain',
            'value': 'malicious.example.com',
            'confidence': 90,
            'description': 'Known malicious domain'
        }

        response = self.client.post('/api/threat-indicators/', data)

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(response.data['value'], 'malicious.example.com')

        # Verify database state
        indicator = ThreatIndicator.objects.get(id=response.data['id'])
        self.assertEqual(indicator.created_by, self.user)

    def test_unauthorized_access(self):
        """Test that unauthorized users cannot access API."""
        self.client.force_authenticate(user=None)

        response = self.client.get('/api/threat-indicators/')

        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

Frontend Tests

```
import React from 'react';
import { render, screen, fireEvent, waitFor } from '@testing-library/react';
import { rest } from 'msw';
import { setupServer } from 'msw/node';
import ThreatDashboard from '../components/ThreatDashboard';

// Mock API server
const server = setupServer(
  rest.get('/api/threats/', (req, res, ctx) => {
    return res(
      ctx.json({
        results: [
          { id: '1', type: 'ip', value: '192.168.1.1', confidence: 85 },
          { id: '2', type: 'domain', value: 'evil.com', confidence: 95 }
        ]
      })
    );
  })
);

beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());
afterAll(() => server.close());

describe('ThreatDashboard', () => {
  test('renders threat data correctly', async () => {
    render(<ThreatDashboard organizationId="org-123" />);

    // Wait for data to load
    await waitFor(() => {
      expect(screen.getByText('192.168.1.1')).toBeInTheDocument();
      expect(screen.getByText('evil.com')).toBeInTheDocument();
    });
  });

  test('handles API errors gracefully', async () => {
    // Override the API response to return an error
    server.use(
      rest.get('/api/threats/', (req, res, ctx) => {
        return res(ctx.status(500), ctx.json({ error: 'Server error' }));
      })
    );

    const mockOnError = jest.fn();
    render(
      <ThreatDashboard
        organizationId="org-123"
        onError={mockOnError}
      />
    );

    await waitFor(() => {
      expect(mockOnError).toHaveBeenCalledWith(
        expect.objectContaining({ message: expect.stringContaining('500') })
      );
    });
  });
  test('refresh button updates data', async () => {
    render(<ThreatDashboard organizationId="org-123" />);

    // Wait for initial load
    await waitFor(() => {
      expect(screen.getByText('192.168.1.1')).toBeInTheDocument();
    });

    // Click refresh button
    const refreshButton = screen.getByRole('button', { name: /refresh/i });
    fireEvent.click(refreshButton);

    // Verify API is called again
    await waitFor(() => {
      expect(screen.getByText('192.168.1.1')).toBeInTheDocument();
    });
  });
});
```

Documentation Standards

Code Documentation

```
def process_threat_indicators(
    indicators: List[Dict[str, Any]],
    organization_id: str,
    confidence_threshold: int = 50
) -> ProcessingResult:
    """
    Process a list of threat indicators for an organization.

    This function validates, normalizes, and stores threat indicators
    while applying organization-specific filtering rules.

    Args:
        indicators: List of indicator dictionaries containing 'type', 'value',
                   and 'confidence' keys
        organization_id: UUID string identifying the target organization
        confidence_threshold: Minimum confidence score to accept indicators (0-100)

    Returns:
        ProcessingResult: Object containing success status, processed count,
                         and any error messages

    Raises:
```

```python
        ValidationError: If indicators contain invalid data
        OrganizationNotFoundError: If organization_id is invalid
        ProcessingError: If processing fails due to system errors

    Example:
        >>> indicators = [
        ...     {'type': 'ip', 'value': '1.2.3.4', 'confidence': 85},
        ...     {'type': 'domain', 'value': 'evil.com', 'confidence': 95}
        ... ]
        >>> result = process_threat_indicators(indicators, 'org-123')
        >>> print(f"Processed {result.processed_count} indicators")
        Processed 2 indicators
    """
    pass
```

API Documentation

```python
from drf_yasg.utils import swagger_auto_schema
from drf_yasg import openapi

class ThreatIndicatorViewSet(viewsets.ModelViewSet):
    """
    ViewSet for managing threat indicators.

    Provides CRUD operations for threat indicators within an organization.
    All operations require authentication and appropriate permissions.
    """

    @swagger_auto_schema(
        operation_description="Create a new threat indicator",
        request_body=openapi.Schema(
            type=openapi.TYPE_OBJECT,
            required=['type', 'value', 'confidence'],
            properties={
                'type': openapi.Schema(
                    type=openapi.TYPE_STRING,
                    description='Type of indicator (ip, domain, hash, etc.)',
                    enum=['ip', 'domain', 'hash', 'url']
                ),
                'value': openapi.Schema(
                    type=openapi.TYPE_STRING,
                    description='The indicator value'
                ),
                'confidence': openapi.Schema(
                    type=openapi.TYPE_INTEGER,
                    description='Confidence score (0-100)',
                    minimum=0,
                    maximum=100
                ),
            }
        ),
        responses={
            201: openapi.Response('Indicator created successfully'),
            400: openapi.Response('Invalid input data'),
            401: openapi.Response('Authentication required'),
            403: openapi.Response('Insufficient permissions')
        }
    )
    def create(self, request):
        """Create a new threat indicator."""
        pass
```

Performance Guidelines

Database Optimization

```
# Use select_related for foreign key relationships
indicators = ThreatIndicator.objects.select_related(
    'organization', 'created_by'
).filter(active=True)

# Use prefetch_related for many-to-many relationships
organizations = Organization.objects.prefetch_related(
    'users', 'threat_feeds'
).all()

# Use only() to limit fields when you don't need full objects
indicators = ThreatIndicator.objects.only(
    'id', 'type', 'value', 'confidence'
).filter(confidence__gte=80)

# Use bulk operations for multiple creates/updates
ThreatIndicator.objects.bulk_create([
    ThreatIndicator(type='ip', value=ip, confidence=85)
    for ip in ip_list
], batch_size=1000)

# Use database functions for aggregation
from django.db.models import Count, Avg
stats = ThreatIndicator.objects.aggregate(
    total_count=Count('id'),
    avg_confidence=Avg('confidence')
)
```

Frontend Optimization

```
// Use React.memo for expensive components
const ThreatList = React.memo(({ threats, onSelect }) => {
  return (
    <div>
      {threats.map(threat => (
        <ThreatItem
          key={threat.id}
          threat={threat}
          onSelect={onSelect}
        />
      ))}
    </div>
  );
});

// Use useMemo for expensive calculations
const ThreatAnalytics = ({ threats }) => {
  const analytics = useMemo(() => {
    return {
      totalThreats: threats.length,
      highConfidence: threats.filter(t => t.confidence > 80).length,
      threatsByType: threats.reduce((acc, threat) => {
        acc[threat.type] = (acc[threat.type] || 0) + 1;
        return acc;
      }, {})
    };
  }, [threats]);

  return <AnalyticsDisplay data={analytics} />;
};

// Use useCallback for event handlers
const ThreatDashboard = () => {
  const [selectedThreat, setSelectedThreat] = useState(null);

  const handleThreatSelect = useCallback((threat) => {
    setSelectedThreat(threat);
    // Track selection for analytics
    analytics.track('threat_selected', { threatId: threat.id });
  }, []);

  return (
    <ThreatList
      threats={threats}
      onSelect={handleThreatSelect}
    />
  );
};
```

Environment and Configuration

Environment Variables

- Use `.env` files for configuration
- Never commit sensitive data to version control
- Use different configurations for development/production
- Document all required environment variables in `.env.example`

Dependencies

- Pin dependency versions in `requirements.txt` and `package.json`
- Regularly update dependencies for security patches
- Use virtual environments for Python development
- Run security audits on dependencies regularly

Tools and Linting

Backend Tools

- **Black**: Code formatting (if configured)
- **flake8**: Linting (if configured)
- **isort**: Import sorting (if configured)
- **Django Debug Toolbar**: Development debugging
- **pytest**: Alternative testing framework

Frontend Tools

- **ESLint**: Configured with React plugins
- **Prettier**: Code formatting (if configured)
- **Vite**: Build tool and development server
- **Vitest**: Testing framework

IDE Configuration

- Use consistent IDE settings across team
- Configure auto-formatting on save
- Enable linting in IDE
- Use consistent indentation (4 spaces for Python, 2 for JavaScript)

Commit Messages

Conventional Commit Format

Use conventional commit format for all commit messages:

```
feat: add user authentication with 2FA support
- Implement JWT-based authentication
- Add two-factor authentication using TOTP
- Update user model with security fields

Fixes #123
```

Commit 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

Guidelines

- Use present tense ("add feature" not "added feature")
- Use imperative mood ("move cursor to…" not "moves cursor to…")
- Include issue numbers when applicable
- Keep first line under 50 characters
- Provide detailed description in body if needed

Code Review Process

Review Guidelines

Code reviews should focus on ensuring that code follows established style guidelines, includes appropriate tests, addresses security considerations, considers performance implications, updates documentation, implements comprehensive error handling, protects sensitive data, optimizes database queries, and maintains readability.

Pull Request Template

```
## Description
Brief description of the changes and their purpose.

## Type of Change
Specify whether this is a bug fix, new feature, breaking change, documentation update, perfo
```

```
## Testing
Describe unit tests, integration tests, manual testing, and performance testing that has bee

## Security Considerations
Document input validation, authorization checks, sensitive data protection, and security hea

## Documentation
Note any code comments, API documentation updates, README changes, or CHANGELOG entries.

## Review Notes
Include any additional context for reviewers about style compliance, self-review completion,
```

Enforcement

These standards are enforced through: - **Automated linting**: Black, flake8, ESLint - **Pre-commit hooks**: Style checking, security scanning - **CI/CD pipeline**: Automated testing and quality checks - **Code review process**: Peer review requirement - **Security scanning**: Regular vulnerability assessments

Conclusion

These coding standards ensure consistency, maintainability, and security across the CRISP project. All team members should follow these guidelines and update them as the project evolves. Regular code reviews and automated tools help enforce these standards.

The standards cover: - **Uniform Style**: Consistent naming conventions and code formatting - **Clarity**: Clear documentation and readable code structure - **Flexibility**: Modular architecture supporting future enhancements - **Reliability**: Comprehensive testing and error handling - **Efficiency**: Performance optimization and best practices

For questions or clarifications about these standards, please refer to the project documentation or consult with team members. These guidelines are living documents and should be updated as the project evolves and new best practices emerge.

Quick Reference

- **Python**: 120 chars, snake_case, type hints, comprehensive docstrings
- **JavaScript**: 120 chars, camelCase, ES6+, React hooks patterns
- **Database**: snake_case, proper indexing, migration safety
- **API**: RESTful design, consistent response format, proper status codes
- **Security**: Input validation, authentication, authorization, secure headers
- **Testing**: 80%+ coverage, unit/integration tests, meaningful test names
- **Git**: Conventional commits, feature branches, peer review required