```
"""
Crypto Service - Cryptographic operations

Provides:
- AES-256-CBC encryption via Fernet
- SHA-256 hashing for pseudonyms
- Secure key generation and management
- Evidence hash calculation
- Encryption/decryption for sensitive data
"""

import hashlib
import logging
import secrets
from datetime import datetime
from pathlib import Path
from typing import Optional, Tuple

from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

# Initialize logger
logger = logging.getLogger(__name__)


class CryptoService:
    """
    Crypto Service - Cryptographic operations.

    Features:
    - AES-256-CBC encryption using Fernet
    - SHA-256 hashing for pseudonyms and evidence
    - Secure random generation
    - Key derivation
    - Evidence integrity verification
    """

    def __init__(self, master_key: Optional[bytes] = None):
        """
        Initialize crypto service.

        Args:
            master_key: Master encryption key (32 bytes).
                        If None, generates a new key (for testing only).
        """
```

```
    if master_key:
        self.master_key = master_key
    else:
        # Generate new key (WARNING: for testing/development only)
        self.master_key = Fernet.generate_key()
        logger.warning(
            "CryptoService initialized with auto-generated key. "
            "This should only be used for testing!"
        )

    # Initialize Fernet cipher
    self.cipher = Fernet(self.master_key)

    logger.info("CryptoService initialized")

def generate_pseudonym(self, submission_id: str, salt: Optional[str] = None) -> str:
    """
    Generate anonymous pseudonym using SHA-256.

    Creates a collision-resistant pseudonym that preserves anonymity
    while allowing submission tracking.

    Args:
        submission_id: Unique submission identifier
        salt: Optional salt for additional randomness

    Returns:
        str: Hex-encoded pseudonym (e.g., "whistleblower-a3b5c7...")
    """
    try:
        # Create input for hash
        hash_input = submission_id.encode('utf-8')

        # Add salt if provided
        if salt:
            hash_input += salt.encode('utf-8')
        else:
            # Generate deterministic salt from submission_id
            salt_hash = hashlib.sha256(submission_id.encode()).digest()
            hash_input += salt_hash

        # Calculate SHA-256 hash
        hash_object = hashlib.sha256(hash_input)
        hash_hex = hash_object.hexdigest()

        # Create readable pseudonym (first 12 chars for readability)
```

```
            pseudonym = f"whistleblower-{hash_hex[:12]}"

            logger.debug(f"Generated pseudonym: {pseudonym}")

            return pseudonym

        except Exception as e:
            logger.error(f"Failed to generate pseudonym: {e}")
            # Fallback pseudonym
            return f"whistleblower-{secrets.token_hex(6)}"

    def hash_data(self, data: bytes) -> str:
        """
        Calculate SHA-256 hash of data.

        Args:
            data: Raw data bytes

        Returns:
            str: Hex-encoded SHA-256 hash
        """
        try:
            hash_object = hashlib.sha256(data)
            return hash_object.hexdigest()

        except Exception as e:
            logger.error(f"Failed to hash data: {e}")
            raise ValueError(f"Hash calculation failed: {str(e)}")

    def hash_file(self, file_path: Path) -> str:
        """
        Calculate SHA-256 hash of file.

        Uses streaming to handle large files efficiently.

        Args:
            file_path: Path to file

        Returns:
            str: Hex-encoded SHA-256 hash

        Raises:
            FileNotFoundError: If file doesn't exist
            IOError: If file read fails
        """
        try:
```

```
            if not file_path.exists():
                raise FileNotFoundError(f"File not found: {file_path}")

            hash_object = hashlib.sha256()

            # Stream file in chunks (64KB)
            chunk_size = 65536

            with open(file_path, 'rb') as f:
                while True:
                    chunk = f.read(chunk_size)
                    if not chunk:
                        break
                    hash_object.update(chunk)

            file_hash = hash_object.hexdigest()

            logger.debug(f"File hash: {file_hash[:16]}... (size={file_path.stat().st_size}
bytes)")

            return file_hash

        except FileNotFoundError:
            raise
        except Exception as e:
            logger.error(f"Failed to hash file {file_path}: {e}")
            raise IOError(f"File hash failed: {str(e)}")

    def encrypt_data(self, data: bytes) -> bytes:
        """
        Encrypt data using AES-256-CBC (via Fernet).

        Args:
            data: Raw data to encrypt

        Returns:
            bytes: Encrypted data (includes IV and MAC)

        Raises:
            ValueError: If encryption fails
        """
        try:
            encrypted = self.cipher.encrypt(data)
            logger.debug(f"Encrypted {len(data)} bytes !' {len(encrypted)} bytes")
            return encrypted
```

```
        except Exception as e:
            logger.error(f"Encryption failed: {e}")
            raise ValueError(f"Encryption failed: {str(e)}")

    def decrypt_data(self, encrypted_data: bytes) -> bytes:
        """
        Decrypt data encrypted with encrypt_data().

        Args:
            encrypted_data: Encrypted data

        Returns:
            bytes: Decrypted data

        Raises:
            ValueError: If decryption fails (wrong key or corrupted data)
        """
        try:
            decrypted = self.cipher.decrypt(encrypted_data)
            logger.debug(f"Decrypted {len(encrypted_data)} bytes !' {len(decrypted)} bytes")
            return decrypted

        except Exception as e:
            logger.error(f"Decryption failed: {e}")
            raise ValueError(f"Decryption failed (invalid key or corrupted data): {str(e)}")

    def encrypt_file(self, input_path: Path, output_path: Path) -> str:
        """
        Encrypt file and save to output path.

        Args:
            input_path: Path to file to encrypt
            output_path: Path to save encrypted file

        Returns:
            str: SHA-256 hash of original file

        Raises:
            FileNotFoundError: If input file doesn't exist
            IOError: If encryption or write fails
        """
        try:
            if not input_path.exists():
                raise FileNotFoundError(f"Input file not found: {input_path}")

            # Read file
```

```
            with open(input_path, 'rb') as f:
                data = f.read()

            # Calculate hash before encryption
            file_hash = self.hash_data(data)

            # Encrypt
            encrypted = self.encrypt_data(data)

            # Write encrypted file
            output_path.parent.mkdir(parents=True, exist_ok=True)
            with open(output_path, 'wb') as f:
                f.write(encrypted)

            logger.info(
                f"File encrypted: {input_path.name} !' {output_path.name} "
                f"({len(data)} !' {len(encrypted)} bytes)"
            )

            return file_hash

        except FileNotFoundError:
            raise
        except Exception as e:
            logger.error(f"File encryption failed: {e}")
            raise IOError(f"File encryption failed: {str(e)}")

    def decrypt_file(self, input_path: Path, output_path: Path) -> None:
        """
        Decrypt file and save to output path.

        Args:
            input_path: Path to encrypted file
            output_path: Path to save decrypted file

        Raises:
            FileNotFoundError: If input file doesn't exist
            ValueError: If decryption fails
            IOError: If write fails
        """
        try:
            if not input_path.exists():
                raise FileNotFoundError(f"Input file not found: {input_path}")

            # Read encrypted file
            with open(input_path, 'rb') as f:
```

```
            encrypted = f.read()

        # Decrypt
        decrypted = self.decrypt_data(encrypted)

        # Write decrypted file
        output_path.parent.mkdir(parents=True, exist_ok=True)
        with open(output_path, 'wb') as f:
            f.write(decrypted)

        logger.info(
            f"File decrypted: {input_path.name} !' {output_path.name} "
            f"({len(encrypted)} !' {len(decrypted)} bytes)"
        )

    except (FileNotFoundError, ValueError):
        raise
    except Exception as e:
        logger.error(f"File decryption failed: {e}")
        raise IOError(f"File decryption failed: {str(e)}")

def generate_secure_token(self, length: int = 32) -> str:
    """
    Generate cryptographically secure random token.

    Args:
        length: Token length in bytes (default: 32)

    Returns:
        str: Hex-encoded random token
    """
    return secrets.token_hex(length)

def derive_key(self, password: str, salt: bytes, iterations: int = 100000) -> bytes:
    """
    Derive encryption key from password using PBKDF2.

    Args:
        password: User password
        salt: Random salt (at least 16 bytes)
        iterations: PBKDF2 iteration count

    Returns:
        bytes: Derived key (32 bytes)
    """
    try:
```

```
            kdf = PBKDF2HMAC(
                algorithm=hashes.SHA256(),
                length=32,
                salt=salt,
                iterations=iterations
            )

            key = kdf.derive(password.encode('utf-8'))

            logger.debug(f"Key derived from password (iterations={iterations})")

            return key

        except Exception as e:
            logger.error(f"Key derivation failed: {e}")
            raise ValueError(f"Key derivation failed: {str(e)}")

    def verify_hash(self, data: bytes, expected_hash: str) -> bool:
        """
        Verify data matches expected hash.

        Args:
            data: Data to verify
            expected_hash: Expected SHA-256 hash (hex)

        Returns:
            bool: True if hash matches, False otherwise
        """
        try:
            actual_hash = self.hash_data(data)
            return secrets.compare_digest(actual_hash, expected_hash)

        except Exception as e:
            logger.error(f"Hash verification failed: {e}")
            return False

    def create_submission_hash_chain(self, *data_items: bytes) -> str:
        """
        Create chained hash of multiple data items.

        Useful for creating composite hashes of submission + evidence + metadata.

        Args:
            *data_items: Variable number of data items to hash

        Returns:
```

```
            str: Hex-encoded chained hash
        """
        try:
            hash_object = hashlib.sha256()

            for item in data_items:
                hash_object.update(item)

            return hash_object.hexdigest()

        except Exception as e:
            logger.error(f"Chain hash failed: {e}")
            raise ValueError(f"Chain hash failed: {str(e)}")

    @staticmethod
    def generate_master_key() -> bytes:
        """
        Generate new master encryption key.

        WARNING: This key must be securely stored! Loss of key means
        loss of all encrypted data.

        Returns:
            bytes: 32-byte master key suitable for Fernet
        """
        return Fernet.generate_key()

    @staticmethod
    def key_to_string(key: bytes) -> str:
        """
        Convert key to base64 string for storage.

        Args:
            key: Encryption key

        Returns:
            str: Base64-encoded key
        """
        return key.decode('utf-8')

    @staticmethod
    def string_to_key(key_string: str) -> bytes:
        """
        Convert base64 string back to key.

        Args:
```

```
            key_string: Base64-encoded key

        Returns:
            bytes: Encryption key
        """
        return key_string.encode('utf-8')

"""
Hash Chain Service - Blockchain-like immutable chain

Provides:
- SHA-256 hash chain for evidence custody
- Genesis block initialization
- Block addition with chaining
- Chain verification
- Tamper detection
- Proof generation
"""

import hashlib
import json
import logging
import threading
from datetime import datetime
from pathlib import Path
from typing import Dict, List, Optional
from filelock import FileLock

# Initialize logger
logger = logging.getLogger(__name__)


class HashChainService:
    """
    Hash Chain Service - Immutable custody chain.

    Features:
    - SHA-256 hash chain (blockchain-like)
    - Genesis block initialization
    - Recursive chaining with previous hash
    - Chain integrity verification
    - Tamper detection
    - Proof of custody generation
    """

    def __init__(self, data_dir: Optional[Path] = None):
```

```
    """
    Initialize hash chain service.

    Args:
        data_dir: Base data directory (defaults to backend/data)
    """
    if data_dir:
        self.data_dir = Path(data_dir)
    else:
        self.data_dir = Path(__file__).parent.parent / "data"

    self.chain_file = self.data_dir / "chain.json"
    self.chain_lock = threading.Lock()

    # Initialize chain
    self._initialize_chain()

    logger.info(f"HashChainService initialized (chain_file={self.chain_file})")

def _initialize_chain(self) -> None:
    """Initialize hash chain with genesis block if not exists."""
    try:
        if not self.chain_file.exists():
            logger.info("Creating genesis block...")

            genesis_block = self._create_genesis_block()
            chain_data = {
                'blocks': [genesis_block],
                'version': '1.0.0',
                'created': datetime.utcnow().isoformat()
            }

            self._save_chain(chain_data)
            logger.info(f"Genesis block created: {genesis_block['hash'][:16]}...")
        else:
            # Verify existing chain
            self.verify_chain()

    except Exception as e:
        logger.error(f"Failed to initialize chain: {e}")
        raise

def _create_genesis_block(self) -> Dict:
    """
    Create genesis block (first block in chain).
```

```
        Returns:
            dict: Genesis block
        """
        genesis = {
            'index': 0,
            'timestamp': datetime.utcnow().isoformat(),
            'submission_id': 'GENESIS',
            'evidence_hash': '0' * 64,
            'previous_hash': '0' * 64,
            'nonce': 0
        }

        # Calculate genesis hash
        genesis['hash'] = self._calculate_block_hash(genesis)

        return genesis

    def add_block(
        self,
        submission_id: str,
        evidence_hash: str,
        metadata: Optional[Dict] = None
    ) -> Dict:
        """
        Add new block to chain.

        Args:
            submission_id: Unique submission identifier
            evidence_hash: SHA-256 hash of evidence
            metadata: Optional metadata to include

        Returns:
            dict: New block

        Raises:
            ValueError: If block addition fails
        """
        try:
            with self.chain_lock:
                chain_data = self.load_chain()
                blocks = chain_data.get('blocks', [])

                # Get previous block
                if not blocks:
                    raise ValueError("Chain has no blocks (genesis missing)")
```

```
            previous_block = blocks[-1]

            # Create new block
            new_block = {
                'index': len(blocks),
                'timestamp': datetime.utcnow().isoformat(),
                'submission_id': submission_id,
                'evidence_hash': evidence_hash,
                'previous_hash': previous_block['hash'],
                'metadata': metadata or {}
            }

            # Calculate hash
            new_block['hash'] = self._calculate_block_hash(new_block)

            # Add to chain
            blocks.append(new_block)
            chain_data['blocks'] = blocks

            # Save chain
            self._save_chain(chain_data)

            logger.info(
                f"Added block {new_block['index']} for {submission_id} "
                f"(hash={new_block['hash'][:16]}...)"
            )

            return new_block

        except Exception as e:
            logger.error(f"Failed to add block: {e}")
            raise ValueError(f"Failed to add block: {str(e)}")

    def verify_chain(self) -> bool:
        """
        Verify integrity of entire chain.

        Returns:
            bool: True if chain is valid, False otherwise

        Raises:
            ValueError: If chain is corrupted
        """
        try:
            chain_data = self.load_chain()
            blocks = chain_data.get('blocks', [])
```

```
            if not blocks:
                raise ValueError("Chain is empty")

            # Verify genesis block
            genesis = blocks[0]
            if genesis['index'] != 0:
                raise ValueError("Invalid genesis block index")

            # Verify each block
            for i in range(1, len(blocks)):
                current_block = blocks[i]
                previous_block = blocks[i - 1]

                # Check index continuity
                if current_block['index'] != i:
                    raise ValueError(f"Block {i} has invalid index: {current_block['index']}")

                # Check previous hash linkage
                if current_block['previous_hash'] != previous_block['hash']:
                    raise ValueError(
                        f"Block {i} has invalid previous_hash: "
                        f"expected {previous_block['hash']}, "
                        f"got {current_block['previous_hash']}"
                    )

                # Verify block hash
                calculated_hash = self._calculate_block_hash(current_block)
                if current_block['hash'] != calculated_hash:
                    raise ValueError(
                        f"Block {i} has invalid hash: "
                        f"expected {calculated_hash}, "
                        f"got {current_block['hash']}"
                    )

            logger.debug(f"Chain verified successfully ({len(blocks)} blocks)")
            return True

        except ValueError as e:
            logger.error(f"Chain verification failed: {e}")
            raise
        except Exception as e:
            logger.error(f"Chain verification error: {e}")
            return False

    def get_proof(self, submission_id: str) -> Dict:
```

```
        """
        Get chain proof for submission.

        Args:
            submission_id: Submission identifier

        Returns:
            dict: Chain proof with block details
        """
        try:
            chain_data = self.load_chain()
            blocks = chain_data.get('blocks', [])

            # Find block for submission
            for block in blocks:
                if block.get('submission_id') == submission_id:
                    return {
                        'block_index': block['index'],
                        'block_hash': block['hash'],
                        'previous_hash': block['previous_hash'],
                        'timestamp': block['timestamp'],
                        'evidence_hash': block['evidence_hash'],
                        'chain_length': len(blocks),
                        'verified': True
                    }

            logger.debug(f"No proof found for submission {submission_id}")
            return {}

        except Exception as e:
            logger.error(f"Failed to get proof: {e}")
            return {}

    def get_chain_statistics(self) -> Dict:
        """
        Get chain statistics.

        Returns:
            dict: Chain statistics
        """
        try:
            chain_data = self.load_chain()
            blocks = chain_data.get('blocks', [])

            # Calculate statistics
            if blocks:
```

```
            first_block = blocks[0]
            last_block = blocks[-1]

            first_time = datetime.fromisoformat(first_block['timestamp'])
            last_time = datetime.fromisoformat(last_block['timestamp'])
            chain_age_seconds = (last_time - first_time).total_seconds()
        else:
            chain_age_seconds = 0

        stats = {
            'total_blocks': len(blocks),
            'chain_version': chain_data.get('version', '1.0.0'),
            'genesis_timestamp': blocks[0]['timestamp'] if blocks else None,
            'latest_timestamp': blocks[-1]['timestamp'] if blocks else None,
            'chain_age_seconds': chain_age_seconds,
            'chain_age_hours': round(chain_age_seconds / 3600, 2),
            'average_block_time_seconds': (
                chain_age_seconds / (len(blocks) - 1)
                if len(blocks) > 1 else 0
            ),
            'chain_verified': self.verify_chain(),
            'timestamp': datetime.utcnow().isoformat()
        }

        return stats

    except Exception as e:
        logger.error(f"Failed to get chain statistics: {e}")
        return {
            'error': str(e),
            'timestamp': datetime.utcnow().isoformat()
        }

def load_chain(self) -> Dict:
    """
    Load chain data from file.

    Returns:
        dict: Chain data with blocks
    """
    try:
        if not self.chain_file.exists():
            logger.warning("Chain file not found")
            return {'blocks': []}

        lock_path = self.chain_file.with_suffix('.lock')
```

```
            lock = FileLock(str(lock_path), timeout=10)

            with lock:
                with open(self.chain_file, 'r', encoding='utf-8') as f:
                    chain_data = json.load(f)

            return chain_data

        except Exception as e:
            logger.error(f"Failed to load chain: {e}")
            return {'blocks': []}

    def _save_chain(self, chain_data: Dict) -> None:
        """
        Save chain data to file with atomic write.

        Args:
            chain_data: Chain data to save
        """
        try:
            lock_path = self.chain_file.with_suffix('.lock')
            lock = FileLock(str(lock_path), timeout=10)

            with lock:
                # Write to temp file first
                temp_path = self.chain_file.with_suffix('.tmp')

                with open(temp_path, 'w', encoding='utf-8') as f:
                    json.dump(chain_data, f, indent=2, ensure_ascii=False)

                # Atomic rename
                temp_path.replace(self.chain_file)

        except Exception as e:
            logger.error(f"Failed to save chain: {e}")
            raise

    def _calculate_block_hash(self, block: Dict) -> str:
        """
        Calculate SHA-256 hash of block.

        Args:
            block: Block dictionary

        Returns:
            str: Hex-encoded SHA-256 hash
```

```
        """
        # Create deterministic string representation
        # Exclude 'hash' field itself
        block_copy = {k: v for k, v in block.items() if k != 'hash'}

        # Sort keys for consistency
        block_string = json.dumps(block_copy, sort_keys=True, ensure_ascii=False)

        # Calculate SHA-256
        hash_object = hashlib.sha256(block_string.encode('utf-8'))
        return hash_object.hexdigest()

    def export_chain(self, output_path: Path) -> bool:
        """
        Export chain to file.

        Args:
            output_path: Path to export file

        Returns:
            bool: True if successful
        """
        try:
            chain_data = self.load_chain()

            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(chain_data, f, indent=2, ensure_ascii=False)

            logger.info(f"Chain exported to {output_path}")
            return True

        except Exception as e:
            logger.error(f"Failed to export chain: {e}")
            return False

    def rebuild_chain(self, backup_path: Path) -> bool:
        """
        Rebuild chain from backup.

        Args:
            backup_path: Path to backup file

        Returns:
            bool: True if successful
        """
        try:
```

```
            with open(backup_path, 'r', encoding='utf-8') as f:
                chain_data = json.load(f)

            # Verify chain before rebuilding
            blocks = chain_data.get('blocks', [])

            if not blocks:
                raise ValueError("Backup chain is empty")

            # Save backup chain
            self._save_chain(chain_data)

            # Verify rebuilt chain
            self.verify_chain()

            logger.info(f"Chain rebuilt from {backup_path}")
            return True

        except Exception as e:
            logger.error(f"Failed to rebuild chain: {e}")
            return False
"""
Metadata Service - File metadata removal

Provides:
- EXIF data stripping from images
- GPS location removal
- Camera/device information removal
- Timestamp sanitization
- Metadata anonymization
"""

import logging
from datetime import datetime
from pathlib import Path
from typing import Dict, Optional

from PIL import Image
from PIL.ExifTags import TAGS

# Initialize logger
logger = logging.getLogger(__name__)


class MetadataService:
```

```
"""
Metadata Service - Remove sensitive metadata from files.

Features:
- EXIF data removal from images (JPEG, PNG, TIFF, etc.)
- GPS location stripping
- Camera model/serial number removal
- Timestamp anonymization
- File metadata sanitization

Protects whistleblower anonymity by removing identifying information.
"""


def __init__(self):
    """Initialize metadata service."""
    logger.info("MetadataService initialized")

def strip_image_metadata(
    self,
    input_path: Path,
    output_path: Path,
    preserve_orientation: bool = True
) -> Dict:
    """
    Strip all EXIF/metadata from image file.

    Args:
        input_path: Path to input image
        output_path: Path to save cleaned image
        preserve_orientation: Keep image orientation (recommended)

    Returns:
        dict: Report of stripped metadata

    Raises:
        FileNotFoundError: If input file doesn't exist
        ValueError: If file is not a valid image
        IOError: If processing fails
    """
    try:
        if not input_path.exists():
            raise FileNotFoundError(f"Input file not found: {input_path}")

        # Open image
        with Image.open(input_path) as img:
            # Extract metadata before stripping (for logging)
```

```
        original_metadata = self._extract_metadata(img)

        # Get image format
        img_format = img.format or 'JPEG'

        # Handle orientation if needed
        if preserve_orientation and hasattr(img, '_getexif'):
            img = self._correct_orientation(img)

        # Create new image without metadata
        # Convert to RGB if necessary (removes alpha channel issues)
        if img.mode not in ('RGB', 'L'):
            if img.mode == 'RGBA':
                # Create white background
                background = Image.new('RGB', img.size, (255, 255, 255))
                background.paste(img, mask=img.split()[3])  # Use alpha as mask
                img = background
            else:
                img = img.convert('RGB')

        # Save without metadata
        output_path.parent.mkdir(parents=True, exist_ok=True)

        # Save parameters (no EXIF)
        save_kwargs = {
            'format': img_format,
            'quality': 95,  # High quality
            'optimize': True
        }

        # For JPEG, explicitly disable EXIF
        if img_format.upper() in ('JPEG', 'JPG'):
            save_kwargs['exif'] = b''

        img.save(output_path, **save_kwargs)

    # Verify metadata was removed
    cleaned_metadata = self._get_metadata_summary(output_path)

    report = {
        'input_file': str(input_path),
        'output_file': str(output_path),
        'original_metadata_count': len(original_metadata),
        'cleaned_metadata_count': len(cleaned_metadata),
        'metadata_removed': original_metadata,
        'sensitive_data_stripped': self._identify_sensitive_fields(original_metadata),
```

```
            'timestamp': datetime.utcnow().isoformat()
        }

        logger.info(
            f"Metadata stripped: {input_path.name} !' {output_path.name} "
            f"({len(original_metadata)} fields removed)"
        )

        return report

    except FileNotFoundError:
        raise
    except Exception as e:
        logger.error(f"Metadata stripping failed for {input_path}: {e}")
        raise IOError(f"Metadata stripping failed: {str(e)}")

def _extract_metadata(self, img: Image.Image) -> Dict:
    """
    Extract all EXIF/metadata from image.

    Args:
        img: PIL Image object

    Returns:
        dict: Metadata key-value pairs
    """
    metadata = {}

    try:
        # Try to get EXIF data
        exif_data = img._getexif()

        if exif_data:
            for tag_id, value in exif_data.items():
                # Get human-readable tag name
                tag = TAGS.get(tag_id, tag_id)

                # Convert bytes to string for logging
                if isinstance(value, bytes):
                    try:
                        value = value.decode('utf-8', errors='ignore')
                    except:
                        value = f"<binary data: {len(value)} bytes>"

                metadata[tag] = str(value)
```

```
        except AttributeError:
            # No EXIF data
            pass
        except Exception as e:
            logger.warning(f"Failed to extract metadata: {e}")

        return metadata

    def _correct_orientation(self, img: Image.Image) -> Image.Image:
        """
        Correct image orientation based on EXIF data.

        Args:
            img: PIL Image object

        Returns:
            Image: Oriented image
        """
        try:
            exif = img._getexif()

            if exif:
                # Orientation tag
                orientation_tag = 0x0112

                if orientation_tag in exif:
                    orientation = exif[orientation_tag]

                    # Rotation mappings
                    rotations = {
                        3: 180,
                        6: 270,
                        8: 90
                    }

                    if orientation in rotations:
                        img = img.rotate(rotations[orientation], expand=True)
                        logger.debug(f"Corrected orientation: {orientation}")

        except Exception as e:
            logger.warning(f"Orientation correction failed: {e}")

        return img

    def _get_metadata_summary(self, image_path: Path) -> Dict:
        """
```

```
    Get summary of remaining metadata in image.

    Args:
        image_path: Path to image

    Returns:
        dict: Metadata summary
    """
    try:
        with Image.open(image_path) as img:
            return self._extract_metadata(img)
    except Exception as e:
        logger.warning(f"Failed to get metadata summary: {e}")
        return {}

def _identify_sensitive_fields(self, metadata: Dict) -> list:
    """
    Identify sensitive metadata fields that were removed.

    Args:
        metadata: Metadata dictionary

    Returns:
        list: List of sensitive field names
    """
    sensitive_fields = []

    # Common sensitive EXIF tags
    sensitive_tags = [
        'GPSInfo',          # GPS location
        'GPSLatitude',
        'GPSLongitude',
        'GPSAltitude',
        'GPSTimeStamp',
        'Make',             # Camera manufacturer
        'Model',            # Camera model
        'Software',         # Editing software
        'DateTime',         # Capture time
        'DateTimeOriginal',
        'DateTimeDigitized',
        'HostComputer',     # Computer name
        'Copyright',        # Copyright info
        'Artist',           # Artist/creator
        'CameraSerialNumber',
        'LensSerialNumber',
        'OwnerName',
```

```
            'UserComment'
        ]

        for tag in sensitive_tags:
            if tag in metadata:
                sensitive_fields.append(tag)

        return sensitive_fields

    def has_gps_data(self, image_path: Path) -> bool:
        """
        Check if image contains GPS location data.

        Args:
            image_path: Path to image

        Returns:
            bool: True if GPS data present, False otherwise
        """
        try:
            with Image.open(image_path) as img:
                metadata = self._extract_metadata(img)

                gps_tags = ['GPSInfo', 'GPSLatitude', 'GPSLongitude']
                return any(tag in metadata for tag in gps_tags)

        except Exception as e:
            logger.warning(f"GPS check failed: {e}")
            return False

    def get_metadata_report(self, image_path: Path) -> Dict:
        """
        Get detailed metadata report for image.

        Args:
            image_path: Path to image

        Returns:
            dict: Metadata report
        """
        try:
            with Image.open(image_path) as img:
                metadata = self._extract_metadata(img)

                report = {
                    'file_path': str(image_path),
```

```
                'format': img.format,
                'size': img.size,
                'mode': img.mode,
                'metadata_count': len(metadata),
                'has_gps': any(k.startswith('GPS') for k in metadata.keys()),
                'has_camera_info': 'Make' in metadata or 'Model' in metadata,
                'has_timestamp': any(
                    k in metadata for k in ['DateTime', 'DateTimeOriginal']
                ),
                'sensitive_fields': self._identify_sensitive_fields(metadata),
                'all_metadata': metadata
            }

            return report

        except Exception as e:
            logger.error(f"Failed to get metadata report: {e}")
            return {
                'error': str(e),
                'file_path': str(image_path)
            }

    def sanitize_filename(self, filename: str) -> str:
        """
        Sanitize filename to remove identifying information.

        Args:
            filename: Original filename

        Returns:
            str: Sanitized filename
        """
        # Get extension
        path = Path(filename)
        extension = path.suffix

        # Generate anonymous name
        timestamp = datetime.utcnow().strftime('%Y%m%d_%H%M%S')
        sanitized = f"evidence_{timestamp}{extension}"

        logger.debug(f"Filename sanitized: {filename} !' {sanitized}")

        return sanitized

    def batch_strip_metadata(
        self,
```

```
    input_dir: Path,
    output_dir: Path,
    extensions: Optional[list] = None
) -> Dict:
    """
    Strip metadata from all images in directory.

    Args:
        input_dir: Input directory
        output_dir: Output directory
        extensions: Image extensions to process (default: jpg, png, jpeg)

    Returns:
        dict: Batch processing report
    """
    if extensions is None:
        extensions = ['.jpg', '.jpeg', '.png', '.tiff', '.bmp']

    processed = []
    failed = []

    try:
        output_dir.mkdir(parents=True, exist_ok=True)

        # Find all images
        image_files = []
        for ext in extensions:
            image_files.extend(input_dir.glob(f"*{ext}"))
            image_files.extend(input_dir.glob(f"*{ext.upper()}"))

        logger.info(f"Found {len(image_files)} images to process")

        # Process each image
        for input_path in image_files:
            try:
                output_path = output_dir / input_path.name

                report = self.strip_image_metadata(input_path, output_path)
                processed.append(report)

            except Exception as e:
                logger.error(f"Failed to process {input_path}: {e}")
                failed.append({
                    'file': str(input_path),
                    'error': str(e)
                })
```

```
            batch_report = {
                'input_directory': str(input_dir),
                'output_directory': str(output_dir),
                'total_files': len(image_files),
                'processed': len(processed),
                'failed': len(failed),
                'processed_files': processed,
                'failed_files': failed,
                'timestamp': datetime.utcnow().isoformat()
            }

            logger.info(
                f"Batch processing complete: {len(processed)} processed, "
                f"{len(failed)} failed"
            )

            return batch_report

        except Exception as e:
            logger.error(f"Batch processing failed: {e}")
            return {
                'error': str(e),
                'processed': len(processed),
                'failed': len(failed)
            }

"""
Metrics Service - Performance metrics tracking with Prometheus integration

Provides:
- Request latency tracking
- Throughput monitoring
- Memory usage tracking
- Model inference time tracking
- System resource monitoring
- Metrics persistence
- Prometheus metrics export
"""

import logging
import os
import psutil
import time
from collections import defaultdict, deque
from datetime import datetime, timedelta
```

```
from pathlib import Path
from typing import Dict, List, Optional, Any

# Prometheus imports (optional - graceful degradation if not available)
try:
    from prometheus_client import Counter, Gauge, Histogram
    PROMETHEUS_AVAILABLE = True
except ImportError:
    PROMETHEUS_AVAILABLE = False
    Counter = Gauge = Histogram = None

# Initialize logger
logger = logging.getLogger(__name__)


class MetricsService:
    """
    Metrics Service - Performance and system metrics tracking with Prometheus
integration.

    Features:
    - Request latency tracking (percentiles)
    - Throughput monitoring (requests per time window)
    - Memory usage tracking
    - Model inference time tracking
    - Error rate monitoring
    - Metrics aggregation and persistence
    - Prometheus metrics export (if available)
    """

    def __init__(
        self,
        metrics_file: Optional[Path] = None,
        window_size: int = 1000,
        persist_interval: int = 300,
        prometheus_metrics: Optional[Dict[str, Any]] = None
    ):
        """
        Initialize metrics service.

        Args:
            metrics_file: Path to metrics persistence file
            window_size: Number of recent metrics to keep in memory
            persist_interval: Persist metrics every N seconds
            prometheus_metrics: Dict of Prometheus metric objects from main.py
        """
```

```
        self.metrics_file = metrics_file
        self.window_size = window_size
        self.persist_interval = persist_interval

        # Prometheus metrics (passed from main.py)
        self.prometheus_metrics = prometheus_metrics or {}
        self.prometheus_enabled = PROMETHEUS_AVAILABLE and
bool(prometheus_metrics)

        if self.prometheus_enabled:
            logger.info("Prometheus metrics integration enabled")
        else:
            logger.info("Prometheus metrics not available - using local metrics only")

        # Request metrics (sliding window)
        self.request_latencies: deque = deque(maxlen=window_size)
        self.request_timestamps: deque = deque(maxlen=window_size)

        # Model metrics
        self.model_inference_times: Dict[str, deque] = defaultdict(
            lambda: deque(maxlen=window_size)
        )

        # Endpoint metrics
        self.endpoint_counts: Dict[str, int] = defaultdict(int)
        self.endpoint_errors: Dict[str, int] = defaultdict(int)

        # Submission metrics
        self.submission_stats = {
            'total': 0,
            'pending': 0,
            'completed': 0,
            'failed': 0
        }

        # Layer-specific metrics
        self.layer_metrics = {
            'layer1_anonymity_violations': 0,
            'layer2_credibility_scores': deque(maxlen=100),
            'layer2_deepfakes_detected': 0,
            'layer3_coordination_detected': 0,
            'layer4_consensus_iterations': deque(maxlen=100),
            'layer4_convergence_times': deque(maxlen=100),
            'layer5_counter_evidence_count': 0,
            'layer5_impacts': deque(maxlen=100),
            'layer6_reports_generated': 0
```

```
        }

        # Security metrics
        self.security_metrics = {
            'hash_chain_failures': 0,
            'validation_failures': defaultdict(int),
            'crypto_failures': defaultdict(int),
            'rate_limit_blocks': defaultdict(int)
        }

        # Storage metrics
        self.storage_metrics = {
            'corruption_detected': 0,
            'index_inconsistencies': 0,
            'backup_failures': 0
        }

        # Queue metrics
        self.queue_metrics = {
            'pending_jobs': 0,
            'processing_jobs': 0,
            'job_failures': defaultdict(int)
        }

        # Research/evaluation metrics
        self.evaluation_metrics = {
            'auroc_score': 0.0,
            'precision_score': 0.0,
            'recall_score': 0.0
        }

        # System metrics
        self.process = psutil.Process(os.getpid())
        self.generic_gauges: Dict[str, float] = {}
        self.start_time = time.time()
        self.last_persist_time = time.time()

        # Counters
        self.total_requests = 0
        self.total_errors = 0

        logger.info("MetricsService initialized")

    # ==================== REQUEST METRICS ====================

    def record_request(
```

```
    self,
    endpoint: str,
    latency: float,
    status_code: int
) -> None:
    """
    Record HTTP request metrics.

    Args:
        endpoint: API endpoint path
        latency: Request latency in seconds
        status_code: HTTP status code
    """
    try:
        # Record latency
        self.request_latencies.append(latency)
        self.request_timestamps.append(time.time())

        # Record endpoint
        self.endpoint_counts[endpoint] += 1

        # Record errors (4xx, 5xx)
        if status_code >= 400:
            self.endpoint_errors[endpoint] += 1
            self.total_errors += 1

        self.total_requests += 1

        # Check if should persist
        if time.time() - self.last_persist_time > self.persist_interval:
            self._persist_metrics()

    except Exception as e:
        logger.error(f"Failed to record request metric: {e}")

def record_model_inference(
    self,
    model_name: str,
    inference_time: float
) -> None:
    """
    Record model inference time.

    Args:
        model_name: Name of ML model
        inference_time: Inference time in seconds
```

```
    """
    try:
        self.model_inference_times[model_name].append(inference_time)

        logger.debug(
            f"Model inference: {model_name} "
            f"({inference_time:.3f}s)"
        )

    except Exception as e:
        logger.error(f"Failed to record model metric: {e}")

def record_submission(self, status: str) -> None:
    """
    Record submission status change.

    Args:
        status: Submission status (pending/completed/failed)
    """
    try:
        self.submission_stats['total'] += 1

        if status in self.submission_stats:
            self.submission_stats[status] += 1

        # Update Prometheus metric
        if self.prometheus_enabled and 'submission_total' in self.prometheus_metrics:
            self.prometheus_metrics['submission_total'].labels(status=status).inc()

    except Exception as e:
        logger.error(f"Failed to record submission metric: {e}")

# ==================== LAYER-SPECIFIC METRICS ====================

def record_anonymity_violation(self) -> None:
    """Record Layer 1 anonymity violation."""
    try:
        self.layer_metrics['layer1_anonymity_violations'] += 1
        logger.warning("Anonymity violation detected")
    except Exception as e:
        logger.error(f"Failed to record anonymity violation: {e}")

def update_credibility_score(self, score: float) -> None:
    """
    Update Layer 2 credibility score.
```

```
        Args:
            score: Credibility score (0.0 to 1.0)
        """
        try:
            self.layer_metrics['layer2_credibility_scores'].append(score)

            # Update Prometheus gauge with average
            if self.prometheus_enabled and 'layer2_score' in self.prometheus_metrics:
                scores = self.layer_metrics['layer2_credibility_scores']
                avg_score = sum(scores) / len(scores) if scores else 0.0
                self.prometheus_metrics['layer2_score'].set(avg_score)

            logger.debug(f"Credibility score updated: {score:.3f}")

        except Exception as e:
            logger.error(f"Failed to update credibility score: {e}")

    def record_deepfake_detection(self) -> None:
        """Record Layer 2 deepfake detection."""
        try:
            self.layer_metrics['layer2_deepfakes_detected'] += 1
            logger.info("Deepfake detected")
        except Exception as e:
            logger.error(f"Failed to record deepfake detection: {e}")

    def record_coordination_detection(self) -> None:
        """Record Layer 3 coordination detection."""
        try:
            self.layer_metrics['layer3_coordination_detected'] += 1

            # Update Prometheus counter
            if self.prometheus_enabled and 'layer3_coordination' in
self.prometheus_metrics:
                self.prometheus_metrics['layer3_coordination'].inc()

            logger.warning("Coordinated attack detected")

        except Exception as e:
            logger.error(f"Failed to record coordination detection: {e}")

    def update_consensus_metrics(
        self,
        iterations: int,
        convergence_time: float
    ) -> None:
        """
```

```
    Update Layer 4 consensus metrics.

    Args:
        iterations: Number of consensus iterations
        convergence_time: Time to reach consensus (seconds)
    """
    try:
        self.layer_metrics['layer4_consensus_iterations'].append(iterations)
        self.layer_metrics['layer4_convergence_times'].append(convergence_time)

        # Update Prometheus gauges
        if self.prometheus_enabled:
            if 'layer4_iterations' in self.prometheus_metrics:
                iter_list = self.layer_metrics['layer4_consensus_iterations']
                avg_iterations = sum(iter_list) / len(iter_list) if iter_list else 0.0
                self.prometheus_metrics['layer4_iterations'].set(avg_iterations)

            if 'layer4_convergence' in self.prometheus_metrics:
                self.prometheus_metrics['layer4_convergence'].set(convergence_time)

        logger.debug(
            f"Consensus metrics: {iterations} iterations, "
            f"{convergence_time:.3f}s convergence time"
        )

    except Exception as e:
        logger.error(f"Failed to update consensus metrics: {e}")

def record_counter_evidence(self, impact_percent: float) -> None:
    """
    Record Layer 5 counter-evidence submission.

    Args:
        impact_percent: Impact percentage on credibility score
    """
    try:
        self.layer_metrics['layer5_counter_evidence_count'] += 1
        self.layer_metrics['layer5_impacts'].append(impact_percent)

        # Update Prometheus metrics
        if self.prometheus_enabled:
            if 'layer5_counter' in self.prometheus_metrics:
                self.prometheus_metrics['layer5_counter'].inc()

            if 'layer5_impact' in self.prometheus_metrics:
                impacts = self.layer_metrics['layer5_impacts']
```

```
                avg_impact = sum(impacts) / len(impacts) if impacts else 0.0
                self.prometheus_metrics['layer5_impact'].set(avg_impact)

            logger.info(f"Counter-evidence recorded: {impact_percent:.1f}% impact")

        except Exception as e:
            logger.error(f"Failed to record counter-evidence: {e}")

    def record_report_generated(self) -> None:
        """Record Layer 6 report generation."""
        try:
            self.layer_metrics['layer6_reports_generated'] += 1
            logger.info("Forensic report generated")
        except Exception as e:
            logger.error(f"Failed to record report generation: {e}")

    # ==================== SECURITY METRICS ====================

    def record_hash_chain_failure(self) -> None:
        """Record hash chain validation failure."""
        try:
            self.security_metrics['hash_chain_failures'] += 1
            logger.error("Hash chain validation failure")
        except Exception as e:
            logger.error(f"Failed to record hash chain failure: {e}")

    def record_validation_failure(self, validation_type: str) -> None:
        """
        Record input validation failure.

        Args:
            validation_type: Type of validation that failed
        """
        try:
            self.security_metrics['validation_failures'][validation_type] += 1
            logger.warning(f"Validation failure: {validation_type}")
        except Exception as e:
            logger.error(f"Failed to record validation failure: {e}")

    def record_crypto_failure(self, operation: str) -> None:
        """
        Record cryptographic operation failure.

        Args:
            operation: Crypto operation that failed
        """
```

```
    try:
        self.security_metrics['crypto_failures'][operation] += 1
        logger.error(f"Cryptographic failure: {operation}")
    except Exception as e:
        logger.error(f"Failed to record crypto failure: {e}")

def record_rate_limit_block(self, reason: str) -> None:
    """
    Record rate limiter block.

    Args:
        reason: Reason for block
    """
    try:
        self.security_metrics['rate_limit_blocks'][reason] += 1
        logger.warning(f"Rate limit block: {reason}")
    except Exception as e:
        logger.error(f"Failed to record rate limit block: {e}")

# ==================== STORAGE METRICS ====================

def record_storage_corruption(self) -> None:
    """Record storage corruption detection."""
    try:
        self.storage_metrics['corruption_detected'] += 1
        logger.error("Storage corruption detected")
    except Exception as e:
        logger.error(f"Failed to record storage corruption: {e}")

def record_index_inconsistency(self) -> None:
    """Record storage index inconsistency."""
    try:
        self.storage_metrics['index_inconsistencies'] += 1
        logger.warning("Storage index inconsistency detected")
    except Exception as e:
        logger.error(f"Failed to record index inconsistency: {e}")

def record_backup_failure(self) -> None:
    """Record backup failure."""
    try:
        self.storage_metrics['backup_failures'] += 1
        logger.error("Backup failure")
    except Exception as e:
        logger.error(f"Failed to record backup failure: {e}")

# ==================== QUEUE METRICS ====================
```

```
def update_queue_metrics(
    self,
    pending: int,
    processing: int
) -> None:
    """
    Update queue metrics.

    Args:
        pending: Number of pending jobs
        processing: Number of processing jobs
    """
    try:
        self.queue_metrics['pending_jobs'] = pending
        self.queue_metrics['processing_jobs'] = processing

        # Update Prometheus gauges
        if self.prometheus_enabled:
            if 'queue_pending' in self.prometheus_metrics:
                self.prometheus_metrics['queue_pending'].set(pending)
            if 'queue_processing' in self.prometheus_metrics:
                self.prometheus_metrics['queue_processing'].set(processing)

    except Exception as e:
        logger.error(f"Failed to update queue metrics: {e}")

def record_queue_failure(self, job_type: str) -> None:
    """
    Record queue job failure.

    Args:
        job_type: Type of job that failed
    """
    try:
        self.queue_metrics['job_failures'][job_type] += 1
        logger.error(f"Queue job failure: {job_type}")
    except Exception as e:
        logger.error(f"Failed to record queue failure: {e}")

# ==================== EVALUATION METRICS ====================

def update_evaluation_scores(
    self,
    auroc: Optional[float] = None,
    precision: Optional[float] = None,
```

```
    recall: Optional[float] = None
) -> None:
    """
    Update research evaluation scores.

    Args:
        auroc: AUROC score
        precision: Precision score
        recall: Recall score
    """
    try:
        if auroc is not None:
            self.evaluation_metrics['auroc_score'] = auroc
        if precision is not None:
            self.evaluation_metrics['precision_score'] = precision
        if recall is not None:
            self.evaluation_metrics['recall_score'] = recall

        logger.info(
            f"Evaluation scores updated: "
            f"AUROC={auroc}, Precision={precision}, Recall={recall}"
        )

    except Exception as e:
        logger.error(f"Failed to update evaluation scores: {e}")

# ==================== METRICS RETRIEVAL ====================

def get_request_stats(self) -> Dict:
    """
    Get request statistics.

    Returns:
        dict: Request metrics
    """
    if not self.request_latencies:
        return {
            'total_requests': 0,
            'avg_latency': 0.0,
            'p50_latency': 0.0,
            'p95_latency': 0.0,
            'p99_latency': 0.0
        }

    # Calculate percentiles
    sorted_latencies = sorted(self.request_latencies)
```

```
        n = len(sorted_latencies)

        p50_idx = int(n * 0.50)
        p95_idx = int(n * 0.95)
        p99_idx = int(n * 0.99)

        stats = {
            'total_requests': self.total_requests,
            'total_errors': self.total_errors,
            'error_rate': round(self.total_errors / self.total_requests * 100, 2) if
self.total_requests > 0 else 0.0,
            'recent_requests': len(self.request_latencies),
            'avg_latency': round(sum(self.request_latencies) / n, 3),
            'min_latency': round(min(self.request_latencies), 3),
            'max_latency': round(max(self.request_latencies), 3),
            'p50_latency': round(sorted_latencies[p50_idx], 3),
            'p95_latency': round(sorted_latencies[p95_idx], 3),
            'p99_latency': round(sorted_latencies[p99_idx], 3)
        }

        return stats

    def get_throughput(self, window_seconds: int = 60) -> float:
        """
        Calculate request throughput (requests per second).

        Args:
            window_seconds: Time window in seconds

        Returns:
            float: Requests per second
        """
        if not self.request_timestamps:
            return 0.0

        current_time = time.time()
        cutoff_time = current_time - window_seconds

        # Count requests in window
        recent_requests = sum(
            1 for ts in self.request_timestamps
            if ts > cutoff_time
        )

        # Calculate throughput
        throughput = recent_requests / window_seconds
```

```
        return round(throughput, 2)

    def get_model_stats(self) -> Dict:
        """
        Get model inference statistics.

        Returns:
            dict: Model metrics by model name
        """
        model_stats = {}

        for model_name, times in self.model_inference_times.items():
            if times:
                model_stats[model_name] = {
                    'total_inferences': len(times),
                    'avg_time': round(sum(times) / len(times), 3),
                    'min_time': round(min(times), 3),
                    'max_time': round(max(times), 3)
                }

        return model_stats

    def get_endpoint_stats(self) -> Dict:
        """
        Get endpoint usage statistics.

        Returns:
            dict: Endpoint metrics
        """
        endpoint_stats = {}

        for endpoint, count in self.endpoint_counts.items():
            errors = self.endpoint_errors.get(endpoint, 0)
            error_rate = (errors / count * 100) if count > 0 else 0.0

            endpoint_stats[endpoint] = {
                'requests': count,
                'errors': errors,
                'error_rate': round(error_rate, 2)
            }

        return endpoint_stats

    def get_layer_stats(self) -> Dict:
        """
```

```
        Get layer-specific statistics.

        Returns:
            dict: Layer metrics
        """
        stats = {
            'layer1': {
                'anonymity_violations': self.layer_metrics['layer1_anonymity_violations']
            },
            'layer2': {
                'deepfakes_detected': self.layer_metrics['layer2_deepfakes_detected'],
                'avg_credibility_score': round(
                    sum(self.layer_metrics['layer2_credibility_scores']) /
                    len(self.layer_metrics['layer2_credibility_scores'])
                    if self.layer_metrics['layer2_credibility_scores'] else 0.0,
                    3
                )
            },
            'layer3': {
                'coordination_detected': self.layer_metrics['layer3_coordination_detected']
            },
            'layer4': {
                'avg_iterations': round(
                    sum(self.layer_metrics['layer4_consensus_iterations']) /
                    len(self.layer_metrics['layer4_consensus_iterations'])
                    if self.layer_metrics['layer4_consensus_iterations'] else 0.0,
                    2
                ),
                'avg_convergence_time': round(
                    sum(self.layer_metrics['layer4_convergence_times']) /
                    len(self.layer_metrics['layer4_convergence_times'])
                    if self.layer_metrics['layer4_convergence_times'] else 0.0,
                    3
                )
            },
            'layer5': {
                'counter_evidence_count':
self.layer_metrics['layer5_counter_evidence_count'],
                'avg_impact_percent': round(
                    sum(self.layer_metrics['layer5_impacts']) /
                    len(self.layer_metrics['layer5_impacts'])
                    if self.layer_metrics['layer5_impacts'] else 0.0,
                    2
                )
            },
            'layer6': {
```

```
            'reports_generated': self.layer_metrics['layer6_reports_generated']
        }
    }

    return stats

def get_security_stats(self) -> Dict:
    """Get security-related statistics."""
    return {
        'hash_chain_failures': self.security_metrics['hash_chain_failures'],
        'validation_failures': dict(self.security_metrics['validation_failures']),
        'crypto_failures': dict(self.security_metrics['crypto_failures']),
        'rate_limit_blocks': dict(self.security_metrics['rate_limit_blocks'])
    }

def get_storage_stats(self) -> Dict:
    """Get storage-related statistics."""
    return dict(self.storage_metrics)

def get_queue_stats(self) -> Dict:
    """Get queue-related statistics."""
    return {
        'pending_jobs': self.queue_metrics['pending_jobs'],
        'processing_jobs': self.queue_metrics['processing_jobs'],
        'job_failures': dict(self.queue_metrics['job_failures'])
    }

def get_evaluation_stats(self) -> Dict:
    """Get research evaluation statistics."""
    return dict(self.evaluation_metrics)

def get_system_metrics(self) -> Dict:
    """
    Get system resource metrics.

    Returns:
        dict: System metrics
    """
    try:
        # Memory info
        memory_info = self.process.memory_info()
        memory_percent = self.process.memory_percent()

        # CPU info
        cpu_percent = self.process.cpu_percent(interval=0.1)
```

```
            # Uptime
            uptime_seconds = time.time() - self.start_time

            metrics = {
                'memory_rss_mb': round(memory_info.rss / 1024 / 1024, 2),
                'memory_vms_mb': round(memory_info.vms / 1024 / 1024, 2),
                'memory_percent': round(memory_percent, 2),
                'cpu_percent': round(cpu_percent, 2),
                'num_threads': self.process.num_threads(),
                'uptime_seconds': int(uptime_seconds),
                'uptime_hours': round(uptime_seconds / 3600, 2)
            }

            return metrics

        except Exception as e:
            logger.error(f"Failed to get system metrics: {e}")
            return {}

    def get_summary(self) -> Dict:
        """
        Get summary of all metrics.

        Returns:
            dict: Summary metrics for logging/monitoring
        """
        request_stats = self.get_request_stats()
        system_metrics = self.get_system_metrics()
        layer_stats = self.get_layer_stats()

        summary = {
            'requests': request_stats.get('total_requests', 0),
            'error_rate': request_stats.get('error_rate', 0),
            'avg_latency': request_stats.get('avg_latency', 0),
            'memory_percent': system_metrics.get('memory_percent', 0),
            'cpu_percent': system_metrics.get('cpu_percent', 0),
            'uptime_hours': system_metrics.get('uptime_hours', 0),
            'submissions': self.submission_stats.get('total', 0),
            'deepfakes_detected': layer_stats['layer2']['deepfakes_detected'],
            'coordination_detected': layer_stats['layer3']['coordination_detected'],
            'reports_generated': layer_stats['layer6']['reports_generated']
        }

        return summary

    def get_all_metrics(self) -> Dict:
```

```
        """
        Get all metrics combined.

        Returns:
            dict: All metrics
        """
        metrics = {
            'timestamp': datetime.utcnow().isoformat(),
            'requests': self.get_request_stats(),
            'throughput_1min': self.get_throughput(60),
            'throughput_5min': self.get_throughput(300),
            'models': self.get_model_stats(),
            'endpoints': self.get_endpoint_stats(),
            'submissions': self.submission_stats.copy(),
            'layers': self.get_layer_stats(),
            'security': self.get_security_stats(),
            'storage': self.get_storage_stats(),
            'queue': self.get_queue_stats(),
            'evaluation': self.get_evaluation_stats(),
            'system': self.get_system_metrics(),
            'generic': self.get_generic_metrics(),
            'prometheus_enabled': self.prometheus_enabled
        }

        return metrics

    def _persist_metrics(self) -> None:
        """Persist metrics to file."""
        if not self.metrics_file:
            return

        try:
            metrics = self.get_all_metrics()

            # Ensure parent directory exists
            self.metrics_file.parent.mkdir(parents=True, exist_ok=True)

            # Append to file (JSON Lines format)
            import json
            with open(self.metrics_file, 'a') as f:
                f.write(json.dumps(metrics) + '\n')

            self.last_persist_time = time.time()

            logger.debug(f"Metrics persisted to {self.metrics_file}")
```

```
        except Exception as e:
            logger.error(f"Failed to persist metrics: {e}")

    def reset_metrics(self) -> None:
        """Reset all metrics."""
        self.request_latencies.clear()
        self.request_timestamps.clear()
        self.model_inference_times.clear()
        self.endpoint_counts.clear()
        self.endpoint_errors.clear()
        self.submission_stats = {
            'total': 0,
            'pending': 0,
            'completed': 0,
            'failed': 0
        }
        self.total_requests = 0
        self.total_errors = 0

        # Reset layer metrics
        self.layer_metrics = {
            'layer1_anonymity_violations': 0,
            'layer2_credibility_scores': deque(maxlen=100),
            'layer2_deepfakes_detected': 0,
            'layer3_coordination_detected': 0,
            'layer4_consensus_iterations': deque(maxlen=100),
            'layer4_convergence_times': deque(maxlen=100),
            'layer5_counter_evidence_count': 0,
            'layer5_impacts': deque(maxlen=100),
            'layer6_reports_generated': 0
        }

        # Reset security metrics
        self.security_metrics = {
            'hash_chain_failures': 0,
            'validation_failures': defaultdict(int),
            'crypto_failures': defaultdict(int),
            'rate_limit_blocks': defaultdict(int)
        }

        # Reset storage metrics
        self.storage_metrics = {
            'corruption_detected': 0,
            'index_inconsistencies': 0,
            'backup_failures': 0
        }
```

```
    # Reset queue metrics
    self.queue_metrics = {
        'pending_jobs': 0,
        'processing_jobs': 0,
        'job_failures': defaultdict(int)
    }

    logger.info("Metrics reset")

def get_health_status(self) -> Dict:
    """
    Get system health status.

    Returns:
        dict: Health status
    """
    system_metrics = self.get_system_metrics()
    request_stats = self.get_request_stats()

    # Determine health status
    status = "healthy"
    issues = []

    # Check memory usage
    if system_metrics.get('memory_percent', 0) > 90:
        status = "unhealthy"
        issues.append("High memory usage")
    elif system_metrics.get('memory_percent', 0) > 75:
        status = "degraded"
        issues.append("Elevated memory usage")

    # Check error rate
    error_rate = request_stats.get('error_rate', 0)
    if error_rate > 10:
        status = "unhealthy"
        issues.append("High error rate")
    elif error_rate > 5:
        if status == "healthy":
            status = "degraded"
        issues.append("Elevated error rate")

    health = {
        'status': status,
        'timestamp': datetime.utcnow().isoformat(),
        'uptime_hours': system_metrics.get('uptime_hours', 0),
```

```
            'issues': issues,
            'metrics': {
                'memory_percent': system_metrics.get('memory_percent', 0),
                'cpu_percent': system_metrics.get('cpu_percent', 0),
                'error_rate': error_rate,
                'avg_latency': request_stats.get('avg_latency', 0)
            }
        }

        return health

    def record_gauge(self, metric_name: str, value: float) -> None:
        """
        Record a generic gauge metric.

        Args:
            metric_name: Name of the metric
            value: Metric value
        """
        try:
            # Store in a generic metrics dict (add to __init__ if needed)
            if not hasattr(self, 'generic_gauges'):
                self.generic_gauges = {}

            self.generic_gauges[metric_name] = value

            logger.debug(f"Gauge metric recorded: {metric_name} = {value}")

        except Exception as e:
            logger.error(f"Failed to record gauge metric {metric_name}: {e}")

    def record(self, metric_name: str, value: float) -> None:
        """
        Record a generic metric (alias for record_gauge).

        Args:
            metric_name: Name of the metric
            value: Metric value
        """
        self.record_gauge(metric_name, value)

    def get_generic_metrics(self) -> Dict[str, float]:
        """
        Get all generic metrics.

        Returns:
```

```
            dict: Generic metrics
        """
        return getattr(self, 'generic_gauges', {})

"""
Queue Service - Background job processing

Provides:
- In-memory job queue (no Redis/Celery)
- Async task processing
- Priority queue support
- Job status tracking
- Retry logic with exponential backoff
- Worker pool management
"""

import asyncio
import logging
import time
import uuid
from collections import defaultdict
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from typing import Any, Callable, Dict, List, Optional

# Initialize logger
logger = logging.getLogger(__name__)


class JobStatus(str, Enum):
    """Job status enumeration."""
    PENDING = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    FAILED = "failed"
    RETRYING = "retrying"
    CANCELLED = "cancelled"


class JobPriority(int, Enum):
    """Job priority levels."""
    LOW = 0
    NORMAL = 1
    HIGH = 2
    CRITICAL = 3
```

```
@dataclass
class Job:
    """
    Job data structure.
    """
    job_id: str
    task_name: str
    func: Callable
    args: tuple = field(default_factory=tuple)
    kwargs: dict = field(default_factory=dict)
    priority: JobPriority = JobPriority.NORMAL
    status: JobStatus = JobStatus.PENDING
    created_at: float = field(default_factory=time.time)
    started_at: Optional[float] = None
    completed_at: Optional[float] = None
    result: Any = None
    error: Optional[str] = None
    retry_count: int = 0
    max_retries: int = 3
    timeout: Optional[int] = None  # seconds

    def __lt__(self, other):
        """Compare jobs by priority (for priority queue)."""
        return self.priority.value > other.priority.value  # Higher priority first


class QueueService:
    """
    Queue Service - Background job processing.

    Features:
    - In-memory priority queue
    - Async job processing
    - Retry logic with exponential backoff
    - Job status tracking
    - Worker pool management
    - Job history
    """

    def __init__(
        self,
        max_workers: int = 4,
        max_queue_size: int = 1000,
        job_timeout: int = 300
```

```
    ):
        """
        Initialize queue service.

        Args:
            max_workers: Maximum concurrent workers
            max_queue_size: Maximum queue size
            job_timeout: Default job timeout in seconds
        """
        self.max_workers = max_workers
        self.max_queue_size = max_queue_size
        self.default_timeout = job_timeout

        # Job storage
        self.jobs: Dict[str, Job] = {}
        self.pending_queue: asyncio.PriorityQueue =
asyncio.PriorityQueue(maxsize=max_queue_size)

        # Worker management
        self.workers: List[asyncio.Task] = []
        self.running = False
        self.queues = defaultdict(list)

        # Statistics
        self.stats = {
            'jobs_submitted': 0,
            'jobs_completed': 0,
            'jobs_failed': 0,
            'jobs_retried': 0,
            'total_processing_time': 0.0
        }

        logger.info(
            f"QueueService initialized "
            f"(workers={max_workers}, queue_size={max_queue_size})"
        )

    async def start(self) -> None:
        """Start worker pool."""
        if self.running:
            logger.warning("Queue service already running")
            return

        self.running = True

        # Start worker tasks
```

```
    for i in range(self.max_workers):
        worker = asyncio.create_task(self._worker(worker_id=i))
        self.workers.append(worker)

    logger.info(f"Started {self.max_workers} workers")

async def stop(self) -> None:
    """Stop worker pool gracefully."""
    if not self.running:
        return

    self.running = False

    # Wait for workers to finish
    if self.workers:
        await asyncio.gather(*self.workers, return_exceptions=True)

    self.workers.clear()
    logger.info("Queue service stopped")

async def submit_job(
    self,
    task_name: str,
    func: Callable,
    *args,
    priority: JobPriority = JobPriority.NORMAL,
    max_retries: int = 3,
    timeout: Optional[int] = None,
    **kwargs
) -> str:
    """
    Submit job to queue.

    Args:
        task_name: Human-readable task name
        func: Function to execute
        *args: Positional arguments
        priority: Job priority
        max_retries: Maximum retry attempts
        timeout: Job timeout in seconds
        **kwargs: Keyword arguments

    Returns:
        str: Job ID

    Raises:
```

```
            ValueError: If queue is full
        """
        try:
            # Check queue size
            if self.pending_queue.qsize() >= self.max_queue_size:
                raise ValueError("Queue is full")

            # Create job
            job_id = str(uuid.uuid4())

            job = Job(
                job_id=job_id,
                task_name=task_name,
                func=func,
                args=args,
                kwargs=kwargs,
                priority=priority,
                max_retries=max_retries,
                timeout=timeout or self.default_timeout
            )

            # Store job
            self.jobs[job_id] = job

            # Add to queue
            await self.pending_queue.put(job)

            self.stats['jobs_submitted'] += 1

            logger.info(
                f"Job submitted: {job_id} ({task_name}) "
                f"[priority={priority.name}]"
            )

            return job_id

        except Exception as e:
            logger.error(f"Failed to submit job: {e}")
            raise

    def enqueue(self, queue_name: str, item: str) -> None:
        """Add item to queue."""
        self.queues[queue_name].append(item)

    def dequeue(self, queue_name: str) -> Optional[str]:
        """Remove and return item from queue."""
```

```
    if self.queues[queue_name]:
        return self.queues[queue_name].pop(0)
    return None

def peek(self, queue_name: str) -> Optional[str]:
    """View next item without removing."""
    if self.queues[queue_name]:
        return self.queues[queue_name][0]
    return None

def size(self, queue_name: str) -> int:
    """Get queue size."""
    return len(self.queues[queue_name])

async def _worker(self, worker_id: int) -> None:
    """
    Worker coroutine that processes jobs from queue.

    Args:
        worker_id: Worker identifier
    """
    logger.info(f"Worker {worker_id} started")

    while self.running:
        try:
            # Get job from queue (with timeout)
            try:
                job = await asyncio.wait_for(
                    self.pending_queue.get(),
                    timeout=1.0
                )
            except asyncio.TimeoutError:
                continue

            # Process job
            await self._process_job(job, worker_id)

        except Exception as e:
            logger.error(f"Worker {worker_id} error: {e}")

    logger.info(f"Worker {worker_id} stopped")

async def _process_job(self, job: Job, worker_id: int) -> None:
    """
    Process a single job.
```

```
    Args:
        job: Job to process
        worker_id: Worker identifier
    """
    try:
        # Update job status
        job.status = JobStatus.RUNNING
        job.started_at = time.time()

        logger.info(
            f"Worker {worker_id} processing job {job.job_id} "
            f"({job.task_name})"
        )

        # Execute job with timeout
        try:
            if asyncio.iscoroutinefunction(job.func):
                # Async function
                result = await asyncio.wait_for(
                    job.func(*job.args, **job.kwargs),
                    timeout=job.timeout
                )
            else:
                # Sync function - run in executor
                loop = asyncio.get_event_loop()
                result = await asyncio.wait_for(
                    loop.run_in_executor(
                        None,
                        lambda: job.func(*job.args, **job.kwargs)
                    ),
                    timeout=job.timeout
                )

            # Job completed successfully
            job.status = JobStatus.COMPLETED
            job.result = result
            job.completed_at = time.time()

            # Update stats
            processing_time = job.completed_at - job.started_at
            self.stats['jobs_completed'] += 1
            self.stats['total_processing_time'] += processing_time

            logger.info(
                f"Job completed: {job.job_id} "
                f"(time={processing_time:.2f}s)"
```

```
                )

            except asyncio.TimeoutError:
                raise TimeoutError(f"Job exceeded timeout of {job.timeout}s")

        except Exception as e:
            # Job failed
            error_msg = str(e)
            logger.error(f"Job failed: {job.job_id} - {error_msg}")

            job.error = error_msg

            # Retry logic
            if job.retry_count < job.max_retries:
                job.retry_count += 1
                job.status = JobStatus.RETRYING

                # Calculate backoff delay (exponential)
                backoff_delay = min(2 ** job.retry_count, 60)  # Max 60s

                logger.info(
                    f"Retrying job {job.job_id} "
                    f"(attempt {job.retry_count}/{job.max_retries}) "
                    f"in {backoff_delay}s"
                )

                # Schedule retry
                await asyncio.sleep(backoff_delay)
                await self.pending_queue.put(job)

                self.stats['jobs_retried'] += 1
            else:
                # Max retries exceeded
                job.status = JobStatus.FAILED
                job.completed_at = time.time()

                self.stats['jobs_failed'] += 1

                logger.error(
                    f"Job failed permanently: {job.job_id} "
                    f"(retries={job.retry_count})"
                )

    def get_job_status(self, job_id: str) -> Optional[Dict]:
        """
        Get job status.
```

```
        Args:
            job_id: Job identifier

        Returns:
            dict: Job status information or None
        """
        job = self.jobs.get(job_id)

        if not job:
            return None

        status_info = {
            'job_id': job.job_id,
            'task_name': job.task_name,
            'status': job.status.value,
            'priority': job.priority.name,
            'created_at': datetime.fromtimestamp(job.created_at).isoformat(),
            'retry_count': job.retry_count,
            'max_retries': job.max_retries
        }

        if job.started_at:
            status_info['started_at'] = datetime.fromtimestamp(job.started_at).isoformat()

        if job.completed_at:
            status_info['completed_at'] =
    datetime.fromtimestamp(job.completed_at).isoformat()
            status_info['duration'] = job.completed_at - job.started_at

        if job.status == JobStatus.COMPLETED:
            status_info['result'] = job.result
        elif job.status == JobStatus.FAILED:
            status_info['error'] = job.error

        return status_info

    def get_job_result(self, job_id: str) -> Optional[Any]:
        """
        Get job result (if completed).

        Args:
            job_id: Job identifier

        Returns:
            Job result or None
```

```
    """
    job = self.jobs.get(job_id)

    if job and job.status == JobStatus.COMPLETED:
        return job.result

    return None

async def cancel_job(self, job_id: str) -> bool:
    """
    Cancel pending job.

    Args:
        job_id: Job identifier

    Returns:
        bool: True if cancelled, False if not found or already running
    """
    job = self.jobs.get(job_id)

    if not job:
        return False

    if job.status == JobStatus.PENDING:
        job.status = JobStatus.CANCELLED
        logger.info(f"Job cancelled: {job_id}")
        return True

    return False

def get_queue_stats(self) -> Dict:
    """
    Get queue statistics.

    Returns:
        dict: Queue statistics
    """
    # Count jobs by status
    status_counts = defaultdict(int)
    for job in self.jobs.values():
        status_counts[job.status.value] += 1

    # Calculate average processing time
    avg_processing_time = 0.0
    if self.stats['jobs_completed'] > 0:
        avg_processing_time = (
```

```
                self.stats['total_processing_time'] /
                self.stats['jobs_completed']
            )

        stats = {
            'running': self.running,
            'workers': len(self.workers),
            'queue_size': self.pending_queue.qsize(),
            'max_queue_size': self.max_queue_size,
            'total_jobs': len(self.jobs),
            'jobs_by_status': dict(status_counts),
            'jobs_submitted': self.stats['jobs_submitted'],
            'jobs_completed': self.stats['jobs_completed'],
            'jobs_failed': self.stats['jobs_failed'],
            'jobs_retried': self.stats['jobs_retried'],
            'avg_processing_time': round(avg_processing_time, 2),
            'timestamp': datetime.utcnow().isoformat()
        }

        return stats

    def cleanup_old_jobs(self, max_age_seconds: int = 86400) -> int:
        """
        Remove old completed/failed jobs.

        Args:
            max_age_seconds: Maximum age in seconds (default: 24 hours)

        Returns:
            int: Number of jobs removed
        """
        current_time = time.time()
        cutoff_time = current_time - max_age_seconds

        jobs_to_remove = []

        for job_id, job in self.jobs.items():
            # Remove if completed/failed and old
            if job.status in (JobStatus.COMPLETED, JobStatus.FAILED,
JobStatus.CANCELLED):
                if job.completed_at and job.completed_at < cutoff_time:
                    jobs_to_remove.append(job_id)

        for job_id in jobs_to_remove:
            del self.jobs[job_id]
```

```
        if jobs_to_remove:
            logger.info(f"Cleaned up {len(jobs_to_remove)} old jobs")

        return len(jobs_to_remove)

"""
Rate Limiter Service - IP-based rate limiting

Provides:
- IP-based request rate limiting
- Sliding window algorithm
- Configurable limits per endpoint
- Temporary IP blocking
- Rate limit status tracking
"""

import logging
import time
from collections import defaultdict
from datetime import datetime, timedelta
from typing import Dict, Optional, Tuple

# Initialize logger
logger = logging.getLogger(__name__)


class RateLimiter:
    """
    Rate Limiter - IP-based request rate limiting.

    Features:
    - Sliding window rate limiting
    - Per-IP tracking
    - Configurable limits (requests per time window)
    - Automatic cleanup of old records
    - Temporary IP blocking for abuse
    - Different limits for different endpoint types
    """

    def __init__(
        self,
        default_limit: int = 10,
        window_seconds: int = 3600,
        cleanup_interval: int = 300
    ):
        """
```

```
    Initialize rate limiter.

    Args:
        default_limit: Default number of requests per window
        window_seconds: Time window in seconds (default: 1 hour)
        cleanup_interval: Cleanup old records every N seconds
    """
    self.default_limit = default_limit
    self.window_seconds = window_seconds
    self.cleanup_interval = cleanup_interval

    # IP tracking: {ip_address: [timestamps]}
    self.requests: Dict[str, list] = defaultdict(list)

    # Blocked IPs: {ip_address: unblock_time}
    self.blocked_ips: Dict[str, float] = {}

    # Last cleanup time
    self.last_cleanup = time.time()

    # Custom limits for specific endpoint types
    self.endpoint_limits = {
        'submission': 1000,   # Increased for evaluation
        'status': 3600,       # 1 per second
        'report': 100,        # Increased
        'health': 3600,       # 1 per second
        'counter_evidence': 100 # Increased
    }

    logger.info(
        f"RateLimiter initialized "
        f"(default={default_limit} req/{window_seconds}s)"
    )

def check_rate_limit(
    self,
    ip_address: str,
    endpoint_type: str = 'default'
) -> Tuple[bool, Dict]:
    """
    Check if request is within rate limit.

    Args:
        ip_address: Client IP address
        endpoint_type: Type of endpoint (submission/status/report/etc)
```

```
Returns:
    tuple: (is_allowed, rate_limit_info)
"""
try:
    current_time = time.time()

    # Cleanup old records periodically
    if current_time - self.last_cleanup > self.cleanup_interval:
        self._cleanup_old_records()

    # Check if IP is blocked
    if ip_address in self.blocked_ips:
        unblock_time = self.blocked_ips[ip_address]

        if current_time < unblock_time:
            remaining = int(unblock_time - current_time)
            logger.warning(
                f"Blocked IP attempted request: {ip_address} "
                f"(blocked for {remaining}s)"
            )

            return False, {
                'allowed': False,
                'reason': 'IP temporarily blocked',
                'retry_after': remaining,
                'blocked_until': datetime.fromtimestamp(unblock_time).isoformat()
            }
        else:
            # Unblock IP
            del self.blocked_ips[ip_address]
            logger.info(f"IP unblocked: {ip_address}")

    # Get limit for endpoint type
    limit = self.endpoint_limits.get(endpoint_type, self.default_limit)

    # Get request history for IP
    request_times = self.requests[ip_address]

    # Remove requests outside time window
    cutoff_time = current_time - self.window_seconds
    request_times = [t for t in request_times if t > cutoff_time]
    self.requests[ip_address] = request_times

    # Check if under limit
    current_count = len(request_times)
```

```
if current_count >= limit:
    # Rate limit exceeded
    logger.warning(
        f"Rate limit exceeded: {ip_address} "
        f"({current_count}/{limit} for {endpoint_type})"
    )

    # Check if should block IP (excessive violations)
    if current_count >= limit * 2:
        self._block_ip(ip_address, duration=3600)  # Block for 1 hour

    # Calculate when next request is allowed
    oldest_request = min(request_times) if request_times else current_time
    retry_after = int(self.window_seconds - (current_time - oldest_request))

    return False, {
        'allowed': False,
        'reason': 'Rate limit exceeded',
        'limit': limit,
        'current': current_count,
        'window_seconds': self.window_seconds,
        'retry_after': retry_after,
        'endpoint_type': endpoint_type
    }

# Add current request
request_times.append(current_time)

# Calculate remaining requests
remaining = limit - (current_count + 1)
reset_time = current_time + self.window_seconds

logger.debug(
    f"Rate limit check passed: {ip_address} "
    f"({current_count + 1}/{limit} for {endpoint_type})"
)

return True, {
    'allowed': True,
    'limit': limit,
    'remaining': remaining,
    'reset': int(reset_time),
    'reset_datetime': datetime.fromtimestamp(reset_time).isoformat(),
    'window_seconds': self.window_seconds,
    'endpoint_type': endpoint_type
}
```

```
        except Exception as e:
            logger.error(f"Rate limit check failed: {e}")
            # Fail open (allow request on error)
            return True, {
                'allowed': True,
                'error': str(e)
            }

    def _block_ip(self, ip_address: str, duration: int = 3600) -> None:
        """
        Temporarily block IP address.

        Args:
            ip_address: IP to block
            duration: Block duration in seconds
        """
        unblock_time = time.time() + duration
        self.blocked_ips[ip_address] = unblock_time

        logger.warning(
            f"IP blocked: {ip_address} "
            f"(duration={duration}s,
until={datetime.fromtimestamp(unblock_time).isoformat()})"
        )

    def unblock_ip(self, ip_address: str) -> bool:
        """
        Manually unblock IP address.

        Args:
            ip_address: IP to unblock

        Returns:
            bool: True if IP was blocked, False if not
        """
        if ip_address in self.blocked_ips:
            del self.blocked_ips[ip_address]
            logger.info(f"IP manually unblocked: {ip_address}")
            return True
        return False

    def _cleanup_old_records(self) -> None:
        """Remove old request records to free memory."""
        try:
            current_time = time.time()
```

```
            cutoff_time = current_time - self.window_seconds

            # Cleanup request history
            cleaned_ips = []
            for ip, timestamps in list(self.requests.items()):
                # Remove old timestamps
                timestamps[:] = [t for t in timestamps if t > cutoff_time]

                # Remove IP if no recent requests
                if not timestamps:
                    cleaned_ips.append(ip)

            for ip in cleaned_ips:
                del self.requests[ip]

            # Cleanup expired blocks
            expired_blocks = [
                ip for ip, unblock_time in self.blocked_ips.items()
                if current_time >= unblock_time
            ]

            for ip in expired_blocks:
                del self.blocked_ips[ip]

            self.last_cleanup = current_time

            if cleaned_ips or expired_blocks:
                logger.debug(
                    f"Cleanup: removed {len(cleaned_ips)} IPs, "
                    f"unblocked {len(expired_blocks)} IPs"
                )

        except Exception as e:
            logger.error(f"Cleanup failed: {e}")

    def get_rate_limit_status(self, ip_address: str) -> Dict:
        """
        Get current rate limit status for IP.

        Args:
            ip_address: IP address

        Returns:
            dict: Rate limit status
        """
        current_time = time.time()
```

```
    cutoff_time = current_time - self.window_seconds

    # Get recent requests
    request_times = [t for t in self.requests.get(ip_address, []) if t > cutoff_time]

    # Check if blocked
    is_blocked = ip_address in self.blocked_ips
    unblock_time = self.blocked_ips.get(ip_address, 0)

    status = {
        'ip_address': ip_address,
        'is_blocked': is_blocked,
        'recent_requests': len(request_times),
        'limits': self.endpoint_limits,
        'window_seconds': self.window_seconds
    }

    if is_blocked:
        remaining = int(unblock_time - current_time)
        status['blocked_remaining'] = remaining
        status['unblock_time'] = datetime.fromtimestamp(unblock_time).isoformat()

    return status

def reset_ip(self, ip_address: str) -> None:
    """
    Reset rate limit counters for IP.

    Args:
        ip_address: IP address
    """
    if ip_address in self.requests:
        del self.requests[ip_address]

    if ip_address in self.blocked_ips:
        del self.blocked_ips[ip_address]

    logger.info(f"Rate limit reset for IP: {ip_address}")

def get_statistics(self) -> Dict:
    """
    Get rate limiter statistics.

    Returns:
        dict: Statistics
    """
```

```
        current_time = time.time()
        cutoff_time = current_time - self.window_seconds

        # Count active IPs (with recent requests)
        active_ips = sum(
            1 for timestamps in self.requests.values()
            if any(t > cutoff_time for t in timestamps)
        )

        # Count total recent requests
        total_requests = sum(
            len([t for t in timestamps if t > cutoff_time])
            for timestamps in self.requests.values()
        )

        stats = {
            'active_ips': active_ips,
            'blocked_ips': len(self.blocked_ips),
            'total_tracked_ips': len(self.requests),
            'recent_requests': total_requests,
            'window_seconds': self.window_seconds,
            'default_limit': self.default_limit,
            'endpoint_limits': self.endpoint_limits,
            'timestamp': datetime.utcnow().isoformat()
        }

        return stats

    def update_limit(self, endpoint_type: str, new_limit: int) -> None:
        """
        Update rate limit for endpoint type.

        Args:
            endpoint_type: Endpoint type
            new_limit: New limit value
        """
        old_limit = self.endpoint_limits.get(endpoint_type, self.default_limit)
        self.endpoint_limits[endpoint_type] = new_limit

        logger.info(
            f"Rate limit updated for {endpoint_type}: "
            f"{old_limit} !' {new_limit}"
        )

"""
Storage Service - JSON-based submission storage
```

```
Provides:
- Atomic file operations with locking
- Submission CRUD operations
- Validator pool management
- Index management for fast lookup
- Directory sharding for scalability
- Archive functionality
"""

import json
import logging
import threading
import os
import time

from datetime import datetime
from pathlib import Path
from typing import Dict, List, Optional
from filelock import FileLock

# Initialize logger
logger = logging.getLogger(__name__)


class StorageService:
    """
    Storage Service for JSON-based data persistence.

    Features:
    - Thread-safe atomic writes with file locking
    - Submission storage with automatic indexing
    - Validator pool persistence
    - Archive functionality
    - Fast lookup via index file
    - Directory sharding by year/month
    """

    def __init__(self, data_dir: Optional[Path] = None):
        """
        Initialize storage service.

        Args:
            data_dir: Base data directory (defaults to backend/data)
        """
        if data_dir:
```

```
            self.data_dir = Path(data_dir)
        else:
            self.data_dir = Path(__file__).parent.parent / "data"

        # Create directory structure
        self.submissions_dir = self.data_dir / "submissions"
        self.evidence_dir = self.data_dir / "evidence"
        self.reports_dir = self.data_dir / "reports"
        self.cache_dir = self.data_dir / "cache"
        self.archive_dir = self.data_dir / "archive"

        # Initialize directories
        self._initialize_directories()

        # Index file
        self.index_file = self.data_dir / "index.json"
        self.index_lock = threading.RLock()

        # Validators file
        self.validators_file = self.data_dir / "validators.json"

        logger.info(f"StorageService initialized (data_dir={self.data_dir})")

    def _initialize_directories(self) -> None:
        """Create all required directories."""
        directories = [
            self.data_dir,
            self.submissions_dir,
            self.evidence_dir,
            self.reports_dir,
            self.cache_dir,
            self.archive_dir
        ]

        for directory in directories:
            directory.mkdir(parents=True, exist_ok=True)

        logger.debug(f"Storage directories initialized")

    def save_submission(self, submission_id: str, data: Dict) -> None:
        """
        Save submission data to JSON file with atomic write.

        Args:
            submission_id: Unique submission identifier
            data: Submission data dictionary
```

```
        Raises:
            IOError: If save fails
        """
        try:
            # Ensure submission_id is in data
            data['id'] = submission_id

            # Add/update timestamp
            if 'timestamp_updated' not in data:
                data['timestamp_updated'] = datetime.utcnow().isoformat()

            # File path
            file_path = self.submissions_dir / f"{submission_id}.json"
            lock_path = self.submissions_dir / f"{submission_id}.json.lock"

            # Atomic write with file lock
            lock = FileLock(str(lock_path), timeout=10)

            with lock:
                # Write to temporary file first
                temp_path = file_path.with_suffix('.tmp')

                with open(temp_path, 'w', encoding='utf-8') as f:
                    json.dump(data, f, indent=2, ensure_ascii=False)

                # Atomic rename with Windows support
                self._safe_replace(temp_path, file_path)

            # Update index
            self._update_index(submission_id, data)

            logger.debug(f"Saved submission {submission_id}")

        except Exception as e:
            logger.error(f"Failed to save submission {submission_id}: {e}")
            raise IOError(f"Failed to save submission: {str(e)}")

    def load_submission(self, submission_id: str) -> Optional[Dict]:
        """
        Load submission data from JSON file.

        Args:
            submission_id: Unique submission identifier

        Returns:
```

```
        dict: Submission data, or None if not found
    """
    try:
        file_path = self.submissions_dir / f"{submission_id}.json"

        if not file_path.exists():
            logger.debug(f"Submission {submission_id} not found")
            return None

        lock_path = self.submissions_dir / f"{submission_id}.json.lock"
        lock = FileLock(str(lock_path), timeout=10)

        with lock:
            with open(file_path, 'r', encoding='utf-8') as f:
                data = json.load(f)

        return data

    except Exception as e:
        logger.error(f"Failed to load submission {submission_id}: {e}")
        return None

def get_submission(self, submission_id: str) -> Optional[Dict]:
    """Alias for load_submission."""
    return self.load_submission(submission_id)

def update_submission(self, submission_id: str, updates: Dict) -> None:
    """
    Update submission with partial data.

    Args:
        submission_id: Submission identifier
        updates: Dictionary of updates to apply
    """
    try:
        lock_path = self.submissions_dir / f"{submission_id}.json.lock"
        lock = FileLock(str(lock_path), timeout=10)

        with lock:
            submission = self.load_submission(submission_id)
            if submission:
                submission.update(updates)
                self.save_submission(submission_id, submission)
    except Exception as e:
        logger.error(f"Failed to update submission {submission_id}: {e}")
```

```
def delete_submission(self, submission_id: str) -> bool:
    """
    Delete submission from storage.

    Args:
        submission_id: Unique submission identifier

    Returns:
        bool: True if deleted, False otherwise
    """
    try:
        file_path = self.submissions_dir / f"{submission_id}.json"

        if not file_path.exists():
            return False

        lock_path = self.submissions_dir / f"{submission_id}.json.lock"
        lock = FileLock(str(lock_path), timeout=10)

        with lock:
            file_path.unlink()

        # Remove lock file
        if lock_path.exists():
            lock_path.unlink()

        # Remove from index
        self._remove_from_index(submission_id)

        logger.info(f"Deleted submission {submission_id}")
        return True

    except Exception as e:
        logger.error(f"Failed to delete submission {submission_id}: {e}")
        return False

def get_all_submissions(self) -> List[Dict]:
    """
    Get all submissions from storage.

    Returns:
        list: List of all submission dictionaries
    """
    submissions = []

    try:
```

```
        # Use index for fast lookup
        index = self._load_index()

        for submission_id in index.keys():
            submission = self.load_submission(submission_id)
            if submission:
                submissions.append(submission)

        logger.debug(f"Loaded {len(submissions)} submissions")

    except Exception as e:
        logger.error(f"Failed to get all submissions: {e}")

    return submissions

def update_submission_status(self, submission_id: str, status: str) -> None:
    """
    Update submission status.

    Args:
        submission_id: Submission identifier
        status: New status value
    """
    submission = self.load_submission(submission_id)

    if submission:
        submission['status'] = status
        submission['status_updated'] = datetime.utcnow().isoformat()
        self.save_submission(submission_id, submission)

def archive_submission(self, submission_id: str) -> bool:
    """
    Archive old submission by moving to archive directory.

    Args:
        submission_id: Submission identifier

    Returns:
        bool: True if archived, False otherwise
    """
    try:
        source = self.submissions_dir / f"{submission_id}.json"

        if not source.exists():
            return False
```

```
        # Create archive directory if needed
        self.archive_dir.mkdir(parents=True, exist_ok=True)

        # Move to archive
        dest = self.archive_dir / f"{submission_id}.json"

        lock_path = self.submissions_dir / f"{submission_id}.json.lock"
        lock = FileLock(str(lock_path), timeout=10)

        with lock:
            source.rename(dest)

        # Clean up lock file
        if lock_path.exists():
            lock_path.unlink()

        # Remove from index
        self._remove_from_index(submission_id)

        logger.info(f"Archived submission {submission_id}")
        return True

    except Exception as e:
        logger.error(f"Failed to archive submission {submission_id}: {e}")
        return False

def save_validators(self, validators: List[Dict]) -> None:
    """
    Save validator pool to storage.

    Args:
        validators: List of validator dictionaries
    """
    try:
        lock_path = self.validators_file.with_suffix('.lock')
        lock = FileLock(str(lock_path), timeout=10)

        with lock:
            temp_path = self.validators_file.with_suffix('.tmp')

            with open(temp_path, 'w', encoding='utf-8') as f:
                json.dump(validators, f, indent=2, ensure_ascii=False)

            temp_path.replace(self.validators_file)

        logger.debug(f"Saved {len(validators)} validators")
```

```
    except Exception as e:
        logger.error(f"Failed to save validators: {e}")
        raise IOError(f"Failed to save validators: {str(e)}")

def load_validators(self) -> List[Dict]:
    """
    Load validator pool from storage.

    Returns:
        list: List of validator dictionaries
    """
    try:
        if not self.validators_file.exists():
            logger.debug("Validators file not found, returning empty list")
            return []

        lock_path = self.validators_file.with_suffix('.lock')
        lock = FileLock(str(lock_path), timeout=10)

        with lock:
            with open(self.validators_file, 'r', encoding='utf-8') as f:
                validators = json.load(f)

        logger.debug(f"Loaded {len(validators)} validators")
        return validators

    except Exception as e:
        logger.error(f"Failed to load validators: {e}")
        return []

def health_check(self) -> bool:
    """
    Check if storage is accessible and healthy.

    Returns:
        bool: True if healthy, False otherwise
    """
    try:
        # Check if directories exist
        directories_ok = (
            self.data_dir.exists() and
            self.submissions_dir.exists() and
            self.evidence_dir.exists()
        )
```

```
        if not directories_ok:
            logger.warning("Storage directories not accessible")
            return False

        # Test write/read
        test_file = self.data_dir / ".health_check"
        test_data = {"timestamp": datetime.utcnow().isoformat()}

        with open(test_file, 'w') as f:
            json.dump(test_data, f)

        with open(test_file, 'r') as f:
            json.load(f)

        test_file.unlink()

        logger.debug("Storage health check passed")
        return True

    except Exception as e:
        logger.error(f"Storage health check failed: {e}")
        return False

def _update_index(self, submission_id: str, data: Dict) -> None:
    """
    Update index file with submission metadata.

    Args:
        submission_id: Submission identifier
        data: Submission data
    """
    try:
        with self.index_lock:
            index = self._load_index()

            # Add/update entry
            index[submission_id] = {
                'pseudonym': data.get('pseudonym'),
                'evidence_type': data.get('evidence_type'),
                'status': data.get('status'),
                'timestamp_submission': data.get('timestamp_submission'),
                'timestamp_updated': data.get('timestamp_updated')
            }

            # Save index
            self._save_index(index)
```

```
    except Exception as e:
        logger.warning(f"Failed to update index: {e}")

def _remove_from_index(self, submission_id: str) -> None:
    """
    Remove submission from index.

    Args:
        submission_id: Submission identifier
    """
    try:
        with self.index_lock:
            index = self._load_index()

            if submission_id in index:
                del index[submission_id]
                self._save_index(index)

    except Exception as e:
        logger.warning(f"Failed to remove from index: {e}")

def _load_index(self) -> Dict:
    """
    Load index file.

    Returns:
        dict: Index data
    """
    try:
        with self.index_lock:
            if not self.index_file.exists():
                return {}

            with open(self.index_file, 'r', encoding='utf-8') as f:
                return json.load(f)

    except Exception as e:
        logger.warning(f"Failed to load index: {e}")
        return {}

def _save_index(self, index: Dict) -> None:
    """
    Save index file.

    Args:
```

```
        index: Index data
    """
    try:
        temp_path = self.index_file.with_suffix('.tmp')

        with open(temp_path, 'w', encoding='utf-8') as f:
            json.dump(index, f, indent=2, ensure_ascii=False)

        self._safe_replace(temp_path, self.index_file)

    except Exception as e:
        logger.error(f"Failed to save index: {e}")

def get_storage_statistics(self) -> Dict:
    """
    Get storage statistics.

    Returns:
        dict: Storage statistics
    """
    try:
        index = self._load_index()

        # Count by status
        status_counts = {}
        for entry in index.values():
            status = entry.get('status', 'unknown')
            status_counts[status] = status_counts.get(status, 0) + 1

        # Count by evidence type
        type_counts = {}
        for entry in index.values():
            evidence_type = entry.get('evidence_type', 'unknown')
            type_counts[evidence_type] = type_counts.get(evidence_type, 0) + 1

        # Calculate storage size
        total_size = 0
        for file_path in self.submissions_dir.glob("*.json"):
            total_size += file_path.stat().st_size

        stats = {
            'total_submissions': len(index),
            'status_distribution': status_counts,
            'type_distribution': type_counts,
            'storage_size_bytes': total_size,
            'storage_size_mb': round(total_size / (1024 * 1024), 2),
```

```
            'data_directory': str(self.data_dir),
            'timestamp': datetime.utcnow().isoformat()
        }

        return stats

    except Exception as e:
        logger.error(f"Failed to get storage statistics: {e}")
        return {
            'error': str(e),
            'timestamp': datetime.utcnow().isoformat()
        }

def get_evidence_path(
    self,
    submission_id: str,
    filename: str,
    create_dirs: bool = True
) -> Path:
    """
    Get path for evidence file with year/month sharding.

    Args:
        submission_id: Submission identifier
        filename: Evidence filename
        create_dirs: Create directories if they don't exist

    Returns:
        Path: Full path for evidence file
    """
    now = datetime.utcnow()
    year = now.strftime('%Y')
    month = now.strftime('%m')

    # Sharded directory structure
    evidence_path = self.evidence_dir / year / month

    if create_dirs:
        evidence_path.mkdir(parents=True, exist_ok=True)

    return evidence_path / filename

def cleanup_cache(self, max_age_hours: int = 24) -> int:
    """
    Cleanup old cache files.
```

```
    Args:
        max_age_hours: Maximum age of cache files in hours

    Returns:
        int: Number of files deleted
    """
    try:
        from datetime import timedelta

        cutoff_time = datetime.utcnow() - timedelta(hours=max_age_hours)
        deleted = 0

        for file_path in self.cache_dir.glob("*"):
            if file_path.is_file():
                file_time = datetime.fromtimestamp(file_path.stat().st_mtime)

                if file_time < cutoff_time:
                    file_path.unlink()
                    deleted += 1

        logger.info(f"Cleaned up {deleted} cache files")
        return deleted

    except Exception as e:
        logger.error(f"Cache cleanup failed: {e}")
        return 0

def save_evidence_file(
    self,
    submission_id: str,
    filename: str,
    content: bytes,
    evidence_type: str
) -> Path:
    """
    Save evidence file content to storage.

    Args:
        submission_id: Submission identifier
        filename: Original filename
        content: File content bytes
        evidence_type: Type of evidence

    Returns:
        Path: Path to saved file
    """
```

```
    try:
        # Get evidence path with sharding
        safe_filename = self._sanitize_filename(filename)
        file_path = self.get_evidence_path(submission_id, safe_filename)

        # Ensure directory exists
        file_path.parent.mkdir(parents=True, exist_ok=True)

        # Write file content
        with open(file_path, 'wb') as f:
            f.write(content)

        logger.debug(f"Saved evidence file: {file_path}")
        return file_path

    except Exception as e:
        logger.error(f"Failed to save evidence file: {e}")
        raise IOError(f"Failed to save evidence file: {str(e)}")

def _sanitize_filename(self, filename: str) -> str:
    """Sanitize filename for safe storage."""
    import re
    # Remove path components
    filename = Path(filename).name
    # Replace unsafe characters
    safe_name = re.sub(r'[^a-zA-Z0-9._-]', '_', filename)
    # Limit length
    if len(safe_name) > 255:
        name, ext = safe_name.rsplit('.', 1) if '.' in safe_name else (safe_name, '')
        safe_name = name[:250] + ('.' + ext if ext else '')
    return safe_name or 'unnamed_file'

def _safe_replace(self, src: Path, dst: Path, retries: int = 3) -> None:
    """
    Safely replace file, handling Windows locking issues.

    Args:
        src: Source path
        dst: Destination path
        retries: Number of retries on PermissionError
    """
    for i in range(retries):
        try:
            src.replace(dst)
            return
        except PermissionError:
```

```
            if i == retries - 1:
                raise

            # On Windows, destination might be open or locked temporarily
            # Try to unlink if it exists
            try:
                if dst.exists():
                    dst.unlink()
            except Exception:
                pass  # Ignore unlink errors, retry replace

            time.sleep(0.1)

def save_evidence_file(
    self,
    submission_id: str,
    filename: str,
    content: bytes,
    evidence_type: str
) -> Path:
    """
    Save evidence file content to storage.

    Args:
        submission_id: Submission identifier
        filename: Original filename
        content: File content bytes
        evidence_type: Type of evidence

    Returns:
        Path: Path to saved file
    """
    try:
        # Get evidence path with sharding
        safe_filename = self._sanitize_filename(filename)
        file_path = self.get_evidence_path(submission_id, safe_filename)

        # Ensure directory exists
        file_path.parent.mkdir(parents=True, exist_ok=True)

        # Write file content
        with open(file_path, 'wb') as f:
            f.write(content)

        logger.debug(f"Saved evidence file: {file_path}")
        return file_path
```

```
        except Exception as e:
            logger.error(f"Failed to save evidence file: {e}")
            raise IOError(f"Failed to save evidence file: {str(e)}")

    def _sanitize_filename(self, filename: str) -> str:
        """Sanitize filename for safe storage."""
        import re
        # Remove path components
        filename = Path(filename).name
        # Replace unsafe characters
        safe_name = re.sub(r'[^a-zA-Z0-9._-]', '_', filename)
        # Limit length
        if len(safe_name) > 255:
            name, ext = safe_name.rsplit('.', 1) if '.' in safe_name else (safe_name, '')
            safe_name = name[:250] + ('.' + ext if ext else '')
        return safe_name or 'unnamed_file'

"""
Validation Service - Input sanitization and validation

Provides:
- File upload validation (type, size, format)
- Text input sanitization
- Evidence type verification
- Malicious content detection
- Input length limits
- Security checks
"""

import logging
import mimetypes
import re
from pathlib import Path
from typing import Dict, List, Optional, Tuple

from PIL import Image

# Initialize logger
logger = logging.getLogger(__name__)


class ValidationService:
    """
    Validation Service - Input sanitization and security validation.
```

```
    Features:
    - File type and size validation
    - Text sanitization (XSS, SQL injection prevention)
    - Evidence format verification
    - Malicious content detection
    - Input length limits
    - Filename sanitization
    """

    # File size limits (bytes)
    MAX_IMAGE_SIZE = 50 * 1024 * 1024  # 50 MB
    MAX_VIDEO_SIZE = 500 * 1024 * 1024  # 500 MB
    MAX_AUDIO_SIZE = 100 * 1024 * 1024  # 100 MB
    MAX_DOCUMENT_SIZE = 20 * 1024 * 1024  # 20 MB

    # Text limits
    MAX_TEXT_LENGTH = 10000  # 10k characters
    MAX_NARRATIVE_LENGTH = 5000  # 5k characters

    # Allowed file extensions
    ALLOWED_IMAGE_EXTENSIONS = {'.jpg', '.jpeg', '.png', '.gif', '.bmp', '.tiff', '.webp'}
    ALLOWED_VIDEO_EXTENSIONS = {'.mp4', '.avi', '.mov', '.mkv', '.webm'}
    ALLOWED_AUDIO_EXTENSIONS = {'.mp3', '.wav', '.m4a', '.ogg', '.flac'}
    ALLOWED_DOCUMENT_EXTENSIONS = {'.pdf', '.doc', '.docx', '.txt'}

    # MIME type mappings
    ALLOWED_MIME_TYPES = {
        'image': {'image/jpeg', 'image/png', 'image/gif', 'image/bmp', 'image/tiff', 'image/
webp'},
        'video': {'video/mp4', 'video/x-msvideo', 'video/quicktime', 'video/x-matroska', 'video/
webm'},
        'audio': {'audio/mpeg', 'audio/wav', 'audio/mp4', 'audio/ogg', 'audio/flac'},
        'document': {'application/pdf', 'application/msword', 'text/plain'}
    }

    def __init__(self):
        """Initialize validation service."""
        logger.info("ValidationService initialized")

    def validate_file_content(
        self,
        filename: str,
        content: bytes,
        evidence_type: str
    ) -> Tuple[bool, Optional[str]]:
        """
```

```
        Validate file content (in-memory) for security and format compliance.

        Args:
            filename: Original filename
            content: File content bytes
            evidence_type: Type of evidence (image/video/audio/document)

        Returns:
            tuple: (is_valid, error_message)
        """
        try:
            # Check file size
            file_size = len(content)

            if evidence_type == 'image':
                if file_size > self.MAX_IMAGE_SIZE:
                    return False, f"Image size exceeds limit ({self.MAX_IMAGE_SIZE //
(1024*1024)} MB)"
            elif evidence_type == 'video':
                if file_size > self.MAX_VIDEO_SIZE:
                    return False, f"Video size exceeds limit ({self.MAX_VIDEO_SIZE //
(1024*1024)} MB)"
            elif evidence_type == 'audio':
                if file_size > self.MAX_AUDIO_SIZE:
                    return False, f"Audio size exceeds limit ({self.MAX_AUDIO_SIZE //
(1024*1024)} MB)"
            elif evidence_type == 'document':
                if file_size > self.MAX_DOCUMENT_SIZE:
                    return False, f"Document size exceeds limit
({self.MAX_DOCUMENT_SIZE // (1024*1024)} MB)"
            else:
                return False, f"Invalid evidence type: {evidence_type}"

            # Check file extension
            extension = Path(filename).suffix.lower() if filename else ''

            allowed_extensions = self.get_allowed_extensions(evidence_type)
            if extension not in allowed_extensions:
                return False, f"Invalid {evidence_type} extension: {extension}"

            # Verify MIME type / Magic Bytes
            # For in-memory, we can check magic bytes directly
            is_mime_valid, mime_error = self._verify_content_type(content, evidence_type)
            if not is_mime_valid:
                return False, mime_error
```

```
            # Additional checks for images
            if evidence_type == 'image':
                is_image_valid, image_error = self._verify_image_content(content)
                if not is_image_valid:
                    return False, image_error

            logger.debug(f"File content validation passed: {filename} ({evidence_type})")
            return True, None

        except Exception as e:
            logger.error(f"File content validation failed: {e}")
            return False, f"Validation error: {str(e)}"

    def validate_file_upload(
        self,
        file_path: Path,
        evidence_type: str
    ) -> Tuple[bool, Optional[str]]:
        """
        Validate file upload for security and format compliance.

        Args:
            file_path: Path to uploaded file
            evidence_type: Type of evidence (image/video/audio/document)

        Returns:
            tuple: (is_valid, error_message)
        """
        try:
            # Check file exists
            if not file_path.exists():
                return False, "File does not exist"

            # Check file size
            file_size = file_path.stat().st_size

            if evidence_type == 'image':
                if file_size > self.MAX_IMAGE_SIZE:
                    return False, f"Image size exceeds limit ({self.MAX_IMAGE_SIZE //
(1024*1024)} MB)"
            elif evidence_type == 'video':
                if file_size > self.MAX_VIDEO_SIZE:
                    return False, f"Video size exceeds limit ({self.MAX_VIDEO_SIZE //
(1024*1024)} MB)"
            elif evidence_type == 'audio':
                if file_size > self.MAX_AUDIO_SIZE:
```

```
                return False, f"Audio size exceeds limit ({self.MAX_AUDIO_SIZE //
(1024*1024)} MB)"
        elif evidence_type == 'document':
            if file_size > self.MAX_DOCUMENT_SIZE:
                return False, f"Document size exceeds limit
({self.MAX_DOCUMENT_SIZE // (1024*1024)} MB)"
        else:
            return False, f"Invalid evidence type: {evidence_type}"

        # Check file extension
        extension = file_path.suffix.lower()

        allowed_extensions = self.get_allowed_extensions(evidence_type)
        if extension not in allowed_extensions:
            return False, f"Invalid {evidence_type} extension: {extension}"

        # Verify MIME type
        is_mime_valid, mime_error = self._verify_mime_type(file_path, evidence_type)
        if not is_mime_valid:
            return False, mime_error

        # Additional checks for images
        if evidence_type == 'image':
            is_image_valid, image_error = self._verify_image_integrity(file_path)
            if not is_image_valid:
                return False, image_error

        logger.debug(f"File validation passed: {file_path.name} ({evidence_type})")
        return True, None

    except Exception as e:
        logger.error(f"File validation failed: {e}")
        return False, f"Validation error: {str(e)}"

def _verify_content_type(self, content: bytes, evidence_type: str) -> Tuple[bool,
Optional[str]]:
    """Verify content type from magic bytes."""
    try:
        import io
        # Use magic bytes check similar to _detect_mime_from_header but on content
        header = content[:16]
        mime_type = self._detect_mime_from_bytes(header)

        if not mime_type:
            # If magic bytes fail, we might trust the caller or return generic valid
            # For now, let's be strict only if we can identify.
```

```
                # Actually, if we can't identify, return True to avoid blocking valid files we just don't know
                return True, None

            allowed_mimes = self.ALLOWED_MIME_TYPES.get(evidence_type, set())
            if mime_type not in allowed_mimes:
                return False, f"File type mismatch (detected {mime_type})"

            return True, None
        except Exception as e:
            logger.warning(f"Content type verification failed: {e}")
            return True, None

    def _verify_image_content(self, content: bytes) -> Tuple[bool, Optional[str]]:
        """Verify image content integrity."""
        try:
            import io
            with Image.open(io.BytesIO(content)) as img:
                img.verify()
                if img.size[0] > 10000 or img.size[1] > 10000:
                    return False, "Image dimensions too large"
                if img.size[0] < 10 or img.size[1] < 10:
                    return False, "Image dimensions too small"
            return True, None
        except Exception as e:
            return False, f"Invalid image content: {e}"

    def _verify_mime_type(
        self,
        file_path: Path,
        evidence_type: str
    ) -> Tuple[bool, Optional[str]]:
        """
        Verify file MIME type matches declared type.

        Args:
            file_path: Path to file
            evidence_type: Declared evidence type

        Returns:
            tuple: (is_valid, error_message)
        """
        try:
            # Guess MIME type from file
            mime_type, _ = mimetypes.guess_type(str(file_path))
```

```
            if not mime_type:
                # Try reading file header for common types
                mime_type = self._detect_mime_from_header(file_path)

            if not mime_type:
                return False, "Could not determine file type"

            # Check if MIME type matches evidence type
            allowed_mimes = self.ALLOWED_MIME_TYPES.get(evidence_type, set())

            if mime_type not in allowed_mimes:
                return False, f"File type mismatch: expected {evidence_type}, got
{mime_type}"

            return True, None

        except Exception as e:
            logger.warning(f"MIME type verification failed: {e}")
            return True, None  # Don't fail validation on MIME check errors

    def _detect_mime_from_bytes(self, header: bytes) -> Optional[str]:
        """Detect MIME type from header bytes."""
        try:
            if header.startswith(b'\xff\xd8\xff'):
                return 'image/jpeg'
            elif header.startswith(b'\x89PNG\r\n\x1a\n'):
                return 'image/png'
            elif header.startswith(b'GIF87a') or header.startswith(b'GIF89a'):
                return 'image/gif'
            elif header.startswith(b'RIFF') and b'WEBP' in header:
                return 'image/webp'
            elif header.startswith(b'%PDF'):
                return 'application/pdf'
            elif header.startswith(b'ID3') or header.startswith(b'\xff\xfb'):
                return 'audio/mpeg'
            return None
        except Exception:
            return None

    def _detect_mime_from_header(self, file_path: Path) -> Optional[str]:
        """
        Detect MIME type from file header (magic bytes).

        Args:
            file_path: Path to file
```

```
    Returns:
        str: MIME type or None
    """
    try:
        with open(file_path, 'rb') as f:
            header = f.read(16)

        return self._detect_mime_from_bytes(header)

    except Exception as e:
        logger.warning(f"Magic byte detection failed: {e}")
        return None

def _verify_image_integrity(self, file_path: Path) -> Tuple[bool, Optional[str]]:
    """
    Verify image can be opened and is not corrupted.

    Args:
        file_path: Path to image

    Returns:
        tuple: (is_valid, error_message)
    """
    try:
        with Image.open(file_path) as img:
            # Try to load image
            img.verify()

            # Check dimensions (reasonable limits)
            if img.size[0] > 10000 or img.size[1] > 10000:
                return False, "Image dimensions too large (max 10000x10000)"

            if img.size[0] < 10 or img.size[1] < 10:
                return False, "Image dimensions too small (min 10x10)"

        return True, None

    except Exception as e:
        logger.warning(f"Image integrity check failed: {e}")
        return False, f"Invalid or corrupted image: {str(e)}"

def sanitize_text(self, text: str, max_length: Optional[int] = None) -> str:
    """
    Sanitize text input to prevent XSS and injection attacks.

    Args:
```

```
        text: Input text
        max_length: Maximum allowed length

    Returns:
        str: Sanitized text
    """
    if not text:
        return ""

    # Convert to string if not already
    text = str(text)

    # Apply length limit
    if max_length:
        text = text[:max_length]
    else:
        text = text[:self.MAX_TEXT_LENGTH]

    # Remove null bytes
    text = text.replace('\x00', '')

    # Remove control characters (except newlines and tabs)
    text = ''.join(char for char in text if char.isprintable() or char in '\n\r\t')

    # Trim whitespace
    text = text.strip()

    return text

def validate_text_narrative(self, narrative: str) -> Tuple[bool, Optional[str]]:
    """
    Validate text narrative for submission.

    Args:
        narrative: Text narrative

    Returns:
        tuple: (is_valid, error_message)
    """
    if not narrative:
        return False, "Narrative cannot be empty"

    # Check length
    if len(narrative) < 10:
        return False, "Narrative too short (minimum 10 characters)"
```

```
        if len(narrative) > self.MAX_NARRATIVE_LENGTH:
            return False, f"Narrative too long (maximum {self.MAX_NARRATIVE_LENGTH}
characters)"

        # Check for suspicious patterns
        suspicious_patterns = [
            r'<script',  # XSS
            r'javascript:',  # XSS
            r'on\w+\s*=',  # Event handlers
            r'eval\s*\(',  # Code execution
            r'exec\s*\(',  # Code execution
        ]

        narrative_lower = narrative.lower()
        for pattern in suspicious_patterns:
            if re.search(pattern, narrative_lower):
                logger.warning(f"Suspicious pattern detected in narrative: {pattern}")
                return False, "Narrative contains potentially malicious content"

        return True, None

    def sanitize_filename(self, filename: str) -> str:
        """
        Sanitize filename to prevent path traversal and other attacks.

        Args:
            filename: Original filename

        Returns:
            str: Sanitized filename
        """
        if not filename:
            return "unnamed_file"

        # Get base name (remove any path components)
        filename = Path(filename).name

        # Remove or replace dangerous characters
        # Allow: alphanumeric, dash, underscore, dot
        safe_chars = re.sub(r'[^a-zA-Z0-9._-]', '_', filename)

        # Prevent double extensions (.php.jpg)
        safe_chars = safe_chars.replace('..', '_')

        # Limit length
        if len(safe_chars) > 255:
```

```
            # Preserve extension
            name, ext = safe_chars.rsplit('.', 1) if '.' in safe_chars else (safe_chars, '')
            safe_chars = name[:250] + ('.' + ext if ext else '')

        # Ensure not empty
        if not safe_chars or safe_chars == '.':
            safe_chars = "unnamed_file"

        return safe_chars

    def validate_submission_id(self, submission_id: str) -> Tuple[bool, Optional[str]]:
        """
        Validate submission ID format.

        Args:
            submission_id: Submission identifier

        Returns:
            tuple: (is_valid, error_message)
        """
        if not submission_id:
            return False, "Submission ID cannot be empty"

        # Check length
        if len(submission_id) < 8 or len(submission_id) > 64:
            return False, "Invalid submission ID length"

        # Check format (alphanumeric, dash, underscore only)
        if not re.match(r'^[a-zA-Z0-9_-]+$', submission_id):
            return False, "Submission ID contains invalid characters"

        return True, None

    def validate_pseudonym(self, pseudonym: str) -> Tuple[bool, Optional[str]]:
        """
        Validate pseudonym format.

        Args:
            pseudonym: Pseudonym string

        Returns:
            tuple: (is_valid, error_message)
        """
        if not pseudonym:
            return False, "Pseudonym cannot be empty"
```

```
    # Check length
    if len(pseudonym) < 5 or len(pseudonym) > 100:
        return False, "Invalid pseudonym length"

    # Should start with expected prefix
    if not pseudonym.startswith('whistleblower-'):
        return False, "Invalid pseudonym format"

    return True, None

def check_sql_injection(self, text: str) -> bool:
    """
    Check text for SQL injection patterns.

    Args:
        text: Text to check

    Returns:
        bool: True if suspicious, False if clean
    """
    if not text:
        return False

    text_lower = text.lower()

    # Common SQL injection patterns
    sql_patterns = [
        r'union\s+select',
        r';\s*drop\s+table',
        r';\s*delete\s+from',
        r';\s*update\s+',
        r';\s*insert\s+into',
        r'--',  # SQL comment
        r'/\*.*\*/',  # SQL comment
        r'xp_cmdshell',
        r'exec\s*\(',
    ]

    for pattern in sql_patterns:
        if re.search(pattern, text_lower):
            logger.warning(f"SQL injection pattern detected: {pattern}")
            return True

    return False

def validate_evidence_type(self, evidence_type: str) -> Tuple[bool, Optional[str]]:
```

```
        """
        Validate evidence type.

        Args:
            evidence_type: Evidence type string

        Returns:
            tuple: (is_valid, error_message)
        """
        valid_types = {'image', 'video', 'audio', 'document'}

        if not evidence_type:
            return False, "Evidence type cannot be empty"

        if evidence_type.lower() not in valid_types:
            return False, f"Invalid evidence type. Must be one of: {', '.join(valid_types)}"

        return True, None

    def get_file_size_limit(self, evidence_type: str) -> int:
        """
        Get file size limit for evidence type.

        Args:
            evidence_type: Evidence type

        Returns:
            int: Size limit in bytes
        """
        limits = {
            'image': self.MAX_IMAGE_SIZE,
            'video': self.MAX_VIDEO_SIZE,
            'audio': self.MAX_AUDIO_SIZE,
            'document': self.MAX_DOCUMENT_SIZE
        }

        return limits.get(evidence_type.lower(), self.MAX_DOCUMENT_SIZE)

    def get_allowed_extensions(self, evidence_type: str) -> set:
        """
        Get allowed file extensions for evidence type.

        Args:
            evidence_type: Evidence type

        Returns:
```

```
            set: Allowed extensions
        """
        extensions = {
            'image': self.ALLOWED_IMAGE_EXTENSIONS,
            'video': self.ALLOWED_VIDEO_EXTENSIONS,
            'audio': self.ALLOWED_AUDIO_EXTENSIONS,
            'document': self.ALLOWED_DOCUMENT_EXTENSIONS
        }

        return extensions.get(evidence_type.lower(), set())

"""
Services Package - Centralized service exports

This module provides centralized imports for all service classes,
making it easier to import services throughout the application.

Usage:
    from backend.services import StorageService, CryptoService
"""

from backend.services.storage_service import StorageService
from backend.services.hash_chain_service import HashChainService
from backend.services.crypto_service import CryptoService
from backend.services.metadata_service import MetadataService
from backend.services.validation_service import ValidationService
from backend.services.rate_limiter import RateLimiter
from backend.services.queue_service import QueueService, Job, JobStatus, JobPriority
from backend.services.metrics_service import MetricsService

__all__ = [
    # Storage and data management
    'StorageService',
    'HashChainService',

    # Security services
    'CryptoService',
    'MetadataService',
    'ValidationService',
    'RateLimiter',

    # Background processing
    'QueueService',
    'Job',
    'JobStatus',
    'JobPriority',
```

```
    # Monitoring
    'MetricsService',
]

__version__ = '0.1.0'
```