Ingestion Engine

Task:

The goal of this task was to create an API based ingestion pipeline, using multi format passes standardised into ontological graphs.

Overview:

I've created a comprehensive API-based ingestion pipeline that processes multiple data formats and standardizes them into ontological graphs. This pipeline provides a flexible and extensible framework for data integration and semantic modelling. It comes with many features which include:

1. **Multi-format Support:** Processes CSV, JSON, XML, RDF, YAML, and text data.
2. **Transformation Options:**
    - Direct mapping - automatic conversion to RDF triples.
    - Schema mapping - structured data mapping using configuration.
    - Ontology alignment - aligns with existing ontologies.
    - Custom transformations - for complex data processing.
3. **Output Formats:**
    - RDF (Resource Description Framework).
    - Graph databases.
    - NetworkX graphs.
    - JSON-LD (JSON for Linked Data).
4. **Pipeline Management:**
    - Create reusable ingestion pipelines.
    - Track job status.
    - Retrieve results in various forms.
5. **Validation:**
    - Schema validation.
    - Ontology consistency checker.

The system uses FastAPI for the API layer and employs RDFLib for graph processing. Jobs are processed asynchronously to handle large datasets efficiently. The transformation pipeline is modular, allowing for easy extension with new formats and transformation types.

Tools/Resources Used:

A variety of tools/resources were used to help create this pipeline, they included:

- Google
- Stack Overflow

- Visual Studio Code
- Claude.ai
- Apache Tika
- SpaCy

Step by Step Process:

1. **Imports**: Libraries are imported at the top of the file.
2. **Logging Setup**: Basic logging is configured.
3. **FastAPI App Initialization**: The app object is created with a title and description.
4. **Class Definitions/Data Models**: Multiple classes are defined in this order:
    - Enum classes (SourceType, TransformationType, StorageFormat, ProcessingStatus).
    - Pydantic models (MappingRule, OntologyMapping, IngestionRequest, IngestionResponse, IngestionStatus).
5. **In-memory Storage**: A dictionary called ‘jobs_store’ is used to track job statuses and results.
6. **Custom Exception**: The ‘IngestionError’ class is defined.
7. **Core Class Definition**: The ‘IngestionPipeline’ class is defined with all its methods:
    - ‘__init__()’, ‘initialize_namespaces()’.
    - ‘process()’ to orchestrate the pipeline.
    - Input parsing methods ‘(parse_input())’.
    - Transformation methods ‘(transform_data()’, ‘direct_mapping_transform()’, etc.).
    - Format conversion methods.
    - Validation methods.
8. **Helper Functions**: Utility functions like ‘generate_job_id()’ and ‘update_job_status()’ are defined.
9. **API Endpoint Definitions**: The FastAPI endpoints are defined in this order:
    - ‘/ingest/file’: Upload and process a file.
    - ‘/ingest/data’: Process data from a request body.
    - ‘/ingest/status/{job_id}’: Check job status.
    - ‘/ingest/result/{job_id}’: Get processing results.
    - ‘/ingest/schema/domain’: Get domain-specific ontology schema.
    - ‘/ingest/mappings/domain-examples’: Get example mapping configurations.
    - ‘/ingest/schema/example’: Get an example ontology schema.
    - ‘/ingest/pipeline’: Create a reusable ingestion pipeline.
    - ‘/ingest/pipelines’: List available pipelines

10. **Domain Schema Definition**: The ‘get_domain_ontology_schema()’ function is defined.
11. **Example Schema Endpoint**: Additional endpoint is defined, to give an idea on how it works.
12. **Main Entry Point**: The application is set up to run with uvicorn on port 8000 if executed.

When an API request is received, the execution flow is:

1. The request is received by the corresponding endpoint.
2. A job ID is generated.
3. An IngestionPipeline instance is created.
4. The data is processed according to the configuration.
5. Status updates are stored in the jobs_store.
6. The response is returned to the client.

The Code Structure:

```
import os
import json
import logging
from typing import Dict, List, Any, Optional, Union
from enum import Enum
from datetime import datetime
from fastapi import FastAPI, File, UploadFile, HTTPException, Body, Depends,
Query, status
from pydantic import BaseModel, Field
from rdflib import Graph, Namespace, Literal, URIRef, BNode
from rdflib.namespace import RDF, RDFS, OWL, XSD
import pandas as pd
import networkx as nx
from concurrent.futures import ThreadPoolExecutor
import xml.etree.ElementTree as ET
import csv
import yaml

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize FastAPI app
app = FastAPI(
    title="Ontological Graph Ingestion Engine",
    description="API for ingesting multi-format data and standardizing into
ontological graphs",
    version="1.0.0"
)
```

```python
# Define data models
class SourceType(str, Enum):
    CSV = "csv"
    JSON = "json"
    XML = "xml"
    RDF = "rdf"
    YAML = "yaml"
    TEXT = "text"
    DATABASE = "database"
    API = "api"

class TransformationType(str, Enum):
    DIRECT_MAPPING = "direct_mapping"
    SCHEMA_MAPPING = "schema_mapping"
    ONTOLOGY_ALIGNMENT = "ontology_alignment"
    CUSTOM = "custom"

class StorageFormat(str, Enum):
    RDF = "rdf"
    GRAPH_DB = "graph_db"
    NETWORKX = "networkx"
    JSON_LD = "json_ld"

class ProcessingStatus(str, Enum):
    PENDING = "pending"
    PROCESSING = "processing"
    COMPLETED = "completed"
    FAILED = "failed"

class MappingRule(BaseModel):
    source_field: str
    target_property: str
    transformation: Optional[str] = None
    data_type: Optional[str] = None

class OntologyMapping(BaseModel):
    source_type: SourceType
    target_namespace: str
    class_mappings: Dict[str, str] = {}
    property_mappings: List[MappingRule] = []
    custom_transformations: Dict[str, str] = {}

class IngestionRequest(BaseModel):
    source_type: SourceType
    transformation_type: TransformationType
    target_format: StorageFormat
    mapping_config: Optional[OntologyMapping] = None
    schema_validation: bool = False
    ontology_validation: bool = False
    pipeline_id: Optional[str] = None
    custom_options: Optional[Dict[str, Any]] = None

class IngestionResponse(BaseModel):
    job_id: str
    status: ProcessingStatus
    created_at: datetime
    message: str
    errors: Optional[List[str]] = None

class IngestionStatus(BaseModel):
    job_id: str
    status: ProcessingStatus
    created_at: datetime
    completed_at: Optional[datetime] = None
    records_processed: Optional[int] = None
    errors: Optional[List[str]] = None
    graph_stats: Optional[Dict[str, Any]] = None

# In-memory storage for job status (would be a database in production)
jobs_store = {}

# Custom exception for ingestion errors
class IngestionError(Exception):
    def __init__(self, message: str, details: Optional[List[str]] = None):
        self.message = message
        self.details = details or []
        super().__init__(self.message)

# Core ingestion pipeline class
class IngestionPipeline:
    def __init__(self, config: IngestionRequest, job_id: str):
        self.config = config
        self.job_id = job_id
        self.graph = Graph()  # RDFLib graph
        self.nx_graph = nx.DiGraph()  # NetworkX graph for alternative
processing
        self.namespaces = {}
        self.errors = []
        self.status = ProcessingStatus.PENDING
        self.records_processed = 0

        # Initialize default namespaces
        self.initialize_namespaces()

    def initialize_namespaces(self):
        # Default namespaces
        self.namespaces["rdf"] = RDF
        self.namespaces["rdfs"] = RDFS
        self.namespaces["owl"] = OWL
        self.namespaces["xsd"] = XSD

        # Add custom namespace if provided in mapping config
        if self.config.mapping_config and 
self.config.mapping_config.target_namespace:
            ns_name = self.config.mapping_config.target_namespace.split(":")[-
1]
            ns_uri = self.config.mapping_config.target_namespace
            self.namespaces[ns_name] = Namespace(ns_uri)
            self.graph.bind(ns_name, self.namespaces[ns_name])

    async def process(self, data, filename: Optional[str] = None) -> bool:
        try:
            self.status = ProcessingStatus.PROCESSING
            update_job_status(self.job_id, self.status)

            # Parse the input data based on source type
            parsed_data = self.parse_input(data, filename)
            if not parsed_data:
                raise IngestionError("Failed to parse input data")

            # Transform data according to the specified transformation type
            transformed_data = self.transform_data(parsed_data)

            # Convert to target format
            result = self.convert_to_target_format(transformed_data)

            # Run validations if configured
            if self.config.schema_validation:
                self.validate_schema()

            if self.config.ontology_validation:
                self.validate_ontology()

            self.status = ProcessingStatus.COMPLETED
            update_job_status(
                self.job_id,
                self.status,
                records_processed=self.records_processed,
                graph_stats=self.get_graph_stats()
            )
            return True

        except IngestionError as e:
```

```python
            self.status = ProcessingStatus.FAILED
            self.errors.append(str(e))
            if e.details:
                self.errors.extend(e.details)
            logger.error(f"Ingestion error: {str(e)}")
            update_job_status(self.job_id, self.status, errors=self.errors)
            return False
        except Exception as e:
            self.status = ProcessingStatus.FAILED
            self.errors.append(f"Unexpected error: {str(e)}")
            logger.exception("Unexpected error during ingestion")
            update_job_status(self.job_id, self.status, errors=self.errors)
            return False

    def parse_input(self, data, filename: Optional[str] = None):
        """Parse input data based on source type"""
        logger.info(f"Parsing input data of type {self.config.source_type}")

        try:
            if self.config.source_type == SourceType.CSV:
                return pd.read_csv(data)

            elif self.config.source_type == SourceType.JSON:
                if isinstance(data, bytes):
                    return json.loads(data.decode('utf-8'))
                return json.loads(data)

            elif self.config.source_type == SourceType.XML:
                return ET.parse(data)

            elif self.config.source_type == SourceType.RDF:
                g = Graph()
                g.parse(data=data, format="xml")
                return g

            elif self.config.source_type == SourceType.YAML:
                if isinstance(data, bytes):
                    return yaml.safe_load(data.decode('utf-8'))
                return yaml.safe_load(data)

            elif self.config.source_type == SourceType.TEXT:
                if isinstance(data, bytes):
                    return data.decode('utf-8')
                return data

            elif self.config.source_type == SourceType.DATABASE:
                # Would implement database connection logic here
                raise NotImplementedError("Database ingestion not
implemented")

            elif self.config.source_type == SourceType.API:
                # Would implement API fetching logic here
                raise NotImplementedError("API ingestion not implemented")

            else:
                raise IngestionError(f"Unsupported source type:
{self.config.source_type}")

        except Exception as e:
            raise IngestionError(f"Error parsing {self.config.source_type}
data", [str(e)])

    def transform_data(self, data):
        """Transform parsed data according to transformation type"""
        logger.info(f"Transforming data using
{self.config.transformation_type}")

        if self.config.transformation_type ==
TransformationType.DIRECT_MAPPING:
            return self.direct_mapping_transform(data)

        elif self.config.transformation_type ==
TransformationType.SCHEMA_MAPPING:
            return self.schema_mapping_transform(data)

        elif self.config.transformation_type ==
TransformationType.ONTOLOGY_ALIGNMENT:
            return self.ontology_alignment_transform(data)

        elif self.config.transformation_type == TransformationType.CUSTOM:
            return self.custom_transform(data)

        else:
            raise IngestionError(f"Unsupported transformation type:
{self.config.transformation_type}")

    def direct_mapping_transform(self, data):
        """Transform data using direct mapping approach"""
        if self.config.source_type == SourceType.CSV or isinstance(data,
pd.DataFrame):
            return self.csv_to_graph(data)
        elif self.config.source_type == SourceType.JSON:
            return self.json_to_graph(data)
        elif self.config.source_type == SourceType.XML:
            return self.xml_to_graph(data)
        elif self.config.source_type == SourceType.RDF:
            # RDF data already in graph format
            self.graph = data
            return self.graph
        else:
            raise IngestionError(f"Direct mapping not supported for
{self.config.source_type}")

    def schema_mapping_transform(self, data):
        """Transform data using schema mapping approach"""
        if not self.config.mapping_config:
            raise IngestionError("Schema mapping requires mapping
configuration")

        # Apply schema mapping rules based on the source type
        if self.config.source_type == SourceType.CSV or isinstance(data,
pd.DataFrame):
            return self.apply_schema_mapping_to_tabular(data)
        elif self.config.source_type == SourceType.JSON:
            return self.apply_schema_mapping_to_json(data)
        elif self.config.source_type == SourceType.XML:
            return self.apply_schema_mapping_to_xml(data)
        else:
            raise IngestionError(f"Schema mapping not implemented for
{self.config.source_type}")

    def ontology_alignment_transform(self, data):
        """Transform data using ontology alignment approach"""
        if not self.config.mapping_config:
            raise IngestionError("Ontology alignment requires mapping
configuration")

        # Apply ontology mapping logic based on source type
        if isinstance(data, Graph):
            return self.align_ontologies(data)
        else:
            # First convert to RDF graph then align
            temp_graph = self.direct_mapping_transform(data)
            return self.align_ontologies(temp_graph)

    def custom_transform(self, data):
        """Apply custom transformations"""
        if not self.config.mapping_config or not
self.config.mapping_config.custom_transformations:
            raise IngestionError("Custom transformation requires custom
transformation definitions")

        # In a real implementation, this might execute custom code or rules
        # For demonstration, we'll fall back to direct mapping
        logger.warning("Custom transformation falling back to direct mapping")
        return self.direct_mapping_transform(data)

    def csv_to_graph(self, df):
        """Convert CSV/DataFrame to RDF graph"""
        base_ns = self.namespaces.get("ex", Namespace("http://example.org/"))

        # Create a class for the CSV table
        table_class = URIRef(base_ns + "Table")
        self.graph.add((table_class, RDF.type, RDFS.Class))

        # Process each row
        for idx, row in df.iterrows():
            # Create a resource for this row
            row_uri = URIRef(f"{base_ns}row_{idx}")
            self.graph.add((row_uri, RDF.type, table_class))

            # Add properties for each column
            for col_name in df.columns:
                if pd.notna(row[col_name]):
                    # Create property
                    prop_uri = URIRef(f"{base_ns}{col_name}")
                    self.graph.add((prop_uri, RDF.type, RDF.Property))

                    # Add value
                    value = row[col_name]
                    if isinstance(value, (int, float)):
                        self.graph.add((row_uri, prop_uri, Literal(value)))
                    else:
                        self.graph.add((row_uri, prop_uri,
Literal(str(value))))

            self.records_processed += 1

        return self.graph

    def json_to_graph(self, json_data):
        """Convert JSON to RDF graph"""
        base_ns = self.namespaces.get("ex", Namespace("http://example.org/"))

        def process_json_object(obj, parent_uri=None, rel=None):
            if parent_uri is None:
                # Root object
                obj_uri = URIRef(base_ns + "root")
                self.graph.add((obj_uri, RDF.type, RDFS.Resource))
            else:
                # Generate URI for this object
                obj_uri = BNode()
                if rel:
                    self.graph.add((parent_uri, URIRef(base_ns + rel),
obj_uri))

            if isinstance(obj, dict):
                # Process dict object
                for key, value in obj.items():
                    if isinstance(value, (dict, list)):
                        process_json_object(value, obj_uri, key)
                    else:
                        # Add as literal property
                        prop_uri = URIRef(base_ns + key)
                        if value is not None:
                            self.graph.add((obj_uri, prop_uri,
Literal(value)))

                self.records_processed += 1

            elif isinstance(obj, list):
                # Process list as multiple related objects
                for i, item in enumerate(obj):
                    if isinstance(item, (dict, list)):
                        item_uri = process_json_object(item)
                        if rel:
                            list_node = BNode()
                            self.graph.add((parent_uri, URIRef(base_ns + rel),
list_node))
                            self.graph.add((list_node, RDF.type, RDF.Seq))
                            self.graph.add((list_node,
URIRef(f"{RDF}_#{i+1}"), item_uri))
                    else:
                        # Add as literal in sequence
                        if rel:
                            list_node = BNode()
                            self.graph.add((parent_uri, URIRef(base_ns + rel),
list_node))
                            self.graph.add((list_node, RDF.type, RDF.Seq))
                            self.graph.add((list_node,
URIRef(f"{RDF}_#{i+1}"), Literal(item)))

            return obj_uri

        process_json_object(json_data)
        return self.graph

    def xml_to_graph(self, xml_data):
        """Convert XML to RDF graph"""
        base_ns = self.namespaces.get("ex", Namespace("http://example.org/"))
        root = xml_data.getroot()

        def process_element(element, parent_uri=None):
            # Create URI for this element
            if parent_uri is None:
                elem_uri = URIRef(base_ns + element.tag)
            else:
                elem_uri = BNode()
                self.graph.add((parent_uri, URIRef(base_ns + element.tag),
elem_uri))

            # Add element class
            self.graph.add((elem_uri, RDF.type, URIRef(base_ns + element.tag +
"Type")))

            # Process attributes
            for attr_name, attr_value in element.attrib.items():
                attr_uri = URIRef(base_ns + attr_name)
                self.graph.add((elem_uri, attr_uri, Literal(attr_value)))

            # Process text content if any and no children
            if element.text and element.text.strip() and len(element) == 0:
                self.graph.add((elem_uri, RDFS.label,
Literal(element.text.strip())))

            # Process child elements
            for child in element:
                process_element(child, elem_uri)

            self.records_processed += 1
            return elem_uri

        process_element(root)
        return self.graph

    def apply_schema_mapping_to_tabular(self, df):
        """Apply schema mapping to tabular data"""
        mapping = self.config.mapping_config
        base_ns = Namespace(mapping.target_namespace)

        # Get class mapping or use default
        class_uri = URIRef(base_ns + list(mapping.class_mappings.values())[0])
if mapping.class_mappings else URIRef(base_ns + "Record")

        # Process each row
        for idx, row in df.iterrows():
            # Create instance
            instance_uri = URIRef(f"{base_ns}instance_{idx}")
            self.graph.add((instance_uri, RDF.type, class_uri))

            # Apply property mappings
            for rule in mapping.property_mappings:
                if rule.source_field in df.columns and
pd.notna(row[rule.source_field]):
                    value = row[rule.source_field]

                    # Apply transformation if specified
                    if rule.transformation:
                        value = self.apply_transformation(value,
rule.transformation)

                    # Create property URI and add to graph
                    prop_uri = URIRef(base_ns + rule.target_property)

                    # Handle data type
                    if rule.data_type and rule.data_type.startswith('xsd:'):
                        xsd_type = getattr(XSD, rule.data_type[4:])
                        self.graph.add((instance_uri, prop_uri, Literal(value,
datatype=xsd_type)))
                    else:
                        self.graph.add((instance_uri, prop_uri,
Literal(value)))

            self.records_processed += 1

        return self.graph

    def apply_schema_mapping_to_json(self, json_data):
        """Apply schema mapping to JSON data"""
        mapping = self.config.mapping_config
        base_ns = Namespace(mapping.target_namespace)

        def extract_value(obj, path):
            """Extract value from nested JSON using dot notation path"""
            if "." in path:
                parts = path.split(".", 1)
                key, rest = parts[0], parts[1]

                # Handle array indexing
                if "[" in key and key.endswith("]"):
                    array_key, idx_str = key.split("[", 1)
                    idx = int(idx_str[:-1])  # Remove closing bracket
                    if array_key in obj and isinstance(obj[array_key], list)
and idx < len(obj[array_key]):
                        return extract_value(obj[array_key][idx], rest)
                    return None

                # Handle nested object
                if key in obj and isinstance(obj[key], dict):
                    return extract_value(obj[key], rest)
                return None
            else:
                return obj.get(path)

        def process_json_object(obj, idx=0):
            # Get class mapping or use default
            class_uri = URIRef(base_ns +
list(mapping.class_mappings.values())[0]) if mapping.class_mappings else
URIRef(base_ns + "Record")

            # Create instance
            instance_uri = URIRef(f"{base_ns}instance_{idx}")
            self.graph.add((instance_uri, RDF.type, class_uri))

            # Apply property mappings
            for rule in mapping.property_mappings:
                value = extract_value(obj, rule.source_field)
                if value is not None:
                    # Apply transformation if specified
                    if rule.transformation:
                        value = self.apply_transformation(value,
rule.transformation)

                    # Create property URI and add to graph
                    prop_uri = URIRef(base_ns + rule.target_property)

                    # Handle data type
                    if rule.data_type and rule.data_type.startswith('xsd:'):
                        xsd_type = getattr(XSD, rule.data_type[4:])
                        self.graph.add((instance_uri, prop_uri, Literal(value,
datatype=xsd_type)))
                    else:
                        self.graph.add((instance_uri, prop_uri,
Literal(value)))

            self.records_processed += 1
            return instance_uri

        # Handle both single object and array of objects
        if isinstance(json_data, list):
            for i, item in enumerate(json_data):
                if isinstance(item, dict):
                    process_json_object(item, i)
```

```python
        elif isinstance(json_data, dict):
            process_json_object(json_data)

        return self.graph

    def apply_schema_mapping_to_xml(self, xml_data):
        """Apply schema mapping to XML data"""
        mapping = self.config.mapping_config
        base_ns = Namespace(mapping.target_namespace)
        root = xml_data.getroot()

        def find_element_by_path(element, path):
            """Find XML element using XPath-like syntax"""
            if "/" in path:
                parts = path.split("/", 1)
                tag, rest = parts[0], parts[1]

                for child in element.findall(tag):
                    result = find_element_by_path(child, rest)
                    if result is not None:
                        return result
                return None
            else:
                return element.find(path)

        def extract_value(element, path):
            """Extract value from XML element using path"""
            if "@" in path:
                # Handle attribute
                elem_path, attr = path.split("@", 1)
                if elem_path:
                    target_elem = find_element_by_path(element, elem_path)
                    if target_elem is not None:
                        return target_elem.get(attr)
                else:
                    return element.get(attr)
                return None
            else:
                # Handle element text
                target_elem = find_element_by_path(element, path)
                if target_elem is not None:
                    return target_elem.text
                return None

        def process_xml_element(element, idx=0):
            # Get class mapping or use default
            class_uri = URIRef(base_ns +
list(mapping.class_mappings.values())[0]) if mapping.class_mappings else
URIRef(base_ns + "Record")

            # Create instance
            instance_uri = URIRef(f"{base_ns}instance_{idx}")
            self.graph.add((instance_uri, RDF.type, class_uri))

            # Apply property mappings
            for rule in mapping.property_mappings:
                value = extract_value(element, rule.source_field)
                if value is not None:
                    # Apply transformation if specified
                    if rule.transformation:
                        value = self.apply_transformation(value,
rule.transformation)

                    # Create property URI and add to graph
                    prop_uri = URIRef(base_ns + rule.target_property)

                    # Handle data type
                    if rule.data_type and rule.data_type.startswith('xsd:'):
                        xsd_type = getattr(XSD, rule.data_type[4:])
                        self.graph.add((instance_uri, prop_uri, Literal(value,
datatype=xsd_type)))
                    else:
                        self.graph.add((instance_uri, prop_uri,
Literal(value)))

            self.records_processed += 1
            return instance_uri

        # Process according to mapping
        process_xml_element(root)

        return self.graph

    def align_ontologies(self, source_graph):
        """Align source ontology with target ontology"""
        mapping = self.config.mapping_config
        target_ns = Namespace(mapping.target_namespace)

        # In a real implementation, this would use sophisticated ontology
alignment techniques
        # For demonstration, we'll do a simple mapping based on the mapping
rules

        # Class mappings
        for source_class, target_class in mapping.class_mappings.items():
            source_class_uri = URIRef(source_class)
            target_class_uri = URIRef(target_ns + target_class)

            # Add target class definition
            self.graph.add((target_class_uri, RDF.type, OWL.Class))

            # Find all instances of source class
            for instance in source_graph.subjects(RDF.type, source_class_uri):
                # Add instance to target class
                self.graph.add((instance, RDF.type, target_class_uri))

        # Property mappings
        for rule in mapping.property_mappings:
            source_prop = URIRef(rule.source_field)
            target_prop = URIRef(target_ns + rule.target_property)

            # Add target property definition
            self.graph.add((target_prop, RDF.type, RDF.Property))

            # Map properties
            for s, o in source_graph.subject_objects(source_prop):
                if rule.transformation:
                    value = self.apply_transformation(o, rule.transformation)
                    self.graph.add((s, target_prop, value))
                else:
                    self.graph.add((s, target_prop, o))

        self.records_processed = len(list(self.graph.subjects(None, None)))
        return self.graph

    def apply_transformation(self, value, transformation):
        """Apply transformation to a value"""
        # This is a simplified version - a real implementation would support
more transformations

        # Handle built-in transformations
        if transformation == "uppercase" and isinstance(value, str):
            return value.upper()
        elif transformation == "lowercase" and isinstance(value, str):
            return value.lower()
        elif transformation == "trim" and isinstance(value, str):
            return value.strip()
        elif transformation.startswith("concat:") and isinstance(value, str):
            suffix = transformation.split(":", 1)[1]
            return value + suffix
        elif transformation == "integer" and not isinstance(value, int):
            try:
                return int(value)
            except (ValueError, TypeError):
                return value
        elif transformation == "float" and not isinstance(value, float):
            try:
                return float(value)
            except (ValueError, TypeError):
                return value

        # Custom transformations would be implemented here
        # This could involve calling external functions, etc.

        # Default: return original value
        return value

    def convert_to_target_format(self, transformed_data):
        """Convert the transformed data to the target format"""
        if self.config.target_format == StorageFormat.RDF:
            # Already in RDFLib format
            return self.graph

        elif self.config.target_format == StorageFormat.GRAPH_DB:
            # In a real implementation, this would store to a graph database
            # For demonstration, we'll just return the RDF
            logger.warning("Graph DB storage simulated - returning RDF")
            return self.graph

        elif self.config.target_format == StorageFormat.NETWORKX:
            # Convert RDFLib graph to NetworkX
            for s, p, o in self.graph:
                self.nx_graph.add_edge(str(s), str(o), predicate=str(p))
            return self.nx_graph

        elif self.config.target_format == StorageFormat.JSON_LD:
            # Convert to JSON-LD format
            jsonld_data = []

            # Group by subject
            subjects = {}
            for s, p, o in self.graph:
                s_str = str(s)
                if s_str not in subjects:
                    subjects[s_str] = {"@id": s_str}

                p_str = str(p)
                if isinstance(o, Literal):
                    o_value = o.value
                    if o.datatype:
                        # Handle typed literals
                        if "@type" not in subjects[s_str]:
                            subjects[s_str]["@type"] = []
                        subjects[s_str]["@type"].append(str(o.datatype))
                else:
                    o_value = {"@id": str(o)}

                if p_str not in subjects[s_str]:
                    subjects[s_str][p_str] = []
                subjects[s_str][p_str].append(o_value)

            # Convert to list
            for subject_data in subjects.values():
                jsonld_data.append(subject_data)

            return jsonld_data

        else:
            raise IngestionError(f"Unsupported target format:
{self.config.target_format}")

    def validate_schema(self):
        """Validate graph against a schema"""
        # In a real implementation, this would validate against SHACL shapes
or similar
        logger.info("Schema validation requested but not implemented")
        return True

    def validate_ontology(self):
        """Validate graph for ontological consistency"""
        # In a real implementation, this would check ontological consistency
        logger.info("Ontology validation requested but not implemented")
        return True

    def get_graph_stats(self):
        """Get statistics about the generated graph"""
        stats = {
            "triples_count": len(self.graph),
            "subjects_count": len(set(self.graph.subjects())),
            "predicates_count": len(set(self.graph.predicates())),
            "objects_count": len(set(self.graph.objects())),
            "classes_count": len(set(self.graph.subjects(RDF.type,
RDFS.Class))),
            "properties_count": len(set(self.graph.subjects(RDF.type,
RDF.Property)))
        }
        return stats
```

```python
# Helper functions
def generate_job_id():
    """Generate a unique job ID"""
    import uuid
    return str(uuid.uuid4())

def update_job_status(job_id, status, **kwargs):
    """Update job status in the store"""
    if job_id not in jobs_store:
        jobs_store[job_id] = {
            "job_id": job_id,
            "status": status,
            "created_at": datetime.now(),
            "errors": [],
            "records_processed": 0
        }
    else:
        jobs_store[job_id]["status"] = status

        if "errors" in kwargs and kwargs["errors"]:
            jobs_store[job_id]["errors"] = kwargs["errors"]

        if "records_processed" in kwargs:
            jobs_store[job_id]["records_processed"] =
kwargs["records_processed"]

        if "graph_stats" in kwargs:
            jobs_store[job_id]["graph_stats"] = kwargs["graph_stats"]

        if status == ProcessingStatus.COMPLETED or status ==
ProcessingStatus.FAILED:
            jobs_store[job_id]["completed_at"] = datetime.now()

# API Endpoints
@app.get("/ingest/schema/domain", response_model=Dict[str, Any])
async def get_domain_schema():
    """
    Get the domain-specific ontology schema with support objects, adaptive
objects, and core objects
    """
    return get_domain_ontology_schema()

@app.get("/ingest/mappings/domain-examples", response_model=Dict[str, Any])
async def get_domain_mapping_examples():
    """
    Get example mapping configurations for domain-specific object types
    """
    examples = {
```

```
    "support_objects_mapping": {
        "source_type": "json",
        "target_namespace": "http://example.org/domain-ontology/",
        "class_mappings": {
            "strategy": "BusinessStrategy",
            "architecture": "ArchitecturalStability",
            "social": "SocialNetwork"
        },
        "property_mappings": [
            {
                "source_field": "name",
                "target_property": "hasName"
            },
            {
                "source_field": "description",
                "target_property": "hasDescription"
            },
            {
                "source_field": "objective",
                "target_property": "hasObjective"
            },
            {
                "source_field": "resilience_score",
                "target_property": "hasResilience",
                "data_type": "xsd:decimal"
            },
            {
                "source_field": "connections_count",
                "target_property": "hasConnections",
                "data_type": "xsd:integer"
            }
        ]
    },
    "adaptive_objects_mapping": {
        "source_type": "json",
        "target_namespace": "http://example.org/domain-ontology/",
        "class_mappings": {
            "feedback": "FeedbackLoop",
            "ml_model": "MachineLearning",
            "threat": "ThreatIntelligence"
        },
        "property_mappings": [
            {
                "source_field": "name",
                "target_property": "hasName"
            },
            {
                "source_field": "description",
                "target_property": "hasDescription"
            },
            {
                "source_field": "input_source",
                "target_property": "hasInput"
            },
            {
                "source_field": "algorithm_type",
                "target_property": "hasAlgorithm"
            },
            {
                "source_field": "accuracy_score",
                "target_property": "hasAccuracy",
                "data_type": "xsd:decimal"
            },
            {
                "source_field": "threat_severity",
                "target_property": "hasSeverity"
            }
        ]
    },
    "core_objects_mapping": {
        "source_type": "json",
        "target_namespace": "http://example.org/domain-ontology/",
        "class_mappings": {
            "structure": "DesignedStructure",
            "endeavor": "HumanEndeavor",
            "information": "SupportedInformation",
            "agent": "Agent",
            "capability": "Capability"
        },
        "property_mappings": [
            {
                "source_field": "name",
                "target_property": "hasName"
            },
            {
                "source_field": "description",
                "target_property": "hasDescription"
            },
            {
                "source_field": "is_essential",
                "target_property": "isEssential",
                "data_type": "xsd:boolean"
            },
            {
                "source_field": "designer",
                "target_property": "hasDesigner"
                },
                {
                    "source_field": "participants",
                    "target_property": "hasParticipants",
                    "data_type": "xsd:integer"
                },
                {
                    "source_field": "information_source",
                    "target_property": "hasSource"
                },
                {
                    "source_field": "autonomy_level",
                    "target_property": "hasAutonomy",
                    "data_type": "xsd:decimal"
                }
            ]
        }
    }

    return examples

@app.post("/ingest/file", response_model=IngestionResponse)
async def ingest_file(
    config: IngestionRequest = Body(...),
    file: UploadFile = File(...),
):
    """
    Ingest data from a file upload and convert to ontological graph
    """
    job_id = generate_job_id()

    try:
        # Create pipeline
        pipeline = IngestionPipeline(config, job_id)

        # Process in background
        with ThreadPoolExecutor() as executor:
            executor.submit(pipeline.process, await file.read(),
file.filename)

        return IngestionResponse(
            job_id=job_id,
            status=ProcessingStatus.PENDING,
            created_at=datetime.now(),
            message=f"Ingestion job started for file: {file.filename}"
        )
    except Exception as e:
        logger.exception("Error starting ingestion job")
```

```python
        return IngestionResponse(
            job_id=job_id,
            status=ProcessingStatus.FAILED,
            created_at=datetime.now(),
            message="Failed to start ingestion job",
            errors=[str(e)]
        )

@app.post("/ingest/data", response_model=IngestionResponse)
async def ingest_data(
    config: IngestionRequest = Body(...),
    data: Dict[str, Any] = Body(..., description="Raw data to ingest")
):
    """
    Ingest data from request body and convert to ontological graph
    """
    job_id = generate_job_id()

    try:
        # Create pipeline
        pipeline = IngestionPipeline(config, job_id)

        # Process in background
        with ThreadPoolExecutor() as executor:
            if config.source_type == SourceType.JSON:
                executor.submit(pipeline.process, json.dumps(data))
            else:
                executor.submit(pipeline.process, str(data))

        return IngestionResponse(
            job_id=job_id,
            status=ProcessingStatus.PENDING,
            created_at=datetime.now(),
            message="Ingestion job started for provided data"
        )
    except Exception as e:
        logger.exception("Error starting ingestion job")
        return IngestionResponse(
            job_id=job_id,
            status=ProcessingStatus.FAILED,
            created_at=datetime.now(),
            message="Failed to start ingestion job",
            errors=[str(e)]
        )

@app.get("/ingest/status/{job_id}", response_model=IngestionStatus)
async def get_job_status(job_id: str):
    """
    Get the status of an ingestion job
    """
    if job_id not in jobs_store:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Job with ID {job_id} not found"
        )

    job_data = jobs_store[job_id]
    return IngestionStatus(**job_data)

@app.get("/ingest/result/{job_id}")
async def get_job_result(
    job_id: str,
    format: str = Query("turtle", description="Output format (turtle, xml, 
json-ld, networkx)")
):
    """
    Get the result of a completed ingestion job
    """
    if job_id not in jobs_store:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Job with ID {job_id} not found"
        )

    job_data = jobs_store[job_id]
    if job_data["status"] != ProcessingStatus.COMPLETED:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Job is not completed (current status: 
{job_data['status']})"
        )

    # In a real implementation, we would retrieve the stored graph
    # For demonstration, we'll return a sample result
    if format == "turtle":
        return {"content_type": "text/turtle", "data": "# Sample Turtle data"}
    elif format == "xml":
        return {"content_type": "application/rdf+xml", "data": "<rdf:RDF><!--
Sample RDF/XML --></rdf:RDF>"}
    elif format == "json-ld":
        return {"content_type": "application/ld+json", "data": [{"@context": 
"http://example.org/"}]}
    elif format == "networkx":
        return {"content_type": "application/json", "data": {"nodes": [], 
"links": []}}
    else:
```

```python
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Unsupported format: {format}"
        )

@app.post("/ingest/pipeline", response_model=Dict[str, Any])
async def create_pipeline(pipeline_config: Dict[str, Any] = Body(...)):
    """
    Create a reusable ingestion pipeline
    """
    pipeline_id = f"pipeline_{len([k for k in jobs_store.keys() if
k.startswith('pipeline_')]) + 1}"

    # Store pipeline configuration
    jobs_store[pipeline_id] = {
        "pipeline_id": pipeline_id,
        "config": pipeline_config,
        "created_at": datetime.now()
    }

    return {
        "pipeline_id": pipeline_id,
        "message": "Pipeline created successfully",
        "created_at": jobs_store[pipeline_id]["created_at"]
    }

@app.get("/ingest/pipelines", response_model=List[Dict[str, Any]])
async def list_pipelines():
    """
    List all available ingestion pipelines
    """
    pipelines = [
        {
            "pipeline_id": k,
            "created_at": v["created_at"]
        }
        for k, v in jobs_store.items()
        if k.startswith("pipeline_")
    ]

    return pipelines

@app.get("/ingest/mappings/examples", response_model=Dict[str, Any])
async def get_mapping_examples():
    """
    Get example mapping configurations for different source types
    """
    examples = {
        "csv_mapping": {
            "source_type": "csv",
            "target_namespace": "http://example.org/ontology/",
            "class_mappings": {
                "csv_row": "Person"
            },
            "property_mappings": [
                {
                    "source_field": "name",
                    "target_property": "hasName",
                    "data_type": "xsd:string"
                },
                {
                    "source_field": "age",
                    "target_property": "hasAge",
                    "data_type": "xsd:integer"
                },
                {
                    "source_field": "email",
                    "target_property": "hasEmail",
                    "transformation": "lowercase"
                }
            ]
        },
        "json_mapping": {
            "source_type": "json",
            "target_namespace": "http://example.org/ontology/",
            "class_mappings": {
                "json_object": "Product"
            },
            "property_mappings": [
                {
                    "source_field": "product_name",
                    "target_property": "name"
                },
                {
                    "source_field": "details.price",
                    "target_property": "price",
                    "data_type": "xsd:decimal"
                },
                {
                    "source_field": "category",
                    "target_property": "category"
                }
            ]
        },
        "xml_mapping": {
            "source_type": "xml",
            "target_namespace": "http://example.org/ontology/",
            "class_mappings": {
                "root": "Order"
            },
            "property_mappings": [
                {
                    "source_field": "@id",
                    "target_property": "orderId"
                },
                {
                    "source_field": "customer/name",
                    "target_property": "customerName"
                },
                {
                    "source_field": "items/item/price",
                    "target_property": "totalPrice",
                    "transformation": "float"
                }
            ]
        }
    }

    return examples
def get_domain_ontology_schema():
    """
    Get the domain-specific ontology schema incorporating support, adaptive, 
and core objects
    """
    domain_schema = {
        "namespace": "http://example.org/domain-ontology/",
        "classes": [
            # Support Objects
            {
                "id": "SupportObject",
                "label": "Support Object",
                "description": "Objects that provide support for core 
operations",
                "properties": ["hasName", "hasDescription", "supportsObject"]
            },
            {
                "id": "BusinessStrategy",
                "label": "Business Strategy",
                "description": "Strategic business planning and approach",
                "subClassOf": "SupportObject",
                "properties": ["hasObjective", "hasMeasure", "hasTimeframe"]
            },
            {
                "id": "ArchitecturalStability",
                "label": "Architectural Stability",
                "description": "Components ensuring stability of
architecture",
                "subClassOf": "SupportObject",
                "properties": ["hasResilience", "hasRedundancy",
"hasScalability"]
            },
            {
                "id": "SocialNetwork",
                "label": "Social Network",
                "description": "Network of social relationships and
connections",
                "subClassOf": "SupportObject",
                "properties": ["hasConnections", "hasNodes", "hasInfluencers"]
            },

            # Adaptive Objects
            {
                "id": "AdaptiveObject",
                "label": "Adaptive Object",
                "description": "Objects that adapt to changes in the
environment",
                "properties": ["hasName", "hasDescription", "adaptsTo"]
            },
            {
                "id": "FeedbackLoop",
                "label": "Feedback Loop",
                "description": "System for providing feedback for adaptation",
                "subClassOf": "AdaptiveObject",
                "properties": ["hasInput", "hasOutput", "hasCycle"]
            },
            {
                "id": "MachineLearning",
                "label": "Machine Learning",
                "description": "Systems that learn from data",
                "subClassOf": "AdaptiveObject",
                "properties": ["hasAlgorithm", "hasTrainingData",
"hasAccuracy"]
            },
            {
                "id": "ThreatIntelligence",
                "label": "Threat Intelligence",
                "description": "Information about potential threats",
                "subClassOf": "AdaptiveObject",
                "properties": ["hasSource", "hasSeverity", "hasTimeliness"]
            },

            # Core Objects
            {
                "id": "CoreObject",
                "label": "Core Object",
                "description": "Essential objects central to the system",
                "properties": ["hasName", "hasDescription", "isEssential"]
            },
            {
                "id": "DesignedStructure",
                "label": "Designed Structure",
                "description": "Deliberately designed structural components",
                "subClassOf": "CoreObject",
                "properties": ["hasDesigner", "hasBlueprint", "hasFunction"]
            },
            {
                "id": "HumanEndeavor",
                "label": "Human Endeavor",
                "description": "Activities undertaken by humans",
                "subClassOf": "CoreObject",
                "properties": ["hasParticipants", "hasGoal", "hasDuration"]
            },
            {
                "id": "SupportedInformation",
                "label": "Supported Information",
                "description": "Information with supporting evidence",
                "subClassOf": "CoreObject",
                "properties": ["hasSource", "hasEvidence", "hasConfidence"]
            },
            {
                "id": "Agent",
                "label": "Agent",
                "description": "Entity capable of action",
                "subClassOf": "CoreObject",
                "properties": ["hasCapability", "hasIntention", "hasAutonomy"]
            },
            {
                "id": "Capability",
                "label": "Capability",
                "description": "Ability to perform certain functions",
                "subClassOf": "CoreObject",
                "properties": ["hasFunction", "hasPrerequisite",
"hasEffectiveness"]
            }
        ],
        "properties": [
            # Common properties
            {
                "id": "hasName",
                "label": "has name",
                "domain": ["SupportObject", "AdaptiveObject", "CoreObject"],
                "range": "xsd:string"
            },
            {
                "id": "hasDescription",
                "label": "has description",
                "domain": ["SupportObject", "AdaptiveObject", "CoreObject"],
                "range": "xsd:string"
            },

            # Support object properties
            {
                "id": "supportsObject",
                "label": "supports object",
                "domain": ["SupportObject"],
                "range": ["CoreObject", "AdaptiveObject"]
            },
            {
                "id": "hasObjective",
                "label": "has objective",
                "domain": ["BusinessStrategy"],
                "range": "xsd:string"
            },
            {
                "id": "hasMeasure",
                "label": "has measure",
                "domain": ["BusinessStrategy"],
                "range": "xsd:string"
            },
            {
                "id": "hasTimeframe",
                "label": "has timeframe",
                "domain": ["BusinessStrategy"],
                "range": "xsd:string"
            },
            {
                "id": "hasResilience",
                "label": "has resilience",
                "domain": ["ArchitecturalStability"],
                "range": "xsd:decimal"
            },
            {
                "id": "hasRedundancy",
                "label": "has redundancy",
                "domain": ["ArchitecturalStability"],
                "range": "xsd:string"
            },
            {
                "id": "hasScalability",
                "label": "has scalability",
                "domain": ["ArchitecturalStability"],
                "range": "xsd:string"
            },
            {
                "id": "hasConnections",
                "label": "has connections",
                "domain": ["SocialNetwork"],
                "range": "xsd:integer"
            },
            {
                "id": "hasNodes",
                "label": "has nodes",
                "domain": ["SocialNetwork"],
                "range": "xsd:integer"
            },
            {
                "id": "hasInfluencers",
                "label": "has influencers",
                "domain": ["SocialNetwork"],
                "range": "xsd:integer"
            },

            # Adaptive object properties
            {
                "id": "adaptsTo",
                "label": "adapts to",
                "domain": ["AdaptiveObject"],
                "range": "xsd:string"
            },
            {
                "id": "hasInput",
                "label": "has input",
                "domain": ["FeedbackLoop"],
                "range": "xsd:string"
            },
            {
                "id": "hasOutput",
                "label": "has output",
                "domain": ["FeedbackLoop"],
                "range": "xsd:string"
            },
            {
                "id": "hasCycle",
                "label": "has cycle",
                "domain": ["FeedbackLoop"],
                "range": "xsd:string"
```

```
        },
        {
            "id": "hasAlgorithm",
            "label": "has algorithm",
            "domain": ["MachineLearning"],
            "range": "xsd:string"
        },
        {
            "id": "hasTrainingData",
            "label": "has training data",
            "domain": ["MachineLearning"],
            "range": "xsd:string"
        },
        {
            "id": "hasAccuracy",
            "label": "has accuracy",
            "domain": ["MachineLearning"],
            "range": "xsd:decimal"
        },
        {
            "id": "hasSeverity",
            "label": "has severity",
            "domain": ["ThreatIntelligence"],
            "range": "xsd:string"
        },
        {
            "id": "hasTimeliness",
            "label": "has timeliness",
            "domain": ["ThreatIntelligence"],
            "range": "xsd:string"
        },

        # Core object properties
        {
            "id": "isEssential",
            "label": "is essential",
            "domain": ["CoreObject"],
            "range": "xsd:boolean"
        },
        {
            "id": "hasDesigner",
            "label": "has designer",
            "domain": ["DesignedStructure"],
            "range": "xsd:string"
        },
        {
            "id": "hasBlueprint",
            "label": "has blueprint",
            "domain": ["DesignedStructure"],
            "range": "xsd:string"
        },
        {
            "id": "hasFunction",
            "label": "has function",
            "domain": ["DesignedStructure", "Capability"],
            "range": "xsd:string"
        },
        {
            "id": "hasParticipants",
            "label": "has participants",
            "domain": ["HumanEndeavor"],
            "range": "xsd:integer"
        },
        {
            "id": "hasGoal",
            "label": "has goal",
            "domain": ["HumanEndeavor"],
            "range": "xsd:string"
        },
        {
            "id": "hasDuration",
            "label": "has duration",
            "domain": ["HumanEndeavor"],
            "range": "xsd:duration"
        },
        {
            "id": "hasSource",
            "label": "has source",
            "domain": ["SupportedInformation", "ThreatIntelligence"],
            "range": "xsd:string"
        },
        {
            "id": "hasEvidence",
            "label": "has evidence",
            "domain": ["SupportedInformation"],
            "range": "xsd:string"
        },
        {
            "id": "hasConfidence",
            "label": "has confidence",
            "domain": ["SupportedInformation"],
            "range": "xsd:decimal"
        },
        {
            "id": "hasCapability",
            "label": "has capability",
```

```json
                "domain": ["Agent"],
                "range": "Capability"
            },
            {
                "id": "hasIntention",
                "label": "has intention",
                "domain": ["Agent"],
                "range": "xsd:string"
            },
            {
                "id": "hasAutonomy",
                "label": "has autonomy",
                "domain": ["Agent"],
                "range": "xsd:decimal"
            },
            {
                "id": "hasPrerequisite",
                "label": "has prerequisite",
                "domain": ["Capability"],
                "range": "xsd:string"
            },
            {
                "id": "hasEffectiveness",
                "label": "has effectiveness",
                "domain": ["Capability"],
                "range": "xsd:decimal"
            }
        ],
        "relationships": [
            {
                "id": "supports",
                "label": "supports",
                "domain": "SupportObject",
                "range": ["CoreObject", "AdaptiveObject"]
            },
            {
                "id": "adaptsTo",
                "label": "adapts to",
                "domain": "AdaptiveObject",
                "range": ["CoreObject", "SupportObject"]
            },
            {
                "id": "requires",
                "label": "requires",
                "domain": "CoreObject",
                "range": ["SupportObject", "AdaptiveObject"]
            },
            {
                "id": "interactsWith",
                "label": "interacts with",
                "domain": ["CoreObject", "AdaptiveObject", "SupportObject"],
                "range": ["CoreObject", "AdaptiveObject", "SupportObject"]
            }
        ]
    }

    return domain_schema
# Example ontology schema endpoint for reference
@app.get("/ingest/schema/example", response_model=Dict[str, Any])
async def get_example_schema():
    """
    Get an example ontology schema
    """
    example_schema = {
        "namespace": "http://example.org/ontology/",
        "classes": [
            {
                "id": "Person",
                "label": "Person",
                "description": "An individual person",
                "properties": ["hasName", "hasAge", "hasEmail"]
            },
            {
                "id": "Organization",
                "label": "Organization",
                "description": "A group, institution or organization",
                "properties": ["hasName", "hasLocation"]
            },
            {
                "id": "Product",
                "label": "Product",
                "description": "A commercial product",
                "properties": ["name", "price", "category"]
            }
        ],
        "properties": [
            {
                "id": "hasName",
                "label": "has name",
                "domain": ["Person", "Organization"],
                "range": "xsd:string"
            },
            {
                "id": "hasAge",
                "label": "has age",
                "domain": ["Person"],
                "range": "xsd:integer"
            },
            {
                "id": "hasEmail",
                "label": "has email",
                "domain": ["Person"],
                "range": "xsd:string"
            },
            {
                "id": "hasLocation",
                "label": "has location",
                "domain": ["Organization"],
                "range": "xsd:string"
            },
            {
                "id": "name",
                "label": "name",
                "domain": ["Product"],
                "range": "xsd:string"
            },
            {
                "id": "price",
                "label": "price",
                "domain": ["Product"],
                "range": "xsd:decimal"
            },
            {
                "id": "category",
                "label": "category",
                "domain": ["Product"],
                "range": "xsd:string"
            }
        ]
    }

    return example_schema

# Main entry point
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```