### **Project Folder Structure**

perspective\_api/  
├── api/  
│ └── v1/  
│ └── endpoints/  
│ └── perspective.py  
├── database/  
│ └── database.py  
├── models/  
│ └── perspective.py  
├── schemas/  
│ └── perspective.py  
├── services/  
│ └── perspective.py  
└── main.py

### main.py

import os  
from flask import Flask, g  
from .api.v1.endpoints.perspective import perspective\_bp  
from .database.database import close\_db\_connection  
  
# Initialize the Flask application  
app = Flask(\_\_name\_\_)  
  
# Register the blueprint for the API routes  
app.register\_blueprint(perspective\_bp, url\_prefix='/api/v1')  
  
# Add a teardown function to close the database connection and cursor  
@app.teardown\_appcontext  
def teardown\_db(exception=None):  
 conn = g.pop('db\_conn', None)  
 curr = g.pop('db\_curr', None)  
 close\_db\_connection(conn, curr)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

### database/database.py

import os  
import psycopg2  
from psycopg2.extras import DictCursor, register\_uuid  
from psycopg2.extensions import connection, cursor  
from flask import g  
from typing import Tuple  
  
# Register UUID support for psycopg2  
register\_uuid()  
  
# Database connection details (replace with your actual details or environment variables).  
DB\_NAME = "skg023"  
DB\_USER = "your\_db\_user"  
DB\_PASSWORD = "your\_db\_password"  
DB\_HOST = "localhost"  
DB\_PORT = "5432"  
DB\_SCHEMA = "recsui"  
  
def get\_db\_connection() -> Tuple[connection, cursor]:  
 """  
 Establishes a connection to the PostgreSQL database and returns  
 both the connection and a cursor object.  
  
 Raises:  
 psycopg2.Error: If the connection fails.  
 """  
 try:  
 conn = psycopg2.connect(  
 dbname=DB\_NAME,  
 user=DB\_USER,  
 password=DB\_PASSWORD,  
 host=DB\_HOST,  
 port=DB\_PORT,  
 )  
 # Use DictCursor to return query results as dictionaries  
 curr = conn.cursor(cursor\_factory=DictCursor)  
 curr.execute(f"SET search\_path TO {DB\_SCHEMA};")  
 print("Successfully connected to the database.")  
 return conn, curr  
 except psycopg2.Error as e:  
 print(f"Error connecting to the database: {e}")  
 raise  
  
def close\_db\_connection(conn: connection, curr: cursor):  
 """Closes the database cursor and connection."""  
 if curr:  
 curr.close()  
 if conn:  
 conn.close()  
 print("Database connection closed.")  
  
def get\_db():  
 """  
 Provides a database connection and cursor that is local to the current request.  
 If they don't exist for the request, it creates new ones.  
 """  
 if 'db\_conn' not in g or 'db\_curr' not in g:  
 try:  
 conn, curr = get\_db\_connection()  
 g.db\_conn = conn  
 g.db\_curr = curr  
 except psycopg2.Error as e:  
 raise ConnectionError(f"Failed to get database connection: {e}") from e  
   
 return g.db\_conn, g.db\_curr

### models/perspective.py

from typing import Optional, List, Dict, Any  
from datetime import datetime  
  
class ColumnState:  
 """Represents the structure of a single item in the column\_state array."""  
 def \_\_init\_\_(self, name: str, view: str, defaultColumns: List[str], default: bool):  
 self.name = name  
 self.view = view  
 self.defaultColumns = defaultColumns  
 self.default = default  
  
class FilterDetail:  
 """Represents the structure of a filter detail."""  
 def \_\_init\_\_(self, type: str, filter: str):  
 self.type = type  
 self.filter = filter  
  
class ViewSetting:  
 """Represents the structure of a single item in the sort\_model or filter\_model arrays."""  
 def \_\_init\_\_(self, name: str, view: str, filters: Dict[str, FilterDetail], default: bool):  
 self.name = name  
 self.view = view  
 self.filters = filters  
 self.default = default  
  
class Perspective:  
 """  
 Data Transfer Object (DTO) for the 'perspectives' table.  
 This class is not tied to an ORM and is used to represent a database row.  
 """  
 def \_\_init\_\_(self, id: int, username: str, layout\_name: str, updated\_by: str,  
 column\_state: List[ColumnState], sort\_model: List[ViewSetting],  
 filter\_model: List[ViewSetting], updated\_time: datetime):  
 self.id = id  
 self.username = username  
 self.layout\_name = layout\_name  
 self.updated\_by = updated\_by  
 self.column\_state = column\_state  
 self.sort\_model = sort\_model  
 self.filter\_model = filter\_model  
 self.updated\_time = updated\_time  
  
 @staticmethod  
 def from\_dict(data: dict):  
 """Converts a dictionary (from a psycopg2 query) to a Perspective object."""  
 if not data:  
 return None  
   
 # Recursively create objects for nested data  
 column\_state = [ColumnState(\*\*cs) for cs in data.get('column\_state', [])]  
   
 sort\_model = []  
 for sm in data.get('sort\_model', []):  
 filters = {k: FilterDetail(\*\*v) for k, v in sm.get('filters', {}).items()}  
 sort\_model.append(ViewSetting(name=sm['name'], view=sm['view'], filters=filters, default=sm['default']))  
  
 filter\_model = []  
 for fm in data.get('filter\_model', []):  
 filters = {k: FilterDetail(\*\*v) for k, v in fm.get('filters', {}).items()}  
 filter\_model.append(ViewSetting(name=fm['name'], view=fm['view'], filters=filters, default=fm['default']))  
   
 return Perspective(  
 id=data['id'],  
 username=data['username'],  
 layout\_name=data['layout\_name'],  
 updated\_by=data['updated\_by'],  
 column\_state=column\_state,  
 sort\_model=sort\_model,  
 filter\_model=filter\_model,  
 updated\_time=data['updated\_time']  
 )

### schemas/perspective.py

from datetime import datetime  
from typing import Optional, List, Dict, Any  
from pydantic import BaseModel, Field, model\_serializer  
  
# Schema for the nested filter details within sort and filter models  
class FilterDetail(BaseModel):  
 type: str  
 filter: str  
  
# Schema for the column\_state data structure  
class ColumnState(BaseModel):  
 name: str  
 view: str  
 defaultColumns: List[str]  
 default: bool  
  
# Schema for the sort\_model and filter\_model data structures  
class ViewSetting(BaseModel):  
 name: str  
 view: str  
 filters: Dict[str, FilterDetail]  
 default: bool  
  
# Base schema for creating or updating a perspective.  
# All fields are validated to be non-empty strings as requested.  
class PerspectiveBase(BaseModel):  
 username: str = Field(..., min\_length=1, description="Username cannot be empty")  
 layout\_name: str = Field(..., min\_length=1, description="Layout name cannot be empty")  
 updated\_by: str = Field(..., min\_length=1, description="Updater's email cannot be empty")  
 column\_state: Optional[List[ColumnState]] = []  
 sort\_model: Optional[List[ViewSetting]] = []  
 filter\_model: Optional[List[ViewSetting]] = []  
  
# Schema for creating a new perspective (inherits from PerspectiveBase)  
class PerspectiveCreate(PerspectiveBase):  
 pass  
  
# Schema for updating an existing perspective.  
# All fields are optional for partial updates.  
class PerspectiveUpdate(PerspectiveBase):  
 username: Optional[str] = Field(None, min\_length=1, description="Username cannot be empty")  
 layout\_name: Optional[str] = Field(None, min\_length=1, description="Layout name cannot be empty")  
 updated\_by: Optional[str] = Field(None, min\_length=1, description="Updater's email cannot be empty")  
 column\_state: Optional[List[ColumnState]] = None  
 sort\_model: Optional[List[ViewSetting]] = None  
 filter\_model: Optional[List[ViewSetting]] = None  
  
# Full schema for a perspective, including auto-generated fields.  
class Perspective(PerspectiveBase):  
 id: int  
 updated\_time: str  
  
 class Config:  
 from\_attributes = True

### services/perspective.py

import psycopg2  
from psycopg2.extras import DictCursor  
from typing import List, Optional  
import json  
from ..models.perspective import Perspective as PerspectiveModel  
from ..schemas.perspective import PerspectiveCreate, PerspectiveUpdate  
from psycopg2.extensions import connection, cursor  
  
class PerspectiveService:  
 """Service class for performing CRUD operations on Perspective data using psycopg2."""  
  
 def \_\_init\_\_(self, db\_conn: connection, db\_curr: cursor):  
 self.db\_conn = db\_conn  
 self.db\_curr = db\_curr  
  
 def get\_all\_perspectives(self) -> List[PerspectiveModel]:  
 """Retrieves all perspective records from the database."""  
 self.db\_curr.execute("SELECT \* FROM recsui.perspectives;")  
 perspectives = self.db\_curr.fetchall()  
 return [PerspectiveModel.from\_dict(p) for p in perspectives]  
  
 def get\_perspective\_by\_id(self, perspective\_id: int) -> Optional[PerspectiveModel]:  
 """Retrieves a single perspective record by its ID."""  
 self.db\_curr.execute("SELECT \* FROM recsui.perspectives WHERE id = %s;", (perspective\_id,))  
 perspective = self.db\_curr.fetchone()  
 return PerspectiveModel.from\_dict(perspective)  
  
 def create\_perspective(self, perspective\_in: PerspectiveCreate) -> PerspectiveModel:  
 """Creates a new perspective record in the database."""  
 # Convert Pydantic models to JSON strings for database insertion  
 column\_state\_json = json.dumps([cs.model\_dump() for cs in perspective\_in.column\_state])  
 sort\_model\_json = json.dumps([sm.model\_dump() for sm in perspective\_in.sort\_model])  
 filter\_model\_json = json.dumps([fm.model\_dump() for fm in perspective\_in.filter\_model])  
   
 try:  
 self.db\_curr.execute(  
 """  
 INSERT INTO recsui.perspectives (username, layout\_name, updated\_by, column\_state, sort\_model, filter\_model)  
 VALUES (%s, %s, %s, %s, %s, %s)  
 RETURNING \*;  
 """,  
 (  
 perspective\_in.username,  
 perspective\_in.layout\_name,  
 perspective\_in.updated\_by,  
 column\_state\_json,  
 sort\_model\_json,  
 filter\_model\_json  
 )  
 )  
 new\_perspective = self.db\_curr.fetchone()  
 self.db\_conn.commit()  
 return PerspectiveModel.from\_dict(new\_perspective)  
 except Exception as e:  
 self.db\_conn.rollback()  
 raise e  
  
 def update\_perspective(self, perspective\_id: int, perspective\_in: PerspectiveUpdate) -> Optional[PerspectiveModel]:  
 """Updates an existing perspective record."""  
 # Check if the perspective exists  
 perspective\_to\_update = self.get\_perspective\_by\_id(perspective\_id)  
 if not perspective\_to\_update:  
 return None  
   
 # Build the update query dynamically  
 update\_clauses = []  
 update\_data = []  
   
 # Convert Pydantic models to JSON strings for database update  
 perspective\_dict = perspective\_in.model\_dump(exclude\_unset=True)  
   
 if 'username' in perspective\_dict:  
 update\_clauses.append("username = %s")  
 update\_data.append(perspective\_dict['username'])  
 if 'layout\_name' in perspective\_dict:  
 update\_clauses.append("layout\_name = %s")  
 update\_data.append(perspective\_dict['layout\_name'])  
 if 'updated\_by' in perspective\_dict:  
 update\_clauses.append("updated\_by = %s")  
 update\_data.append(perspective\_dict['updated\_by'])  
 if 'column\_state' in perspective\_dict:  
 update\_clauses.append("column\_state = %s")  
 update\_data.append(json.dumps([cs.model\_dump() for cs in perspective\_in.column\_state]))  
 if 'sort\_model' in perspective\_dict:  
 update\_clauses.append("sort\_model = %s")  
 update\_data.append(json.dumps([sm.model\_dump() for sm in perspective\_in.sort\_model]))  
 if 'filter\_model' in perspective\_dict:  
 update\_clauses.append("filter\_model = %s")  
 update\_data.append(json.dumps([fm.model\_dump() for fm in perspective\_in.filter\_model]))  
  
 if not update\_clauses:  
 return perspective\_to\_update # No changes to apply  
  
 query = f"UPDATE recsui.perspectives SET {', '.join(update\_clauses)} WHERE id = %s RETURNING \*;"  
 update\_data.append(perspective\_id)  
   
 try:  
 self.db\_curr.execute(query, tuple(update\_data))  
 updated\_perspective = self.db\_curr.fetchone()  
 self.db\_conn.commit()  
 return PerspectiveModel.from\_dict(updated\_perspective)  
 except Exception as e:  
 self.db\_conn.rollback()  
 raise e  
  
 def delete\_perspective(self, perspective\_id: int) -> bool:  
 """Deletes a perspective record by its ID."""  
 try:  
 self.db\_curr.execute("DELETE FROM recsui.perspectives WHERE id = %s RETURNING id;", (perspective\_id,))  
 deleted\_row = self.db\_curr.fetchone()  
 if deleted\_row:  
 self.db\_conn.commit()  
 return True  
 else:  
 self.db\_conn.rollback()  
 return False  
 except Exception as e:  
 self.db\_conn.rollback()  
 raise e

### api/v1/endpoints/perspective.py

from flask import Blueprint, jsonify, request  
import psycopg2  
from pydantic import ValidationError  
from typing import List  
from ..database.database import get\_db  
from ..schemas.perspective import Perspective, PerspectiveCreate, PerspectiveUpdate  
from ..services.perspective import PerspectiveService  
  
# Create a Blueprint for this module  
perspective\_bp = Blueprint('perspective\_bp', \_\_name\_\_)  
  
@perspective\_bp.route("/perspectives/", methods=["GET"])  
def get\_all():  
 """Retrieves all perspectives from the database."""  
 conn, curr = get\_db()  
 try:  
 service = PerspectiveService(conn, curr)  
 perspectives = service.get\_all\_perspectives()  
 # The serializer in the Perspective schema now expects a dictionary-like object,  
 # which our DTO provides.  
 return jsonify([Perspective(\*\*p.\_\_dict\_\_).model\_dump() for p in perspectives])  
 except Exception as e:  
 return jsonify({"detail": str(e)}), 500  
  
@perspective\_bp.route("/perspectives/<int:perspective\_id>", methods=["GET"])  
def get\_by\_id(perspective\_id: int):  
 """Retrieves a single perspective by its ID."""  
 conn, curr = get\_db()  
 try:  
 service = PerspectiveService(conn, curr)  
 perspective = service.get\_perspective\_by\_id(perspective\_id)  
 if not perspective:  
 return jsonify({"detail": "Perspective not found"}), 404  
 return jsonify(Perspective(\*\*perspective.\_\_dict\_\_).model\_dump())  
 except Exception as e:  
 return jsonify({"detail": str(e)}), 500  
  
@perspective\_bp.route("/perspectives/", methods=["POST"])  
def create():  
 """Creates a new perspective in the database."""  
 conn, curr = get\_db()  
 try:  
 perspective\_in = PerspectiveCreate(\*\*request.json)  
 except ValidationError as e:  
 return jsonify({"detail": e.errors()}), 422  
   
 try:  
 service = PerspectiveService(conn, curr)  
 new\_perspective = service.create\_perspective(perspective\_in)  
 return jsonify(Perspective(\*\*new\_perspective.\_\_dict\_\_).model\_dump()), 201  
 except Exception as e:  
 return jsonify({"detail": str(e)}), 500  
  
@perspective\_bp.route("/perspectives/<int:perspective\_id>", methods=["PUT"])  
def update(perspective\_id: int):  
 """Updates an existing perspective by its ID."""  
 conn, curr = get\_db()  
 try:  
 perspective\_in = PerspectiveUpdate(\*\*request.json)  
 except ValidationError as e:  
 return jsonify({"detail": e.errors()}), 422  
   
 try:  
 service = PerspectiveService(conn, curr)  
 updated\_perspective = service.update\_perspective(perspective\_id, perspective\_in)  
 if not updated\_perspective:  
 return jsonify({"detail": "Perspective not found"}), 404  
 return jsonify(Perspective(\*\*updated\_perspective.\_\_dict\_\_).model\_dump())  
 except Exception as e:  
 return jsonify({"detail": str(e)}), 500  
  
@perspective\_bp.route("/perspectives/<int:perspective\_id>", methods=["DELETE"])  
def delete(perspective\_id: int):  
 """Deletes a perspective by its ID."""  
 conn, curr = get\_db()  
 try:  
 service = PerspectiveService(conn, curr)  
 if not service.delete\_perspective(perspective\_id):  
 return jsonify({"detail": "Perspective not found"}), 404  
 return "", 204  
 except Exception as e:  
 return jsonify({"detail": str(e)}), 500