Reading: About the Product Model

As detailed in the Final Project Overview section, you must create a microservice for the Product Catalogue backend to an eCommerce Application.

Let us look at the file service/models.py to understand and analyze the Product Model, its various attributes, the imported modules, the Product Class, and the different methods used.

Estimated reading time: 10 minutes

Attributes

The different attributes used in this model are:

1. id - id of the product
2. name - name of the product
3. description - description of the product
4. price - price of the product
5. available - if the product is available
6. category - category under which the product belongs

Import Statements

1. **import logging** - You use Python's statement `import logging` to import the built-in logging module. Once you import the logging module, you can use its functions and classes to log messages at various levels of severity, such as DEBUG, INFO, WARNING, ERROR, and CRITICAL.

2. **from enum import Enum** - You use the statement `from enum import Enum` in Python to import the Enum class from the enum module. Enums, or enumerations, are a way to define a set of named values in Python. By defining your enums, you can create a set of symbolic names representing a fixed set of values.

3. **from decimal import Decimal** - You use the statement `from decimal import Decimal` in Python to import the Decimal class from the decimal module. The Decimal class supports decimal floating-point arithmetic and is particularly useful when performing precise decimal calculations.

4. **from flask import Flask** - You use the statement `from flask import Flask` in Python to import the Flask class from the flask module. Flask is a popular web framework in Python that allows you to build web applications. The Flask class is the core component of the Flask framework and is responsible for creating a Flask application instance.

5. **from flask_sqlalchemy import SQLAlchemy** - You use the statement `from flask_sqlalchemy import SQLAlchemy` in Python to import the SQLAlchemy class from the flask_sqlalchemy module. Flask SQLAlchemy is an extension for the Flask web framework that integrates with SQLAlchemy, a popular Object-Relational Mapping (ORM) library in Python.

Methods & Classes

1. **db = SQLAlchemy()** - This statement creates a SQLAlchemy object associated with a Flask application instance to establish a database connection and integrate SQLAlchemy with the Flask application.
   The below code initializes the database.

```
def init_db(app):
    """Initialize the SQLAlchemy app"""
    Product.init_db(app)
```

2. **class DataValidationError(Exception)**
   This statement defines a custom exception class called `DataValidationError` that inherits from the built-in Exception class. By creating custom exception classes, you can define specific error conditions or exceptional scenarios in your code and raise those exceptions when necessary. By using this custom exception, when you do data validation, it raises a `DataValidationError` exception with an appropriate error message if the data is not valid.

3. **Enumeration of Product Categories**
   The below code snippet defines the Category enumeration with additional categories and assigns explicit integer values to each member. Each member represents a valid product category.

```
class Category(Enum):
    """Enumeration of valid Product Categories"""
    UNKNOWN = 0
    CLOTHS = 1
    FOOD = 2
    HOUSEWARES = 3
    AUTOMOTIVE = 4
    TOOLS = 5
```

4. **Product Class**
   The code snippet below defines a Product model class using db-model as the base class. The Product model represents a table in the database, and you define each table column as a class attribute.
   The column definitions in the Product model are detailed below:

- id: An integer column representing the primary key of the Product table.
- name: A string column with a maximum length of 100 characters, representing the product's name. `nullable=False` indicates that the corresponding column in the database cannot have a NULL value.
- description: A string column with a maximum length of 250 characters representing the product's description. `nullable=False` indicates that the corresponding column in the database cannot have a NULL value.
- price: A numeric column representing the price of the product. You use the `db.Numeric` type for precise decimal calculations. `nullable=False` indicates that the corresponding column in the database cannot have a NULL value.
- available: A boolean column representing the availability of the product.
  It has a default value of True. `nullable=False` indicates that the corresponding column in the database cannot have a NULL value.
- category: An enum column representing the category of the product. It uses the Category enum defined previously. You use the `db.Enum` type to map the enum values to the corresponding database values. The `server_default` argument is set to Category.UNKNOWN.name, which provides a default value for the column.

By defining the Product model, you can interact with the corresponding table in the database using SQLAlchemy's ORM features. This interaction allows you to perform CRUD (Create, Read, Update, Delete) operations on Product objects and query the database using the defined columns.

```
class Product(db.Model):
    """
    Class that represents a Product
    This version uses a relational database for a hidden persistence
    from us by SQLAlchemy's object-relational mappings (ORM)
    """
    ##################################################
    # Table Schema
    ##################################################
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    description = db.Column(db.String(250), nullable=False)
    price = db.Column(db.Numeric, nullable=False)
    available = db.Column(db.Boolean(), nullable=False, default=True)
    category = db.Column(
        db.Enum(Category), nullable=False, server_default=(Category.UNKNOWN.name)
    )
```

Instance Methods

1. `__repr__()` method

The below code snippet shows the Product model in which you define `__repr__()` method to return a string representation of a Product object. The returned string includes the name and id of the product.

```
def __repr__(self):
        return f"<Product {self.name} id=[{self.id}]>"
```

2. `create()` method
   The code snippet below shows the `create()` method within the Product class responsible for creating a new Product object in the database.

```
def create(self):
        """
        Creates a Product to the database
        """
        logger.info("Creating %s", self.name)
        # id must be none to generate next primary key
        self.id = None  # pylint: disable=invalid-name
        db.session.add(self)
        db.session.commit()
```

3. `update()` method

The code snippet below shows the `update()` method within the Product class responsible for updating an existing Product object in the database.

```
def update(self):
        """
        Updates a Product to the database
        """
        logger.info("Saving %s", self.name)
        if not self.id:
            raise DataValidationError("Update called with empty ID field")
        db.session.commit()
```

Note: There is a check done on the id attribute of the Product object. If the id is empty, it raises a DataValidationError, indicating that the update is being called on a product with a blank ID field.

4. `delete()` method

The code snippet below shows the `delete()` method within the Product class responsible for deleting a Product object from the database.

```
def delete(self):
        """Removes a Product from the data store"""
        logger.info("Deleting %s", self.name)
        db.session.delete(self)
        db.session.commit()
```

5. `serialize()` method
   The below code snippet shows the `serialize()` method within the Product class that converts a Product object into a dictionary representation, which can be useful for various purposes such as JSON serialization, data transfer, or API responses.

```
 def serialize(self) -> dict:
        """Serializes a Product into a dictionary"""
        return {
            "id": self.id,
            "name": self.name,
            "description": self.description,
            "price": str(self.price),
            "available": self.available,
            "category": self.category.name  # convert enum to string
        }
```

6. `deserialize()` method

The below code snippet shows the `deserialize()` method within the Product class that allows you to populate a Product object with data from a dictionary representation. This deserialization process converts the structured data into an object with corresponding attributes.

```
def deserialize(self, data: dict):
        """
        Deserializes a Product from a dictionary
        Args:
            data (dict): A dictionary containing the Product data
        """
        try:
            self.name = data["name"]
            self.description = data["description"]
            self.price = Decimal(data["price"])
            if isinstance(data["available"], bool):
                self.available = data["available"]
            else:
                raise DataValidationError(
                    "Invalid type for boolean [available]: "
                    + str(type(data["available"]))
                )
            self.category = getattr(Category, data["category"])  # create enum from string
        except AttributeError as error:
            raise DataValidationError("Invalid attribute: " + error.args[0]) from error
        except KeyError as error:
            raise DataValidationError("Invalid product: missing " + error.args[0]) from error
        except TypeError as error:
            raise DataValidationError(
                "Invalid product: body of request contained bad or no data " + str(error)
            ) from error
        return self
```

Note: Various exceptions (AttributeError, KeyError, TypeError) are caught and re-raised as DataValidationError exceptions with appropriate error messages. These exceptions handle scenarios where the provided data dictionary is missing the required keys or contains invalid data.

Class Methods

1. `init_db()` method

The below code snippet includes a class method called `init_db` within the Product class, which is responsible for initializing the database session and creating the necessary SQLAlchemy tables.

```
@classmethod
    def init_db(cls, app: Flask):
        """Initializes the database session
        :param app: the Flask app
        :type data: Flask
        """
        logger.info("Initializing database")
        # This is where we initialize SQLAlchemy from the Flask app
        db.init_app(app)
        app.app_context().push()
        db.create_all()  # make our sqlalchemy tables
```

2. `all()` method

The below code snippet includes a class method called `all()` within the Product class which retrieves all the Product objects from the database.

```
@classmethod
    def all(cls) -> list:
        """Returns all of the Products in the database"""
        logger.info("Processing all Products")
        return cls.query.all()
```

3. `find()` method
   The below code snippet includes a class method called `find()` within the Product class, which is responsible for finding a Product by its ID in the database.

```
@classmethod
    def find(cls, product_id: int):
        """Finds a Product by it's ID
        :param product_id: the id of the Product to find
        :type product_id: int
        :return: an instance with the product_id, or None if not found
        :rtype: Product
        """
        logger.info("Processing lookup for id %s ...", product_id)
        return cls.query.get(product_id)
```

Note: If a Product object with the specified ID is found in the database, it is returned. Otherwise, it returns None.

`4. find_by_name() method`

The code snippet below includes a class method called `find_by_name()` within the Product class, which retrieves all the Product objects from the database with a matching name.

```
@classmethod
    def find_by_name(cls, name: str) -> list:
        """Returns all Products with the given name
```

```
        :param name: the name of the Products you want to match
        :type name: str
        :return: a collection of Products with that name
        :rtype: list
        """
        logger.info("Processing name query for %s ...", name)
        return cls.query.filter(cls.name == name)
```

Note: By calling this method on the Product class and providing a valid name, you will get a collection of Product objects matching that name.

5. `find_by_price()` method
   The code snippet below includes a class method called `find_by_price()` within the Product class, which retrieves all the Product objects from the database with a matching price.

```
@classmethod
    def find_by_price(cls, price: Decimal) -> list:
        """Returns all Products with the given price
        :param price: the price to search for
        :type name: float
        :return: a collection of Products with that price
        :rtype: list
        """
        logger.info("Processing price query for %s ...", price)
        price_value = price
        if isinstance(price, str):
            price_value = Decimal(price.strip(' "'))
        return cls.query.filter(cls.price == price_value)
```

Note: By calling this method on the Product class and providing a valid price, you will get a collection of Product objects matching that price.

6. `find_by_availability()` method
   The code snippet below includes a class method called `find_by_availability()` within the Product class, which retrieves all the Product objects from the database based on availability.

```
@classmethod
    def find_by_availability(cls, available: bool = True) -> list:
        """Returns all Products by their availability
        :param available: True for available products
        :type available: str
        :return: a collection of available Products
        :rtype: list
        """
        logger.info("Processing available query for %s ...", available)
        return cls.query.filter(cls.available == available)
```

Note: By calling this method on the Product class and providing a valid availability value (defaulting to True if not provided), you will get a collection of Product objects that match the specified availability.

7. `find_by_category()` method
   The below code snippet includes a class method called `find_by_category()` within the Product class, which retrieves all the Product objects from the database based on their category.

```
@classmethod
    def find_by_category(cls, category: Category = Category.UNKNOWN) -> list:
        """Returns all Products by their Category
        :param category: values are ['MALE', 'FEMALE', 'UNKNOWN']
        :type available: enum
        :return: a collection of available Products
        :rtype: list
        """
        logger.info ("Processing category query for %s ...", category.name)
        return cls.query.filter(cls.category == category)
```

Note: By calling this method on the Product class and providing a valid category value (defaulting to Category. UNKNOWN if not provided), you will get a collection of Product objects that match the specified category.

Author(s)

- Anita Narain