Trade Surveillance Project.

Requirement:

Develop an application by using the Python, PySpark and FastAPI and it will have two parts.

Data processing part and REST API part.

It will be like below-

First Load csv data from a network storage and load into PySpark dataframes, then we will process/filter/join this data as per the given business logic and at the end we will have the processed data.

Then use this data to prepare JSON files and insert this JSON data into PostgreSQL.

Then our FastAPI code will read this JSON data from PostgreSQL and then our business users can consume these API to load that processed data into some kind of dashboard.

Identify:

- Outline for this application.
- Software Architecture style.
- Design strategy for this Application
- Required Tools and Technologies.
- Classification for different Module.
- Edge cases.

Data Pipeline and API Integration Architecture with clear separation between data processing and API services is appropriate. Here's a suggested outline and architecture for your application:

1. **Software Architecture Style**

- **Microservices Architecture**: Since you have distinct parts for data processing and REST API, a microservices or modular architecture is suitable. Each component can scale independently and provides fault tolerance if you need to restart or update the pipeline or the API service separately.

- **Batch Data Processing Pipeline**: For data extraction, transformation, and loading (ETL) with PySpark, a batch processing pipeline is a good choice, given your need for CSV processing and large-scale data transformation.

2. **High-Level Design Strategy**

- Data Processing Layer: Handles all the CSV ingestion, transformation, and processing in PySpark.

- Data Persistence Layer: Stores the processed data in JSON format within PostgreSQL, with JSONB fields for efficient querying.

- API Layer: FastAPI serves processed data via REST endpoints, allowing the dashboard to request specific data as needed.

3. **Tools and Technologies**

- **Python**: Main programming language for PySpark and FastAPI integration.

- **PySpark**: For distributed data processing, ideal for large-scale data transformations, joins, and filters.

- **FastAPI**: REST API framework for Python, lightweight and fast, and can serve data from PostgreSQL.

- **PostgreSQL**: Database with native JSON support to store processed JSON data.

- **Docker/Kubernetes** (optional): For containerization and deployment, allowing easy scaling and management of both the data pipeline and API services.

- **Network File Storage** (NFS or Cloud Storage): Accessible network storage to fetch CSV data for processing.

4. **Modules and Functionality**

- **Data Ingestion Module**:

  - Functionality: Loads CSV data from network storage, converts it into PySpark dataframes, and handles schema validation and error handling.

  - Edge Cases: Missing or malformed files, corrupted rows, and schema mismatches.

- **Data Processing Module**:

  - Functionality: Filters, joins, and aggregates data according to business rules, handling any transformations needed to prepare data for consumption.

  - Edge Cases: Handling null values, inconsistent data types, large dataset memory management, and optimization for join operations.

- **Data Serialization Module**:

  - Functionality: Converts PySpark dataframes to JSON format and writes them to the network storage for backup and to PostgreSQL.

  - Edge Cases: Handling large JSON objects, ensuring JSON serialization compatibility, and handling storage limitations.

- **Database Insertion Module**:

  - Functionality: Inserts processed JSON data into PostgreSQL in JSONB format to support flexible querying.

  - Edge Cases: Duplicate entries, data integrity issues, and transaction management.

- **API Service Module (FastAPI**):

  - Functionality: Serves processed data from PostgreSQL via RESTful endpoints, includes filtering and pagination for client requests.

  - Edge Cases: High-concurrency handling, query performance optimization, and ensuring API authentication/authorization.

- **Data Integrity and Quality Module**:

  - Functionality: Validates data at each stage (from ingestion to API output) and monitors for anomalies.

  - Edge Cases: Data mismatch between stages, data loss, and API downtime monitoring.

5. **Additional Suggestions**

- Logging and Monitoring:

  - Use tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Prometheus and Grafana to log and monitor each module's performance, which helps in troubleshooting and maintaining data pipeline and API uptime.

- Testing and Validation: Unit and integration tests are critical for the data pipeline (especially for PySpark transformations), API validation, and database I/O operations.

- Data Versioning: If data changes over time, version your data and APIs so that users can refer to different processed datasets.

- Data Security: Set up network access control, encrypt sensitive data, and implement API authentication (e.g., OAuth2 or API tokens) to secure data access.

6. **Workflow Example**

- Step 1: The Data Ingestion Module triggers, loading CSV files from network storage into PySpark.

- Step 2: The Data Processing Module transforms data according to business rules, outputting it as JSON files.

- Step 3: JSON data is written to PostgreSQL by the Database Insertion Module.

- Step 4: FastAPI fetches processed JSON data from PostgreSQL and exposes it to end-users through REST endpoints.

- Step 5: Monitoring tools keep track of all modules and send alerts if issues arise.

Required Design patterns

1. **Data Processing Layer (PySpark)**

- **Builder** Pattern: This pattern is useful for building complex PySpark transformations by chaining steps together. For example, you can create a transformation pipeline that applies multiple filters, joins, and aggregations on your data. A Builder can help configure the transformations step-by-step, making it easier to add or remove steps without changing the main logic.

- **Factory** Pattern: You may have different types of CSV files (e.g., sales data, inventory data) that require different transformations. Using a Factory pattern to generate the appropriate transformation pipeline based on the file type can make your code more flexible and easier to extend.

- **Strategy** Pattern: For data processing tasks that might require different approaches based on certain criteria (e.g., different join strategies or filtering methods), the Strategy pattern allows you to switch between algorithms without modifying the core processing logic. This pattern makes it easier to adapt to different business rules by encapsulating each strategy as a separate class.

2. **Data Serialization and Persistence Layer**

- **Repository** Pattern: This pattern is useful for abstracting the details of database interaction. A repository class can manage CRUD operations for PostgreSQL, handling JSON inserts, reads, and updates. This keeps your database code separate from the business logic, making it easier to test and swap out the database if needed.

- **Data Mapper** Pattern: Since you're storing JSON in PostgreSQL, Data Mapper can be helpful in mapping PySpark DataFrames to JSON structures and then into PostgreSQL. This keeps the transformation logic modular and independent from database-specific operations.

- **Adapter** Pattern: When transforming the data into JSON and inserting it into PostgreSQL, the Adapter pattern can standardize how different types of data (e.g., PySpark DataFrames) are converted into JSON format. This makes it easier to support different data structures if needed in the future.

3. **API Layer (FastAPI)**

- **Controller** Pattern: In the FastAPI code, a Controller pattern organizes API endpoints and handles incoming HTTP requests. Controllers help route requests to the correct service functions, keeping the API layer clean and focused on handling HTTP requests.

- **Dependency** Injection (DI): FastAPI has built-in DI capabilities, which allow you to inject dependencies (e.g., database connections, configurations) directly into your endpoints or service functions. Using DI reduces coupling and makes the code easier to test and maintain.

- **DTO (Data Transfer Object) Pattern**: If your FastAPI endpoints need to format or filter JSON data before returning it, you could use DTOs to handle the conversion between internal data structures and the JSON structure expected by clients. This pattern simplifies validation and can make it easier to adjust response formats without changing underlying business logic.

4. **Cross-Cutting Concerns (Error Handling, Logging, etc.)**

- Singleton Pattern: For components like logging, configuration, and database connection pooling, Singleton can help ensure that there's only one instance of these resources across the application, reducing overhead and improving performance.

- **Observer Pattern**: If you have monitoring or logging requirements that need to react to certain events in the data pipeline (e.g., new file ingestion or data transformation completion), an Observer pattern can help decouple the main logic from the monitoring code. Observers can listen to events and handle them independently, making it easy to add or remove logging and monitoring functionality as needed.

- **Circuit Breaker Pattern**: When your application depends on external services (such as network storage or database connections), Circuit Breaker can help prevent repeated failures if an external service is down. It temporarily breaks the connection after a certain number of failures and retries later, which can improve reliability.

5. **Application Core Design Patterns**

- **Facade** Pattern: A Facade can simplify interactions between your FastAPI, PySpark, and PostgreSQL modules by providing a single, simplified interface. This can make it easier for developers to work with the different modules without needing to know their inner workings.

- **Pipeline** Pattern: Since your app is a data pipeline, the Pipeline pattern helps define the sequence of steps that your data goes through, from ingestion to transformation to storage. Each step in the pipeline (such as loading, transformation, serialization, etc.) could be its own module, and data flows sequentially through these steps. This pattern also simplifies debugging, as each step is clearly defined.

- **Event-Driven Pattern**: For a microservices setup, consider using an event-driven architecture for actions like processing completion or data availability, where modules can notify others about state changes. This pattern is helpful if you plan to scale the application further or integrate additional services that need to respond to these events asynchronously.

Suggested Structure

- **Data Pipeline Module**:

Implements Builder, Strategy, and Pipeline patterns for configurable, sequential data processing.

- Database and Persistence Module:

Uses Repository, Data Mapper, and Singleton patterns to manage PostgreSQL storage, hiding database interactions from the main business logic.

- API Module:

Uses Controller and Dependency Injection patterns to manage endpoints and dependencies.

- Cross-Cutting Module:

Handles logging and error handling using Singleton and Observer patterns, implementing consistent logging across all modules.

Builder Pattern is particularly useful in the Data Processing Layer when building complex, step-by-step transformations on dataframes. Here's how to use the Builder Pattern in your application to handle different processing stages (e.g., filtering, joining, aggregating) in a modular and configurable way.

Let's imagine that your data processing pipeline consists of a series of transformations applied to a PySpark DataFrame. Using the Builder Pattern, you can chain these transformations in a flexible way. Here's an example:

```
### 1. **Define the Data Processing Builder Class**

# In this example, we'll build a `DataFrameBuilder` class
# that encapsulates various transformation steps,
# such as filtering, joining, and aggregating.

from pyspark.sql import DataFrame, SparkSession
from pyspark.sql.functions import col



class DataFrameBuilder:
    def __init__(self, dataframe: DataFrame):
        self._dataframe = dataframe

    def filter_data(self, column_name: str, condition: str):
        """Filter rows based on a condition."""
        self._dataframe = self._dataframe.filter(col(column_name) == condition)
        return self  # Return self to allow chaining

    def join_data(self, other: DataFrame, join_column: str, join_type: str = "inner"):
        """Join with another DataFrame."""
        self._dataframe = self._dataframe.join(other, on=join_column, how=join_type)
        return self

    def select_columns(self, *columns):
        """Select specific columns."""
        self._dataframe = self._dataframe.select(*columns)
        return self

    def aggregate_data(self, group_by_column: str, agg_column: str, agg_func: str):
        """Aggregate data using functions like sum, avg, etc."""
        if agg_func == 'sum':
            self._dataframe = self._dataframe.groupBy(group_by_column).sum(agg_column)
        elif agg_func == 'avg':
            self._dataframe = self._dataframe.groupBy(group_by_column).avg(agg_column)
        # Additional aggregation functions as needed
        return self

    def build(self):
        """Return the fully constructed DataFrame."""
        return self._dataframe
```

```
# Using the Builder Pattern
# With the DataFrameBuilder,
# you can create a transformation pipeline for your PySpark DataFrame
# by chaining the operations together.
# This allows you to easily define the processing steps in a flexible, readable way.


# Initialize Spark session
spark = SparkSession.builder.appName("DataProcessingApp").getOrCreate()

# Sample data to create initial DataFrames
data1 = [("Alice", 34), ("Bob", 45), ("Cathy", 29)]
data2 = [("Alice", "HR"), ("Bob", "Engineering")]

# Create sample DataFrames
df1 = spark.createDataFrame(data1, ["name", "age"])
df2 = spark.createDataFrame(data2, ["name", "department"])

# Use the Builder to apply transformations
processed_df = (DataFrameBuilder(df1)
                .filter_data("age", "34")                       # Filter to get age 34
                .join_data(df2, "name")                         # Join with another
DataFrame
                .select_columns("name", "age", "department")    # Select specific columns
                .aggregate_data("department", "age", "avg")     # Aggregate by department
                .build())                                       # Complete the build

# Show the final result
processed_df.show()
```

### Explanation of the Builder Pattern Usage

- **Chaining Transformations**: Each method in `DataFrameBuilder` returns `self`, allowing for chaining multiple transformations in a single, readable statement.

- **Flexible Data Processing Pipelines**: You can dynamically add or remove steps without modifying the core logic in the main data processing code.

- **Build Step**: The `build()` method returns the fully constructed `DataFrame`, giving you the final, processed data ready for output or further processing.

### Edge Cases

- **Error Handling**: If an incorrect column or transformation is applied, handle exceptions in each transformation method for better debugging.

- **Conditional Logic**: Add logic to selectively apply transformations based on conditions (e.g., skip a join if no matching column exists).

Factory Pattern is ideal for situations where you need to create different data processing pipelines based on specific criteria, such as the type of CSV file being ingested. With this pattern, you can define a factory that dynamically instantiates different processing pipelines based on a configuration or file type.

Below is an example where the Factory Pattern is used to build different data processing pipelines depending on the type of data.

### Scenario

```
# Let’s say you have two types of CSV files:

# Sales Data: Requires filtering, aggregation, and formatting.
# Inventory Data: Requires joining with another dataset and performing some calculations

from abc import ABC, abstractmethod
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import col

class DataProcessor(ABC):
    @abstractmethod
    def process(self, dataframe: DataFrame) -> DataFrame:
        """Process the DataFrame based on specific logic."""
        pass


### 2. **Define Specific Processing Classes**
# Each class below defines a different type of data processing,
# implementing the `process` method as per the specific business logic for that data type.

#### Sales Data Processor

class SalesDataProcessor(DataProcessor):
    def process(self, dataframe: DataFrame) -> DataFrame:
        # Filter for completed sales and calculate total revenue
        df = dataframe.filter(col("status") == "Completed")
        df = df.groupBy("product_id").sum("revenue")
        return df


#### Inventory Data Processor


class InventoryDataProcessor(DataProcessor):
    def process(self, dataframe: DataFrame) -> DataFrame:
        # Example: Perform a join with a products DataFrame to add product details
        products_df = dataframe.sparkSession.createDataFrame(
            [("A101", "Gadget"), ("B202", "Widget")], ["product_id", "product_name"]
        )
        df = dataframe.join(products_df, on="product_id", how="left")
        df = df.withColumn("stock_value", col("quantity") * col("price_per_unit"))
        return df
```

```
### 3. **Implement the Factory Class**
# The factory, `DataProcessorFactory`,
# will determine the type of processor to instantiate based on a provided identifier,
# such as the file type.


class DataProcessorFactory:
    @staticmethod
    def get_processor(file_type: str) -> DataProcessor:
        """Factory method to get the appropriate DataProcessor."""
        if file_type == "sales":
            return SalesDataProcessor()
        elif file_type == "inventory":
            return InventoryDataProcessor()
        else:
            raise ValueError(f"Unknown file type: {file_type}")


### 4. **Using the Factory to Process Data**
# Now, you can use the `DataProcessorFactory`
# to dynamically create and run the appropriate data processing pipeline.


# Initialize Spark session
spark = SparkSession.builder.appName("DataProcessingApp").getOrCreate()

# Sample data for Sales and Inventory
sales_data = [("A101", "Completed", 500.0), ("A102", "Pending", 300.0)]
inventory_data = [("A101", 100, 10.0), ("B202", 200, 15.0)]

# Create DataFrames for Sales and Inventory
sales_df = spark.createDataFrame(sales_data, ["product_id", "status", "revenue"])
inventory_df = spark.createDataFrame(inventory_data, ["product_id", "quantity",
"price_per_unit"])

# Processing Sales Data
processor = DataProcessorFactory.get_processor("sales")
processed_sales_df = processor.process(sales_df)
processed_sales_df.show()

# Processing Inventory Data
processor = DataProcessorFactory.get_processor("inventory")
processed_inventory_df = processor.process(inventory_df)
processed_inventory_df.show()
```

### **Explanation of Factory Pattern Usage**

- ****Factory Class** (`DataProcessorFactory`)**: The factory class provides a `get_processor` method that determines which processing pipeline to instantiate based on the `file_type`.

- **Processor Classes**: Each processing class (e.g., `SalesDataProcessor`, `InventoryDataProcessor`) implements a `process` method according to the business logic needed for each data type.

- **Dynamic Creation**: Using the factory allows you to dynamically choose the correct processor at runtime, keeping the main code clean and extendable.

### Advantages of Using the Factory Pattern

- **Modularity**: Each processor has its own logic, making it easy to add or modify data transformations without affecting other processors.

- **Flexibility**: New processors can be added (e.g., `CustomerDataProcessor`) without changing the existing code, as long as they follow the `DataProcessor` interface.

- **Separation of Concerns**: The factory separates the decision logic of choosing a processor from the actual data processing code.

Strategy Pattern is perfect when you need to apply different algorithms or processing strategies to your data depending on conditions, such as business rules or user input. By using this pattern, you can define a family of processing strategies, encapsulate them as separate classes, and switch between them at runtime based on certain criteria.

For example, let's say your data processing needs to apply different filter strategies depending on the type of analysis being performed.

### **Scenario**

Suppose you have different strategies for filtering:

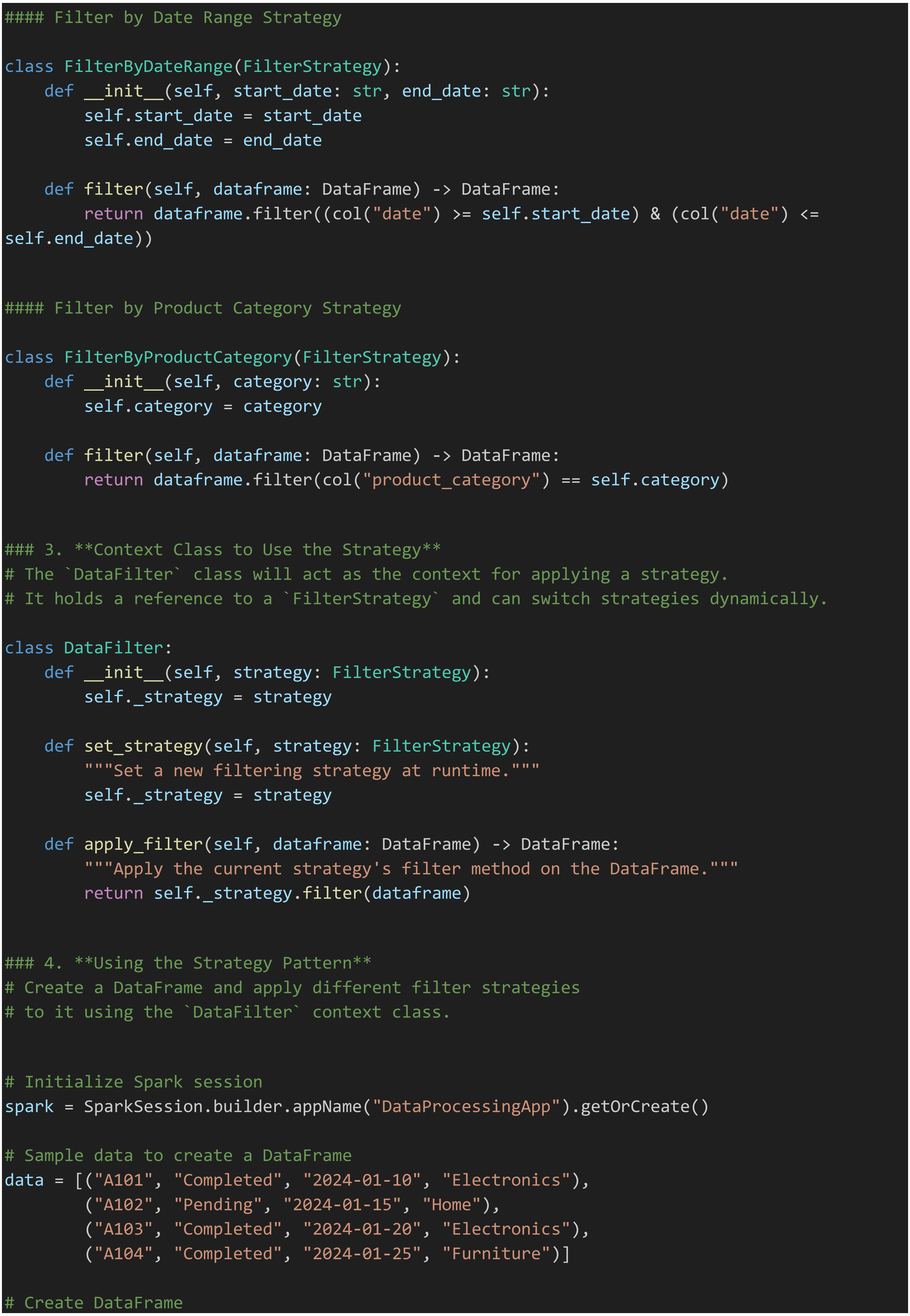
1. ****Filter by Status****: Only include rows with a specific status, such as "Completed."

2. ****Filter by Date Range****: Include rows within a specified date range.

3. ****Filter by Product Category****: Include rows of a certain product category.

Using the Strategy Pattern, we can encapsulate each filter as a separate strategy class and apply them based on user input or other runtime conditions.

```python
### 1. **Define the Strategy Interface**
# Create an abstract base class to define the interface for all filter strategies.
# Each filter strategy will implement a `filter` method.


from abc import ABC, abstractmethod
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import col


class FilterStrategy(ABC):
    @abstractmethod
    def filter(self, dataframe: DataFrame) -> DataFrame:
        """Apply a filtering strategy to the DataFrame."""
        pass


### 2. **Implement Concrete Strategies**
# Each concrete strategy class implements a specific
# filtering approach by defining its own `filter` method.
#### Filter by Status Strategy


class FilterByStatus(FilterStrategy):
    def __init__(self, status: str):
        self.status = status

    def filter(self, dataframe: DataFrame) -> DataFrame:
        return dataframe.filter(col("status") == self.status)
```



```
#### Filter by Date Range Strategy

class FilterByDateRange(FilterStrategy):
    def __init__(self, start_date: str, end_date: str):
        self.start_date = start_date
        self.end_date = end_date

    def filter(self, dataframe: DataFrame) -> DataFrame:
        return dataframe.filter((col("date") >= self.start_date) & (col("date") <=
self.end_date))


#### Filter by Product Category Strategy

class FilterByProductCategory(FilterStrategy):
    def __init__(self, category: str):
        self.category = category

    def filter(self, dataframe: DataFrame) -> DataFrame:
        return dataframe.filter(col("product_category") == self.category)


### 3. **Context Class to Use the Strategy**
# The `DataFilter` class will act as the context for applying a strategy.
# It holds a reference to a `FilterStrategy` and can switch strategies dynamically.

class DataFilter:
    def __init__(self, strategy: FilterStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: FilterStrategy):
        """Set a new filtering strategy at runtime."""
        self._strategy = strategy

    def apply_filter(self, dataframe: DataFrame) -> DataFrame:
        """Apply the current strategy's filter method on the DataFrame."""
        return self._strategy.filter(dataframe)


### 4. **Using the Strategy Pattern**
# Create a DataFrame and apply different filter strategies
# to it using the `DataFilter` context class.


# Initialize Spark session
spark = SparkSession.builder.appName("DataProcessingApp").getOrCreate()

# Sample data to create a DataFrame
data = [("A101", "Completed", "2024-01-10", "Electronics"),
        ("A102", "Pending", "2024-01-15", "Home"),
        ("A103", "Completed", "2024-01-20", "Electronics"),
        ("A104", "Completed", "2024-01-25", "Furniture")]

# Create DataFrame
```

```
df = spark.createDataFrame(data, ["order_id", "status", "date", "product_category"])

# Instantiate a DataFilter context with a specific strategy
data_filter = DataFilter(strategy=FilterByStatus("Completed"))
filtered_df = data_filter.apply_filter(df)
print("Filtered by Status:")
filtered_df.show()

# Switch to a date range strategy
data_filter.set_strategy(FilterByDateRange("2024-01-10", "2024-01-20"))
filtered_df = data_filter.apply_filter(df)
print("Filtered by Date Range:")
filtered_df.show()

# Switch to a product category strategy
data_filter.set_strategy(FilterByProductCategory("Electronics"))
filtered_df = data_filter.apply_filter(df)
print("Filtered by Product Category:")
filtered_df.show()
```

### Explanation of Strategy Pattern Usage

- **Strategy Interface (`FilterStrategy`)**: Provides a common `filter` method that all filtering strategies must implement, enforcing consistency across strategies.

- **Concrete Strategies (`FilterByStatus`, `FilterByDateRange`, `FilterByProductCategory`)**: Each concrete class defines a specific filtering technique, making the logic modular and allowing for different implementations.

- **Context Class (`DataFilter`)**: Holds a reference to a `FilterStrategy` and can switch strategies at runtime via `set_strategy`. This allows for dynamic selection of filtering strategies based on runtime conditions.

### Benefits of Using the Strategy Pattern

- **Flexibility**: Strategies can be swapped in and out easily, allowing you to modify filtering behavior dynamically.

- **Modularity**: Each filter strategy is encapsulated in its own class, making it easy to test, maintain, and extend.

- **Reusability**: Strategies can be reused across different parts of the application where similar filtering needs arise.

Repository Pattern is an excellent choice for organizing the data access and persistence logic in your application. It abstracts the details of data storage, allowing the business logic to remain unaware of whether data is stored in a database, a file system, or other sources. In the context of your application, the Repository Pattern can help separate the logic of saving data to PostgreSQL and retrieving JSON data for further processing and API responses.

Here's a breakdown of how to implement the Repository Pattern for both serialization to JSON and persistence to PostgreSQL.

### **Scenario**

Suppose you want to save processed data into PostgreSQL as JSON, retrieve it for the FastAPI layer, and handle serialization and deserialization for interacting with the data.

```
### 1. **Define the Repository Interface**
# Start by defining an abstract base class, `DataRepository`,
# which will serve as the interface for any concrete repository that interacts with
PostgreSQL.


import json
from abc import ABC, abstractmethod
from pyspark.sql import SparkSession, DataFrame, Row


class DataRepository(ABC):
    @abstractmethod
    def save(self, dataframe: DataFrame, table_name: str):
        """Save DataFrame to the database."""
        pass

    @abstractmethod
    def load(self, table_name: str) -> DataFrame:
        """Load data from the database into a DataFrame."""
        pass


### 2. Implement a PostgreSQL Repository
# This concrete repository uses PySpark's
# JDBC support to save and load data to and from PostgreSQL.
# It handles serialization and deserialization to JSON,
# saving JSON data into the specified table in PostgreSQL.


class PostgresDataRepository(DataRepository):
    def __init__(self, spark: SparkSession, jdbc_url: str, db_properties: dict):
        self.spark = spark
        self.jdbc_url = jdbc_url
        self.db_properties = db_properties

    def save(self, dataframe: DataFrame, table_name: str):
        """Save DataFrame as JSON to the PostgreSQL table."""
        dataframe.write.jdbc(
            url=self.jdbc_url,
            table=table_name,
            mode="append",
            properties=self.db_properties
        )
        print(f"Data saved to PostgreSQL table: {table_name}")

    def load(self, table_name: str) -> DataFrame:
        """Load JSON data from PostgreSQL and return as DataFrame."""
        dataframe = self.spark.read.jdbc(
            url=self.jdbc_url,
            table=table_name,
            properties=self.db_properties
        )
        print(f"Data loaded from PostgreSQL table: {table_name}")
        return dataframe


### 3. **Implement Serialization and Deserialization Helper Methods**
# If JSON serialization/deserialization is required for data
# before saving to or after loading from PostgreSQL,
# you can add helper functions in the repository.


class PostgresDataRepository(DataRepository):
    def __init__(self, spark: SparkSession, jdbc_url: str, db_properties: dict):
        self.spark = spark
        self.jdbc_url = jdbc_url
        self.db_properties = db_properties

    def save(self, dataframe: DataFrame, table_name: str):
        # Save data as JSON strings in the database
        json_df = dataframe.toJSON().toDF("json_data")
        json_df.write.jdbc(
            url=self.jdbc_url,
            table=table_name,
            mode="append",
            properties=self.db_properties
        )
        print(f"JSON data saved to PostgreSQL table: {table_name}")

    def load(self, table_name: str) -> DataFrame:
        # Load JSON data and parse back to DataFrame rows
        json_df = self.spark.read.jdbc(
            url=self.jdbc_url,
            table=table_name,
            properties=self.db_properties
        )
        parsed_df = self.spark.read.json(json_df.rdd.map(lambda row: row.json_data))
        print(f"JSON data loaded from PostgreSQL table: {table_name}")
        return parsed_df
```

```
### 4. **Using the Repository in the Application**
# Here’s how you can use `PostgresDataRepository` to save and load data in your
application.
# Set up the Spark session and database properties


spark = SparkSession.builder.appName("DataProcessingApp").getOrCreate()
jdbc_url = "jdbc:postgresql://localhost:5432/mydb"
db_properties = {
    "user": "myuser",
    "password": "mypassword",
    "driver": "org.postgresql.Driver"
}

# Create the repository instance
repository = PostgresDataRepository(spark, jdbc_url, db_properties)

# Example data
data = [("A101", 1000), ("B202", 2000)]
df = spark.createDataFrame(data, ["product_id", "revenue"])

# Save the data
repository.save(df, "processed_data")

# Load the data back
loaded_df = repository.load("processed_data")
loaded_df.show()
```

Explanation:

Repository Interface (`**DataRepository**`): Defines an interface for all repository classes, providing consistent methods (`save` and `load`) to interact with the database, regardless of the specific DB.

Concrete Repository (`**PostgresDataRepository**`): Implements the `DataRepository` interface and provides logic to save PySpark DataFrames as JSON strings and load JSON strings from PostgreSQL. This ensures the rest of the application is not tightly coupled to PostgreSQL.

Serialization and Deserialization: By converting DataFrames to JSON strings before saving and parsing JSON strings back into DataFrames after loading, you maintain a consistent data format in the database that FastAPI can easily consume.

Benefits:

Decoupling of Business Logic and Data Access: The repository pattern abstracts database interaction, allowing business logic to focus on data handling without knowing about storage details.

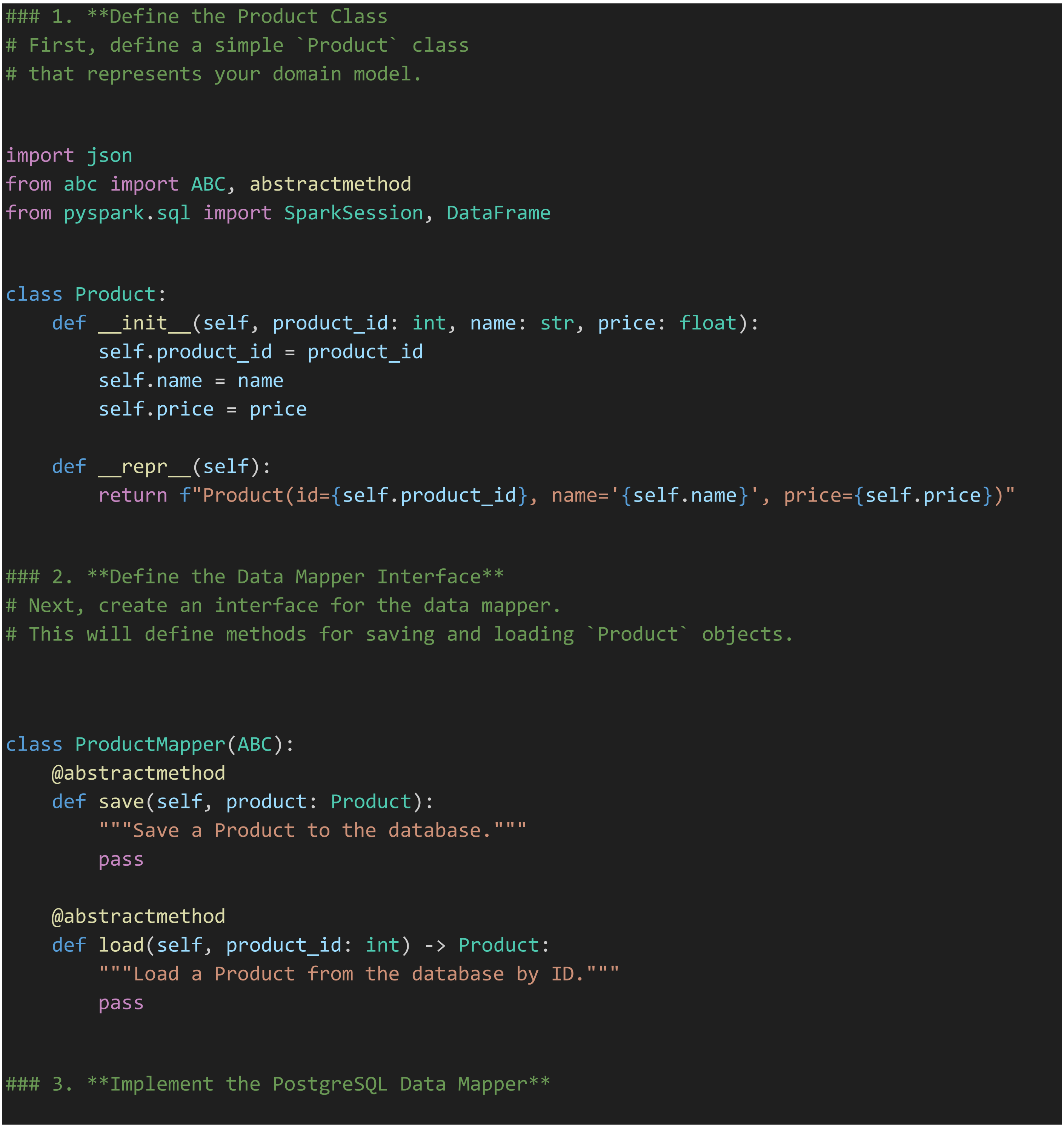
Flexibility and Scalability: Different repository implementations (e.g., `FileSystemRepository`, `MongoRepository`) can be added without modifying business logic.

Consistency: The repository interface provides a standard way to save and load data across different parts of the application, helping enforce consistency.

Data Mapper Pattern is a design pattern that separates the in-memory representation of data (i.e., your business logic) from the database. It allows you to map your objects to the database tables without tightly coupling your data access logic to your domain model. This pattern is especially useful when you want to maintain a clear separation between your application's business logic and data storage logic.

### **Scenario**

Let’s create an example where we need to manage a simple `Product` entity with fields such as `product_id`, `name`, and `price`. We'll implement a data mapper to handle the serialization and persistence of `Product` objects to a PostgreSQL database.



```python
### 1. **Define the Product Class
# First, define a simple `Product` class
# that represents your domain model.


import json
from abc import ABC, abstractmethod
from pyspark.sql import SparkSession, DataFrame


class Product:
    def __init__(self, product_id: int, name: str, price: float):
        self.product_id = product_id
        self.name = name
        self.price = price

    def __repr__(self):
        return f"Product(id={self.product_id}, name='{self.name}', price={self.price})"


### 2. **Define the Data Mapper Interface**
# Next, create an interface for the data mapper.
# This will define methods for saving and loading `Product` objects.



class ProductMapper(ABC):
    @abstractmethod
    def save(self, product: Product):
        """Save a Product to the database."""
        pass

    @abstractmethod
    def load(self, product_id: int) -> Product:
        """Load a Product from the database by ID."""
        pass


### 3. **Implement the PostgreSQL Data Mapper**
```

```
# Now, implement the `ProductMapper`
# interface for PostgreSQL using a class.
# This class will handle the connection to the database
# and implement the logic to save and load products.



class PostgresProductMapper(ProductMapper):
    def __init__(self, spark: SparkSession, jdbc_url: str, db_properties: dict):
        self.spark = spark
        self.jdbc_url = jdbc_url
        self.db_properties = db_properties

    def save(self, product: Product):
        """Save a Product to the database."""
        # Convert Product to DataFrame
        product_data = [(product.product_id, product.name, product.price)]
        df = self.spark.createDataFrame(product_data, ["product_id", "name", "price"])

        # Save DataFrame to PostgreSQL
        df.write.jdbc(
            url=self.jdbc_url,
            table="products",
            mode="append",
            properties=self.db_properties
        )
        print(f"Product saved to PostgreSQL: {product}")

    def load(self, product_id: int) -> Product:
        """Load a Product from the database by ID."""
        df = self.spark.read.jdbc(
            url=self.jdbc_url,
            table="products",
            properties=self.db_properties
        )

        product_row = df.filter(df.product_id == product_id).collect()
        if product_row:
            return Product(product_row[0].product_id, product_row[0].name,
product_row[0].price)
        else:
            return None


### 4. **Using the Data Mapper in Your Application**

# Finally, let's see how to use the `PostgresProductMapper`
# to save and load products.


# Set up the Spark session and database properties

spark = SparkSession.builder.appName("DataProcessingApp").getOrCreate()
jdbc_url = "jdbc:postgresql://localhost:5432/mydb"
```

```
db_properties = {
    "user": "myuser",
    "password": "mypassword",
    "driver": "org.postgresql.Driver"
}

# Create the product mapper instance
product_mapper = PostgresProductMapper(spark, jdbc_url, db_properties)

# Example product
new_product = Product(product_id=1, name="Laptop", price=1200.50)

# Save the product
product_mapper.save(new_product)

# Load the product by ID
loaded_product = product_mapper.load(1)
print("Loaded Product:", loaded_product)
```

Explanation:

Domain Model (`Product`): This class represents the entity in your business logic. It contains no knowledge of how data is persisted or loaded.

Mapper Interface (`ProductMapper`): Defines methods for saving and loading products, ensuring that any implementation adheres to the same contract.

Concrete Mapper (`PostgresProductMapper`): Implements the data access logic for PostgreSQL, encapsulating the specifics of how products are saved and retrieved from the database. This keeps the business logic clean and decoupled from the data access code.

Benefits

1. Separation of Concerns: By separating the data access logic from the domain model, the codebase remains clean and easier to maintain.

2. Flexibility: Changing the underlying storage or database does not require significant changes in your business logic. You would just need to change or replace the mapper implementation.

3. Testability: You can easily mock the mapper for unit testing, allowing you to test the business logic without hitting the database.

4. Encapsulation of Data Access Logic: The data access details are encapsulated in the mapper, making it easier to manage and evolve over time.

Adapter Pattern is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two interfaces, allowing them to communicate without needing to modify their existing code. In the context of a data serialization and persistence layer, you can use the Adapter Pattern to adapt different data storage or serialization methods to a common interface that your application can work with.

### **Scenario**

Let's consider a scenario where you want to serialize and persist a `Product` object, but you have two different storage options: PostgreSQL and a JSON file. You want to create a common interface for saving and loading `Product` objects while using different adapters for each storage option.

```
### 1. **Define the Product Class**
# First, define the `Product`
# class representing your domain model.

import json
import os
from abc import ABC, abstractmethod
from pyspark.sql import SparkSession

class Product:
    def __init__(self, product_id: int, name: str, price: float):
        self.product_id = product_id
        self.name = name
        self.price = price

    def __repr__(self):
        return f"Product(id={self.product_id}, name='{self.name}', price={self.price})"


### 2. **Define the Common Interface**

# Next, create an interface for the product storage,
# which will define methods for saving and loading products.


class ProductStorage(ABC):
    @abstractmethod
    def save(self, product: Product):
        """Save a Product."""
        pass

    @abstractmethod
    def load(self, product_id: int) -> Product:
        """Load a Product by ID."""
        pass


### 3. **Implement the PostgreSQL Adapter**

# Now, implement an adapter for PostgreSQL
# that adapts the `ProductStorage` interface
# to work with a PostgreSQL database.
```

```
class PostgresProductAdapter(ProductStorage):
    def __init__(self, spark: SparkSession, jdbc_url: str, db_properties: dict):
        self.spark = spark
        self.jdbc_url = jdbc_url
        self.db_properties = db_properties

    def save(self, product: Product):
        """Save a Product to PostgreSQL."""
        product_data = [(product.product_id, product.name, product.price)]
        df = self.spark.createDataFrame(product_data, ["product_id", "name", "price"])

        # Save DataFrame to PostgreSQL
        df.write.jdbc(
            url=self.jdbc_url,
            table="products",
            mode="append",
            properties=self.db_properties
        )
        print(f"Product saved to PostgreSQL: {product}")

    def load(self, product_id: int) -> Product:
        """Load a Product from PostgreSQL by ID."""
        df = self.spark.read.jdbc(
            url=self.jdbc_url,
            table="products",
            properties=self.db_properties
        )

        product_row = df.filter(df.product_id == product_id).collect()
        if product_row:
            return Product(product_row[0].product_id, product_row[0].name,
product_row[0].price)
        else:
            return None


### 4. **Implement the JSON File Adapter**

# Next, create an adapter for saving and loading products to and from a JSON file.


class JsonFileProductAdapter(ProductStorage):
    def __init__(self, json_file: str):
        self.json_file = json_file

    def save(self, product: Product):
        """Save a Product to a JSON file."""
        if not os.path.exists(self.json_file):
            data = []
        else:
            with open(self.json_file, 'r') as file:
                data = json.load(file)
        data.append({
            "product_id": product.product_id,
            "name": product.name,
            "price": product.price
        })

        with open(self.json_file, 'w') as file:
            json.dump(data, file, indent=4)
        print(f"Product saved to JSON file: {product}")

    def load(self, product_id: int) -> Product:
        """Load a Product from a JSON file by ID."""
        if not os.path.exists(self.json_file):
            return None

        with open(self.json_file, 'r') as file:
            data = json.load(file)

        for item in data:
            if item['product_id'] == product_id:
                return Product(item['product_id'], item['name'], item['price'])

        return None


### 5. **Using the Adapters in Your Application**

# Finally, demonstrate how to use the adapters
# to save and load products using both PostgreSQL and JSON file storage.


# Set up the Spark session and database properties for PostgreSQL
spark = SparkSession.builder.appName("DataProcessingApp").getOrCreate()
jdbc_url = "jdbc:postgresql://localhost:5432/mydb"
db_properties = {
    "user": "myuser",
    "password": "mypassword",
    "driver": "org.postgresql.Driver"
}

# Create the PostgreSQL adapter instance
postgres_adapter = PostgresProductAdapter(spark, jdbc_url, db_properties)

# Create the JSON file adapter instance
json_adapter = JsonFileProductAdapter("products.json")

# Example product
new_product = Product(product_id=1, name="Laptop", price=1200.50)

# Save the product using PostgreSQL
postgres_adapter.save(new_product)

# Load the product by ID from PostgreSQL
```

```
loaded_product_postgres = postgres_adapter.load(1)
print("Loaded Product from PostgreSQL:", loaded_product_postgres)

# Save the product using JSON file storage
json_adapter.save(new_product)

# Load the product by ID from JSON file
loaded_product_json = json_adapter.load(1)
print("Loaded Product from JSON file:", loaded_product_json)
```

Explanation:

Domain Model (`Product`)**: This class represents the entity in your business logic without knowledge of how it is persisted or loaded.

Common Interface (`ProductStorage`)**: Defines the contract for saving and loading products, allowing different adapters to implement this interface.

Adapters:

PostgresProductAdapter: Implements the interface for PostgreSQL, providing methods to save and load products from the database.

JsonFileProductAdapter: Implements the interface for JSON file storage, providing methods to save and load products to and from a JSON file.

Benefits:

1. ****Flexibility****: Easily switch between different storage implementations without changing the business logic. You can add new adapters for other storage options as needed.

2. ****Decoupling****: Your business logic interacts with a common interface (`ProductStorage`), reducing dependency on specific storage technologies.

3. ****Testability****: You can easily mock the adapters in your unit tests to simulate different storage scenarios without needing a real database or file system.

4. ****Reusability****: The same business logic can work with different storage implementations, promoting code reuse.

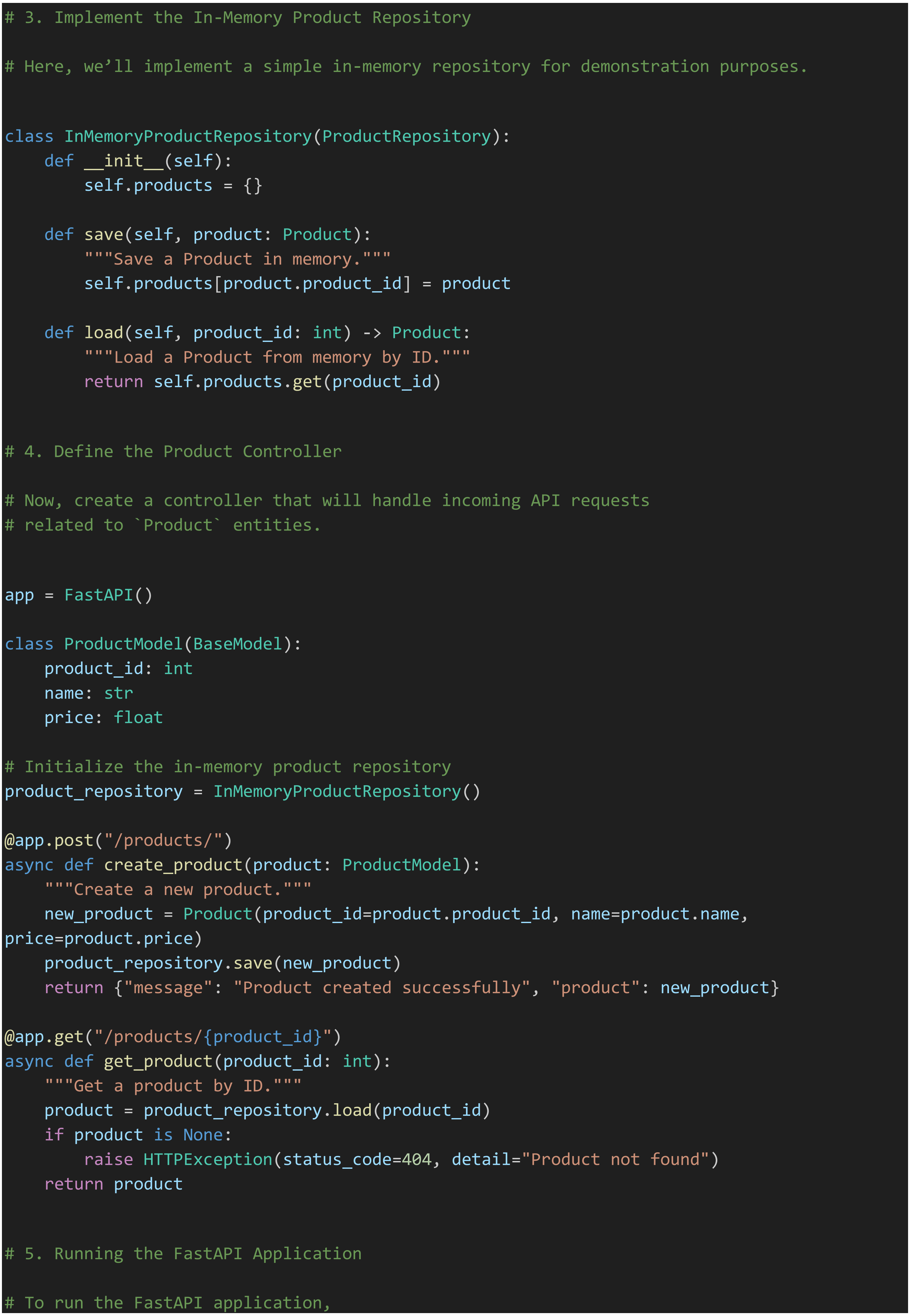
Controller Pattern is commonly used in the context of web applications and APIs to manage the flow of data between the model (business logic) and the view (UI or response). In this pattern, controllers act as intermediaries that handle incoming requests, process them (often by interacting with the model), and return responses to the client.

In the context of your FastAPI application for the data processing layer, you can use the Controller Pattern to handle API requests for managing `Product` entities.

Scenario

Let’s create a simple API using FastAPI to manage `Product` objects. We'll define endpoints to create and retrieve products, demonstrating how the Controller Pattern can be applied in this context.

```
# 1. Define the Product Class

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from abc import ABC, abstractmethod

# First, define the `Product` class
# representing your domain model.


class Product:
    def __init__(self, product_id: int, name: str, price: float):
        self.product_id = product_id
        self.name = name
        self.price = price

    def __repr__(self):
        return f"Product(id={self.product_id}, name='{self.name}', price={self.price})"


# 2. Define the Product Repository Interface

# Next, create an interface for the repository,
# which will define methods for saving and loading products.


class ProductRepository(ABC):
    @abstractmethod
    def save(self, product: Product):
        """Save a Product."""
        pass

    @abstractmethod
    def load(self, product_id: int) -> Product:
        """Load a Product by ID."""
        pass
```



```
# 3. Implement the In-Memory Product Repository

# Here, we’ll implement a simple in-memory repository for demonstration purposes.


class InMemoryProductRepository(ProductRepository):
    def __init__(self):
        self.products = {}

    def save(self, product: Product):
        """Save a Product in memory."""
        self.products[product.product_id] = product

    def load(self, product_id: int) -> Product:
        """Load a Product from memory by ID."""
        return self.products.get(product_id)


# 4. Define the Product Controller

# Now, create a controller that will handle incoming API requests
# related to `Product` entities.


app = FastAPI()

class ProductModel(BaseModel):
    product_id: int
    name: str
    price: float

# Initialize the in-memory product repository
product_repository = InMemoryProductRepository()

@app.post("/products/")
async def create_product(product: ProductModel):
    """Create a new product."""
    new_product = Product(product_id=product.product_id, name=product.name,
price=product.price)
    product_repository.save(new_product)
    return {"message": "Product created successfully", "product": new_product}

@app.get("/products/{product_id}")
async def get_product(product_id: int):
    """Get a product by ID."""
    product = product_repository.load(product_id)
    if product is None:
        raise HTTPException(status_code=404, detail="Product not found")
    return product


# 5. Running the FastAPI Application

# To run the FastAPI application,
# save the code above in a file (e.g., `main.py`) and run it with Uvicorn:

# bash
# uvicorn main:app --reload


# 6. Testing the API

# You can test the API using tools like Postman or Curl.
# Here’s how you can test it:

# Create a Product
# POST request to -> http://127.0.0.1:8000/products/

# json
# {
#     "product_id": 1,
#     "name": "Laptop",
#     "price": 1200.50
# }


# Get a Product
#  GET request to -> http://127.0.0.1:8000/products/1

# You should receive the product details in the response.
```

Explanation

Domain Model (`Product`)**: This class represents the entity in your business logic without knowledge of how it is persisted or loaded.

Repository Interface (`ProductRepository`)**: Defines the contract for saving and loading products, allowing different implementations to be used.

In-Memory Repository (`InMemoryProductRepository`)**: Provides an in-memory storage solution for demonstration purposes.

Controller: The `create_product` and `get_product` functions act as controllers that handle incoming requests. They interact with the repository to manage products, following the Controller Pattern.

Benefits

1. **Separation of Concerns**: Controllers manage the interaction between the API layer and business logic, ensuring that each part of the application has a distinct responsibility.

2. **Maintainability**: With a clear separation of responsibilities, it’s easier to maintain and modify your application as requirements change.

3. **Testability**: Controllers can be easily tested in isolation, allowing you to ensure that your API logic behaves as expected.

4. **Scalability**: The Controller Pattern makes it easier to extend your application by adding new endpoints or modifying existing ones without affecting other parts of the codebase.

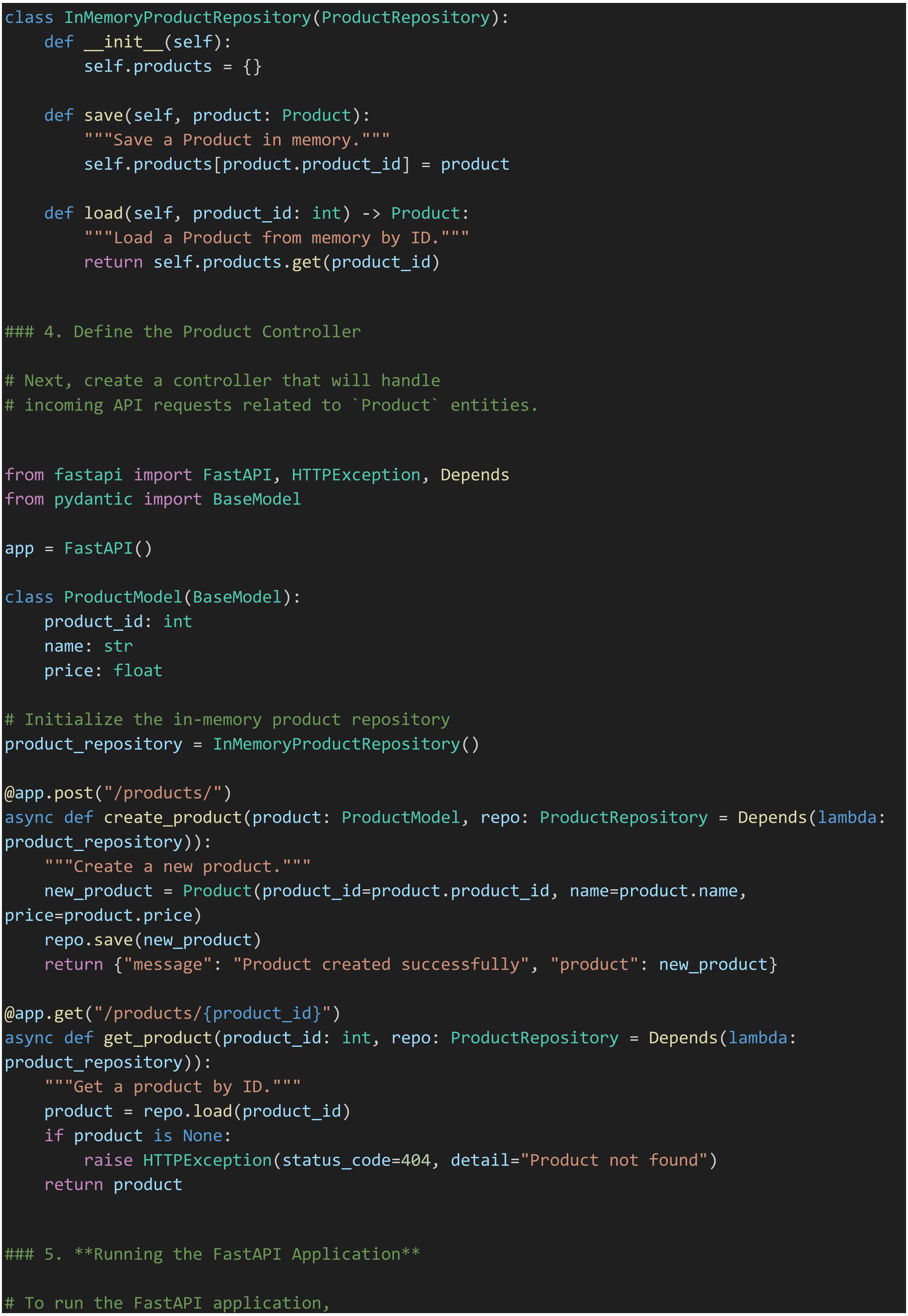
Dependency Injection (DI) Pattern is a design pattern that allows you to achieve Inversion of Control (IoC) between classes and their dependencies. It promotes loose coupling and enhances the testability of your application by injecting dependencies rather than having the classes instantiate them internally.

In a FastAPI application, you can use dependency injection to manage the lifecycle of dependencies like services or repositories, ensuring they are shared and efficiently managed.

Scenario

Let’s create a simple FastAPI application that manages `Product` entities, demonstrating how to use the Dependency Injection Pattern for the API layer.

```
### 1. Define the Product Class

# First, define the `Product` class
# representing your domain model.


class Product:
    def __init__(self, product_id: int, name: str, price: float):
        self.product_id = product_id
        self.name = name
        self.price = price

    def __repr__(self):
        return f"Product(id={self.product_id}, name='{self.name}', price={self.price})"


### 2. Define the Product Repository Interface

# Next, create an interface for the repository,
# which will define methods for saving and loading products.


from abc import ABC, abstractmethod

class ProductRepository(ABC):
    @abstractmethod
    def save(self, product: Product):
        """Save a Product."""
        pass

    @abstractmethod
    def load(self, product_id: int) -> Product:
        """Load a Product by ID."""
        pass


### 3. Implement the In-Memory Product Repository

# We’ll implement an in-memory repository for demonstration purposes.
```



```
class InMemoryProductRepository(ProductRepository):
    def __init__(self):
        self.products = {}

    def save(self, product: Product):
        """Save a Product in memory."""
        self.products[product.product_id] = product

    def load(self, product_id: int) -> Product:
        """Load a Product from memory by ID."""
        return self.products.get(product_id)


### 4. Define the Product Controller

# Next, create a controller that will handle
# incoming API requests related to `Product` entities.


from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel

app = FastAPI()

class ProductModel(BaseModel):
    product_id: int
    name: str
    price: float

# Initialize the in-memory product repository
product_repository = InMemoryProductRepository()

@app.post("/products/")
async def create_product(product: ProductModel, repo: ProductRepository = Depends(lambda:
product_repository)):
    """Create a new product."""
    new_product = Product(product_id=product.product_id, name=product.name,
price=product.price)
    repo.save(new_product)
    return {"message": "Product created successfully", "product": new_product}

@app.get("/products/{product_id}")
async def get_product(product_id: int, repo: ProductRepository = Depends(lambda:
product_repository)):
    """Get a product by ID."""
    product = repo.load(product_id)
    if product is None:
        raise HTTPException(status_code=404, detail="Product not found")
    return product


### 5. **Running the FastAPI Application**

# To run the FastAPI application,
# save the code above in a file (e.g., `main.py`) and run it with Uvicorn:

# bash
# uvicorn main:app --reload


# ### 6. Testing the API

# You can test the API using tools like Postman or Curl.
# Here’s how you can test it:

# Create a Product (POST request to `http://127.0.0.1:8000/products/`):

# json
# {
#     "product_id": 1,
#     "name": "Laptop",
#     "price": 1200.50
# }


# Get a Product (GET request to `http://127.0.0.1:8000/products/1`):

# You should receive the product details in the response.
```

Explanation

Domain Model (`Product`): This class represents the entity in your business logic.

Repository Interface (`ProductRepository`): Defines the contract for saving and loading products.

In-Memory Repository (`InMemoryProductRepository`): Provides an in-memory storage solution.

Controller: The `create_product` and `get_product` functions use FastAPI's `Depends` to inject the `ProductRepository` dependency. This allows the controller to interact with the repository without needing to create it or manage its lifecycle explicitly.

Benefits

1. Loose Coupling: Classes depend on abstractions (interfaces) rather than concrete implementations, making it easy to swap out implementations or change the behavior.

2. Testability: Dependencies can be easily mocked or replaced, allowing for unit testing of components in isolation.

3. Reusability: The same repository can be reused across different parts of the application, promoting code reuse.

4. Flexibility: You can easily change the implementation of a dependency without affecting the dependent classes.

Data Transfer Object (DTO) Pattern is used to encapsulate data and transfer it between layers of an application, often over a network. DTOs are particularly useful in APIs to decouple the internal representation of data from the external representation exposed to clients. This helps in reducing the amount of data sent over the network and makes it easier to manage data serialization and deserialization.

Scenario

Let's create a simple FastAPI application that manages `Product` entities and demonstrates the use of DTOs to transfer product data between the API layer and the clients.

```
### 1. Define the Domain Model

# First, define the `Product` class
# representing your domain model.

from abc import ABC, abstractmethod

from pydantic import BaseModel
from fastapi import FastAPI, HTTPException, Depends

class Product:
    def __init__(self, product_id: int, name: str, price: float):
        self.product_id = product_id
        self.name = name
        self.price = price

    def __repr__(self):
        return f"Product(id={self.product_id},
            name='{self.name}', price={self.price})"


### 2. Define the Product DTO

# Next, create a DTO class to
# define the structure of the product data
# that will be transferred over the API.


class ProductDTO(BaseModel):
    product_id: int
    name: str
    price: float

    class Config:
        orm_mode = True  # Enables compatibility with ORM models


### 3. Define the Product Repository Interface

# Now, create an interface for the repository,
# which will define methods for saving and loading products.
```

```python
class ProductRepository(ABC):
    @abstractmethod
    def save(self, product: Product):
        """Save a Product."""
        pass

    @abstractmethod
    def load(self, product_id: int) -> Product:
        """Load a Product by ID."""
        pass


### 4. Implement the In-Memory Product Repository
# Here, we’ll implement
# an in-memory repository for demonstration purposes.


class InMemoryProductRepository(ProductRepository):
    def __init__(self):
        self.products = {}

    def save(self, product: Product):
        """Save a Product in memory."""
        self.products[product.product_id] = product

    def load(self, product_id: int) -> Product:
        """Load a Product from memory by ID."""
        return self.products.get(product_id)


### 5. Define the Product Controller with DTO
# Now, create a controller that will handle incoming API requests
# related to `Product` entities, using the DTO for data transfer.
app = FastAPI()

# Initialize the in-memory product repository
product_repository = InMemoryProductRepository()


@app.post("/products/", response_model=ProductDTO)
async def create_product(product_dto: ProductDTO):
    """Create a new product."""
    new_product = Product(product_id=product_dto.product_id, name=product_dto.name,
price=product_dto.price)
    product_repository.save(new_product)
    return new_product  # Return the ProductDTO response


@app.get("/products/{product_id}", response_model=ProductDTO)
async def get_product(product_id: int):
    """Get a product by ID."""
    product = product_repository.load(product_id)
    if product is None:
        raise HTTPException(status_code=404, detail="Product not found")
    return ProductDTO(product_id=product.product_id, name=product.name,
price=product.price)



### 6. Running the FastAPI Application

# To run the FastAPI application,
# save the code above in a file (e.g., `main.py`)
# and run it with Uvicorn:

# bash
# uvicorn main:app --reload


# ### 7. Testing the API

# You can test the API using tools
# like Postman or Curl. Here’s how you can test it:

# Create a Product-
# POST request to -> http://127.0.0.1:8000/products/:

# json
# {
#     "product_id": 1,
#     "name": "Laptop",
#     "price": 1200.50
# }


# Get a Product-
# GET request to -> http://127.0.0.1:8000/products/1:

# You should receive the product details in the response.
```

Explanation

Domain Model (` Product`): Represents the entity in your business logic, typically used for internal operations.

-DTO (` ProductDTO`): This class defines the structure of the product data to be sent over the API. It uses Pydantic for validation and serialization.

-Repository Interface (` ProductRepository`): Defines the contract for saving and loading products.

In-Memory Repository (` InMemoryProductRepository`): Provides an in-memory storage solution.

Controller: The `create_product` and `get_product` functions use the `ProductDTO` to handle incoming and outgoing data. The DTO acts as a bridge between the API layer and the internal model.

Benefits

1. Decoupling: DTOs separate the internal representation of data from the external representation exposed to clients, allowing changes in the internal model without affecting the API.

2. Reduced Data Transfer: DTOs can aggregate or filter data, reducing the size of the data sent over the network.

3. Validation: Using a DTO can enforce validation rules for incoming data, helping to ensure that only valid data is processed.

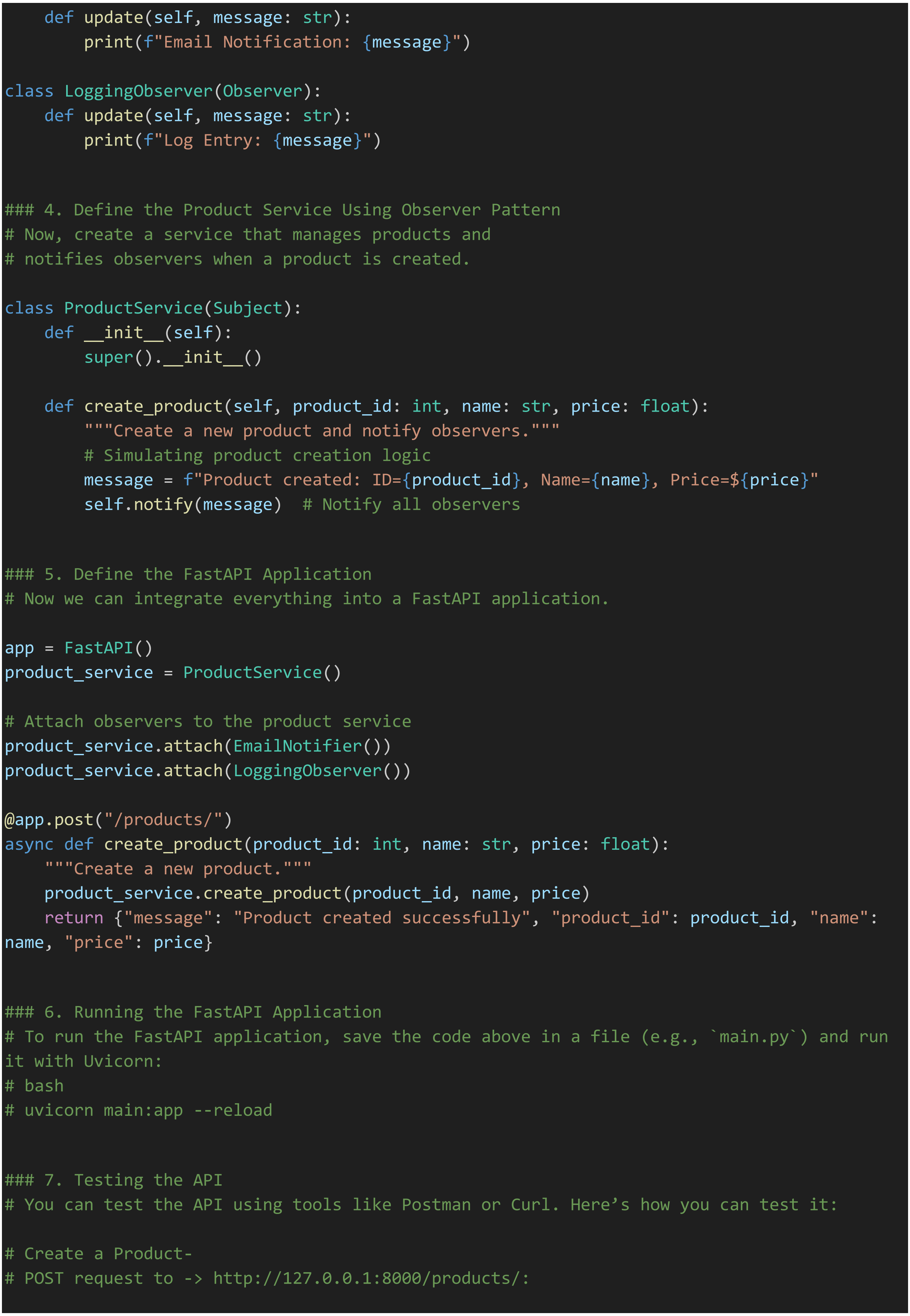
4. Improved Maintainability: Changes to the API can be made in the DTOs without impacting the underlying business logic, making the application easier to maintain.

Observer Pattern is a behavioural design pattern that defines a one-to-many dependency between objects, allowing multiple observers to be notified of changes in a subject's state. This pattern is useful for implementing event-driven systems, where various components need to respond to events or changes in state.

Scenario

Let's create a simple FastAPI application that demonstrates the Observer Pattern with a notification system. In this example, we'll create a `Product` service that notifies registered observers (subscribers) whenever a new product is created.

```
### 1. Define the Observer Interface
# First, define an interface for the observers that will be notified of changes.

from abc import ABC, abstractmethod
from fastapi import FastAPI


class Observer(ABC):
    @abstractmethod
    def update(self, message: str):
        """Receive update from the subject."""
        pass


### 2. Define the Subject Class
# Next, create a `Subject` class that manages observers
# and notifies them when a product is created.


class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer: Observer):
        """Attach an observer to the subject."""
        self._observers.append(observer)

    def detach(self, observer: Observer):
        """Detach an observer from the subject."""
        self._observers.remove(observer)

    def notify(self, message: str):
        """Notify all observers of a change."""
        for observer in self._observers:
            observer.update(message)


### 3. Define Concrete Observers
# Implement concrete observer classes that will react to notifications.

class EmailNotifier(Observer):
```



```
    def update(self, message: str):
        print(f"Email Notification: {message}")

class LoggingObserver(Observer):
    def update(self, message: str):
        print(f"Log Entry: {message}")


### 4. Define the Product Service Using Observer Pattern
# Now, create a service that manages products and
# notifies observers when a product is created.

class ProductService(Subject):
    def __init__(self):
        super().__init__()

    def create_product(self, product_id: int, name: str, price: float):
        """Create a new product and notify observers."""
        # Simulating product creation logic
        message = f"Product created: ID={product_id}, Name={name}, Price=${price}"
        self.notify(message)  # Notify all observers


### 5. Define the FastAPI Application
# Now we can integrate everything into a FastAPI application.

app = FastAPI()
product_service = ProductService()

# Attach observers to the product service
product_service.attach(EmailNotifier())
product_service.attach(LoggingObserver())

@app.post("/products/")
async def create_product(product_id: int, name: str, price: float):
    """Create a new product."""
    product_service.create_product(product_id, name, price)
    return {"message": "Product created successfully", "product_id": product_id, "name":
name, "price": price}


### 6. Running the FastAPI Application
# To run the FastAPI application, save the code above in a file (e.g., `main.py`) and run
it with Uvicorn:
# bash
# uvicorn main:app --reload


### 7. Testing the API
# You can test the API using tools like Postman or Curl. Here’s how you can test it:

# Create a Product-
# POST request to -> http://127.0.0.1:8000/products/:
```

```
# json
# {
#     "product_id": 1,
#     "name": "Laptop",
#     "price": 1200.50
# }
#
```

Explanation

1. Observer Interface: The `Observer` interface defines a method (`update`) that observers must implement to receive notifications.

2. Subject Class: The `Subject` class manages the list of observers, allowing them to be attached or detached and notifying them of changes.

3. Concrete Observers: `EmailNotifier` and `LoggingObserver` are concrete implementations of the `Observer` interface that react to notifications by performing specific actions (sending an email or logging a message).

4. Product Service: The `ProductService` extends the `Subject` class and manages the creation of products. When a product is created, it sends notifications to all attached observers.

Benefits

1. Loose Coupling: Observers and subjects are loosely coupled, allowing for easy addition or removal of observers without affecting the subject.

2. Dynamic Behaviour: Observers can be added or removed at runtime, allowing the application to change its behaviour dynamically based on the observers present.

3. Simplified Event Handling: The Observer Pattern simplifies event handling in an application by providing a clear mechanism for notifying multiple components of changes.

4. Separation of Concerns: It promotes separation of concerns, as the subject is only responsible for notifying observers, while observers handle their specific responses.

Circuit Breaker Pattern is a resilience design pattern used to handle and prevent cascading failures in distributed systems. It monitors external service calls, allowing a system to "trip" and halt calls if a service repeatedly fails, switching to a failure state. This pattern typically has three states:

1. **Closed**: Calls flow as normal, and failures are counted.

2. **Open**: After reaching a failure threshold, the circuit "trips" to stop requests, preventing further failures.

3. **Half-Open**: After a cooldown period, a limited number of test calls are allowed to check if the service has recovered.

Below is a Python example of the Circuit Breaker pattern, simulating a service call and handling potential failures.

```
### Example Code

import time
import random

class CircuitBreaker:
    def __init__(self, failure_threshold: int, recovery_time: int):

        self.failure_threshold = failure_threshold
        self.recovery_time = recovery_time
        self.fail_count = 0
        self.last_failed_time = 0
        self.state = "CLOSED"


    def call(self, func, *args, **kwargs):
        """Executes the service call and implements circuit breaker logic."""

        if self.state == "OPEN":
            if (time.time() - self.last_failed_time) >= self.recovery_time:
                self.state = "HALF-OPEN"
            else:
                print("Circuit open! Call blocked.")
                return None

        try:
            result = func(*args, **kwargs)
            self.reset()    # Reset fail count if call succeeds
            return result
        except Exception as e:
            self.record_failure()
            print(f"Call failed: {e}")
            if self.fail_count >= self.failure_threshold:
                self.trip()
            return None
    def record_failure(self):
        self.fail_count += 1
        self.last_failed_time = time.time()


    def trip(self):
        self.state = "OPEN"
        print("Circuit tripped to OPEN! Blocking further calls.")


    def reset(self):
        self.state = "CLOSED"
        self.fail_count = 0


# Mock service to simulate success and failure
def unreliable_service():
    if random.choice([True, False, False]):  # Simulate 33% chance of success
        return "Service response"
    else:
        raise Exception("Service failure")


# Usage
circuit_breaker = CircuitBreaker(failure_threshold=3, recovery_time=10)


for i in range(10):
    print(f"Attempt {i+1}:")
    response = circuit_breaker.call(unreliable_service)
    if response:
        print(f"Received response: {response}")
    time.sleep(2)
```

Explanation

1. Initialization: The `CircuitBreaker` is initialized with a failure threshold and a recovery time (in seconds).

2. State Transitions:

- Closed: Calls are allowed, and if they fail, the failure count increments.

- Open: If failures exceed the threshold, the circuit trips to "Open," blocking further calls until the recovery period has passed.

- Half-Open: After the cooldown period, a limited number of test calls are allowed to see if the service has recovered. If successful, it resets to "Closed"; if it fails, it goes back to "Open."

3. Error Handling: The `call` method encapsulates the logic, recording failures, transitioning states, and retrying the service call based on the circuit's state.

Benefits

Improved Resilience: Protects the system from repeatedly attempting calls to a failing service, reducing load and dependency issues.

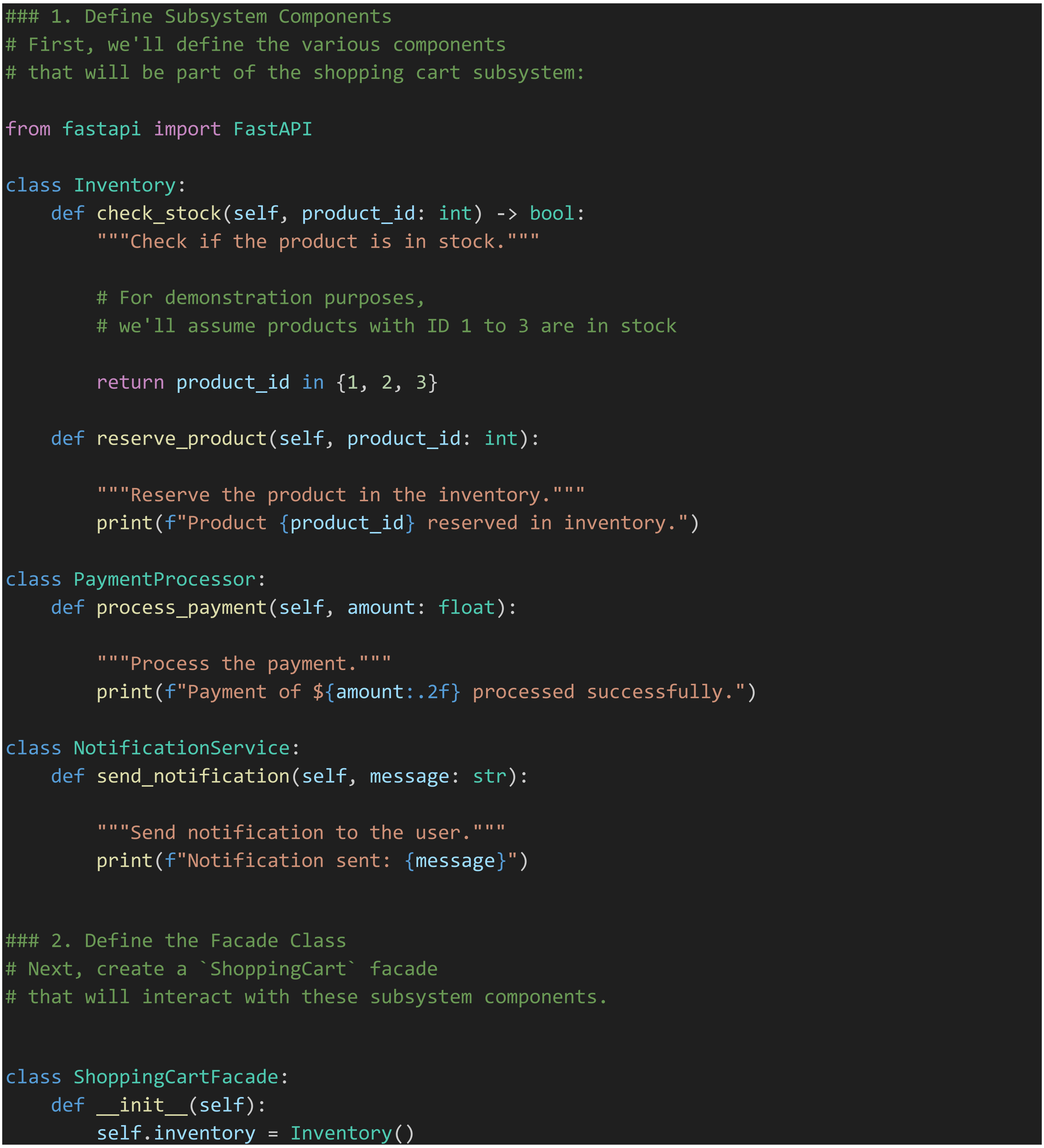
Fail-Fast Mechanism: Quickly fails requests when the circuit is "Open," conserving resources.

Automated Recovery: Moves to "Half-Open" after a cooldown period to allow recovery.

Facade Pattern is a structural design pattern that provides a simplified interface to a complex subsystem or a set of interfaces in a system. It allows clients to interact with the subsystem in a more straightforward way, hiding the complexity and details of the subsystem components. This pattern is useful for simplifying interactions and providing a unified interface for various functionalities.

Scenario

Let's create a simple FastAPI application that demonstrates the Facade Pattern. In this example, we'll implement a `**ShoppingCart**` facade that simplifies the interactions between various components involved in the shopping cart operations, such as inventory management and payment processing.



```
### 1. Define Subsystem Components
# First, we'll define the various components
# that will be part of the shopping cart subsystem:

from fastapi import FastAPI

class Inventory:
    def check_stock(self, product_id: int) -> bool:
        """Check if the product is in stock."""

        # For demonstration purposes,
        # we'll assume products with ID 1 to 3 are in stock

        return product_id in {1, 2, 3}

    def reserve_product(self, product_id: int):

        """Reserve the product in the inventory."""
        print(f"Product {product_id} reserved in inventory.")

class PaymentProcessor:
    def process_payment(self, amount: float):

        """Process the payment."""
        print(f"Payment of ${amount:.2f} processed successfully.")

class NotificationService:
    def send_notification(self, message: str):

        """Send notification to the user."""
        print(f"Notification sent: {message}")


### 2. Define the Facade Class
# Next, create a `ShoppingCart` facade
# that will interact with these subsystem components.


class ShoppingCartFacade:
    def __init__(self):
        self.inventory = Inventory()
```

```python
        self.payment_processor = PaymentProcessor()
        self.notification_service = NotificationService()

    def checkout(self, product_id: int, quantity: int, price: float):
        """Handle the checkout process."""

        if self.inventory.check_stock(product_id):
            self.inventory.reserve_product(product_id)
            total_amount = quantity * price
            self.payment_processor.process_payment(total_amount)
            self.notification_service.send_notification(
                f"Order placed for product {product_id}. Total: ${total_amount:.2f}")

            return {"message": "Checkout successful", "total_amount": total_amount}
        else:
            return {"message": "Product is out of stock."}


### 3. Define the FastAPI Application
# Now, we can integrate everything into a FastAPI application.

app = FastAPI()
shopping_cart = ShoppingCartFacade()

@app.post("/checkout/")
async def checkout(product_id: int, quantity: int, price: float):
    """Checkout endpoint."""
    result = shopping_cart.checkout(product_id, quantity, price)
    return result


### 4. Running the FastAPI Application
# To run the FastAPI application,
# save the code and run it with Uvicorn:
# bash
# uvicorn main:app --reload

# ### 5. Testing the API
# You can test the API using tools like Postman or Curl.
# Here’s how you can test it:
# Checkout Request
# POST request to `http://127.0.0.1:8000/checkout/:

# json
# {
#     "product_id": 1,
#     "quantity": 2,
#     "price": 150.00
# }
```

Explanation

1. Subsystem Components: The `Inventory`, `PaymentProcessor`, and `NotificationService` classes represent the different parts of the shopping cart subsystem. Each has its own responsibilities.

2. Facade Class: The `ShoppingCartFacade` class provides a simplified interface for clients (like the FastAPI application) to perform the checkout process. It internally manages the interactions with the subsystem components, hiding their complexity.

3. Unified Interface: Clients interact with the `ShoppingCartFacade` without needing to understand the details of how the inventory check, payment processing, and notifications are implemented.

Benefits

1. Simplified Interface: The Facade Pattern provides a simplified interface for complex subsystems, making it easier for clients to interact with them.

2. Encapsulation of Complexity: It hides the complexities of the subsystem components, allowing clients to focus on higher-level operations.

3. Loose Coupling: Clients are loosely coupled to the subsystem, as they interact only with the facade instead of multiple components.

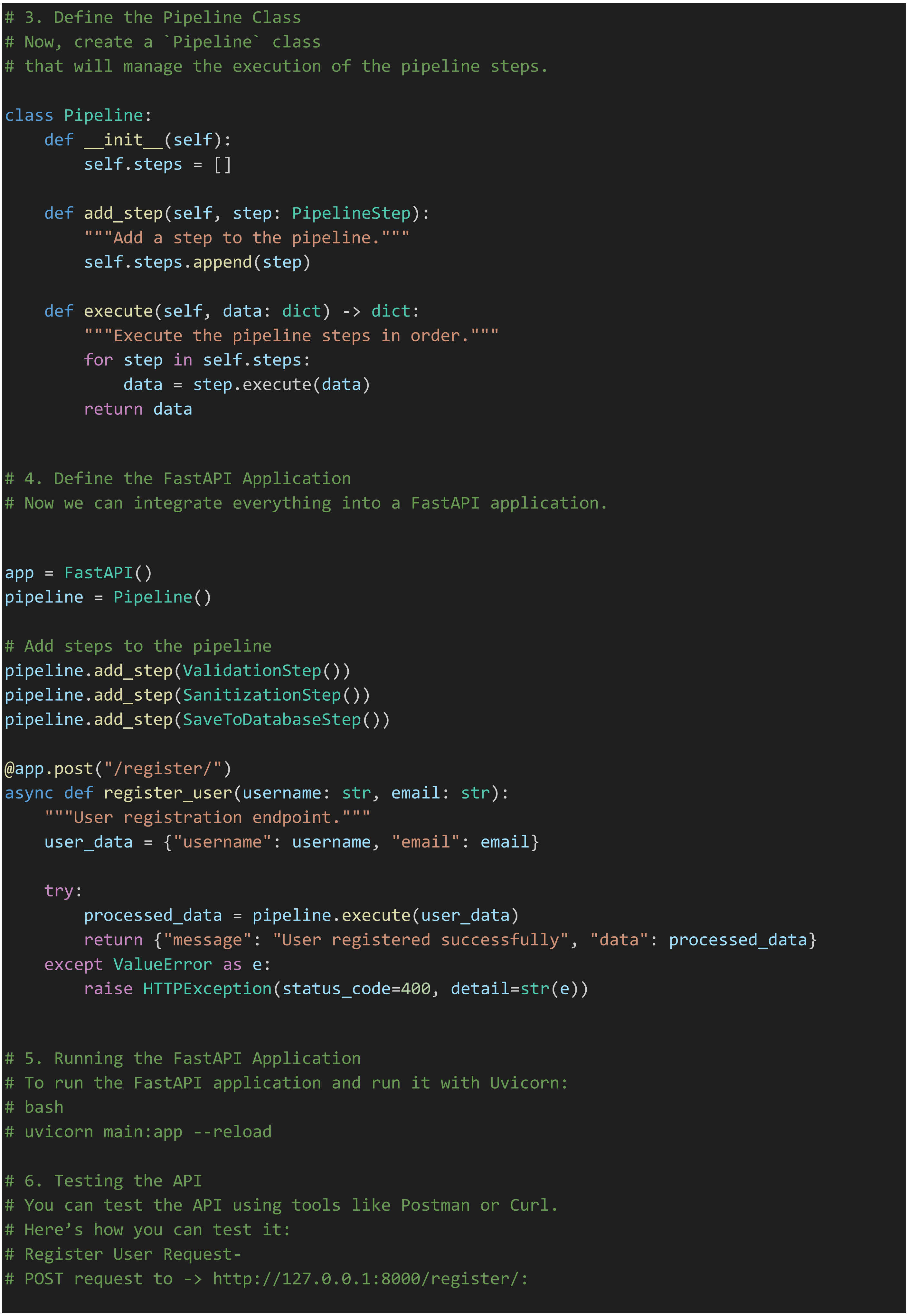
4. Ease of Maintenance: Changes to the internal workings of the subsystem can be made without affecting the clients, making the system easier to maintain.

Pipeline Pattern is a **behavioural** design pattern that allows you to create a series of processing steps (or stages) where the output of one step is passed as input to the next. This pattern is useful for scenarios where data needs to be processed in multiple steps, each potentially transforming the data in some way. It promotes a clean separation of concerns, as each step can focus on a specific task.

Scenario

Let’s create a simple FastAPI application that demonstrates the Pipeline Pattern for processing user input through several steps. In this example, we'll implement a pipeline for processing a user registration request, which includes validation, sanitization, and saving to a database.

```python
# 1. Define the Pipeline Step Interface
# First, define an interface for pipeline steps
# that will be used in the processing pipeline.

from abc import ABC, abstractmethod
from fastapi import FastAPI, HTTPException

class PipelineStep(ABC):
    @abstractmethod
    def execute(self, data: dict) -> dict:
        """Process the input data and return the modified data."""
        pass


# 2. Define Concrete Pipeline Steps
# Now define concrete pipeline steps for validation,
# sanitization, and saving the user data.


class ValidationStep(PipelineStep):
    def execute(self, data: dict) -> dict:
        """Validate the user data."""
        if "username" not in data or "email" not in data:
            raise ValueError("Missing username or email.")
        return data

class SanitizationStep(PipelineStep):
    def execute(self, data: dict) -> dict:
        """Sanitize the user data."""
        data["username"] = data["username"].strip()
        data["email"] = data["email"].strip().lower()
        return data

class SaveToDatabaseStep(PipelineStep):
    def execute(self, data: dict) -> dict:
        """Simulate saving user data to a database."""
        # In a real application, you would perform the database operation here.

        print(f"User {data['username']} saved to database.")
        return data
```



```
# 3. Define the Pipeline Class
# Now, create a `Pipeline` class
# that will manage the execution of the pipeline steps.

class Pipeline:
    def __init__(self):
        self.steps = []

    def add_step(self, step: PipelineStep):
        """Add a step to the pipeline."""
        self.steps.append(step)

    def execute(self, data: dict) -> dict:
        """Execute the pipeline steps in order."""
        for step in self.steps:
            data = step.execute(data)
        return data


# 4. Define the FastAPI Application
# Now we can integrate everything into a FastAPI application.


app = FastAPI()
pipeline = Pipeline()

# Add steps to the pipeline
pipeline.add_step(ValidationStep())
pipeline.add_step(SanitizationStep())
pipeline.add_step(SaveToDatabaseStep())

@app.post("/register/")
async def register_user(username: str, email: str):
    """User registration endpoint."""
    user_data = {"username": username, "email": email}

    try:
        processed_data = pipeline.execute(user_data)
        return {"message": "User registered successfully", "data": processed_data}
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))


# 5. Running the FastAPI Application
# To run the FastAPI application and run it with Uvicorn:
# bash
# uvicorn main:app --reload

# 6. Testing the API
# You can test the API using tools like Postman or Curl.
# Here’s how you can test it:
# Register User Request-
# POST request to -> http://127.0.0.1:8000/register/:
```

```
# json
# {
#     "username": "  JohnDoe ",
#     "email": "  JohnDoe@Example.com  "
# }
```

Explanation

1. Pipeline Step Interface: The `PipelineStep` interface defines a method (`execute`) that concrete steps must implement to process the data.

2. Concrete Steps: `ValidationStep`, `SanitizationStep`, and `SaveToDatabaseStep` are concrete implementations of the `PipelineStep` interface that handle specific tasks.

ValidationStep: Checks for the presence of required fields.

SanitizationStep: Cleans up the input data.

SaveToDatabaseStep: Simulates saving the user data to a database.

3. Pipeline Class: The `Pipeline` class manages the addition and execution of steps, allowing the steps to be executed in order.

4. FastAPI Application: The FastAPI application uses the `Pipeline` to process user registration requests, handling validation, sanitization, and saving data.

Benefits

1. Separation of Concerns: Each processing step focuses on a specific task, promoting clean and maintainable code.

2. Flexibility: New steps can be added, or existing steps modified without changing the overall structure of the pipeline.

3. Reusability: Steps can be reused across different pipelines or applications, enhancing code reuse.

4. Easier Debugging: Isolating steps makes it easier to debug and test each part of the processing logic.

Event-Driven Pattern is a **behavioural** design pattern that promotes asynchronous communication between components in a system. In this pattern, events are used to signal state changes, and event listeners (or subscribers) react to these events. This approach is beneficial for decoupling components, allowing for more flexible and maintainable architectures.

Scenario

Let’s create a simple FastAPI application that demonstrates the Event-Driven Pattern. In this example, we'll implement an event system for user registration, where an event is emitted when a user successfully registers, and multiple listeners (subscribers) react to that event, such as sending a welcome email and logging the registration.

```
# 1. Define the Event Class
# First, create an event class to represent user registration events.

from fastapi import FastAPI, HTTPException

class UserRegisteredEvent:
    def __init__(self, username: str, email: str):
        self.username = username
        self.email = email


# 2. Define the Event Bus
# Now, we’ll create an event bus that manages
# the subscription and publication of events.

class EventBus:
    def __init__(self):
        self.subscribers = {}

    def subscribe(self, event_type: str, subscriber):
        """Subscribe a listener to an event type."""

        if event_type not in self.subscribers:
            self.subscribers[event_type] = []
        self.subscribers[event_type].append(subscriber)

    def publish(self, event):
        """Publish an event to all subscribed listeners."""

        event_type = type(event).__name__
        for subscriber in self.subscribers.get(event_type, []):
            subscriber.handle(event)


# 3. Define Event Handlers (Listeners)
# Now, implement event handlers
# that will respond to user registration events.


class WelcomeEmailHandler:
```

```python
    def handle(self, event: UserRegisteredEvent):
        """Send a welcome email to the user."""

        print(f"Sending welcome email to {event.email}")

class RegistrationLogger:
    def handle(self, event: UserRegisteredEvent):
        """Log the registration event."""

        print(f"User registered: {event.username} with email {event.email}")


# 4. Define the FastAPI Application
# Now we can integrate everything into a FastAPI application.


app = FastAPI()
event_bus = EventBus()

# Create event handlers
welcome_email_handler = WelcomeEmailHandler()
registration_logger = RegistrationLogger()

# Subscribe handlers to the UserRegisteredEvent
event_bus.subscribe(UserRegisteredEvent.__name__, welcome_email_handler)
event_bus.subscribe(UserRegisteredEvent.__name__, registration_logger)

@app.post("/register/")
async def register_user(username: str, email: str):
    """User registration endpoint."""
    # Simulate user registration logic here

    if not username or not email:
        raise HTTPException(status_code=400,
                            detail="Username and email are required.")

    # Create a UserRegisteredEvent
    event = UserRegisteredEvent(username=username, email=email)

    # Publish the event
    event_bus.publish(event)

    return {"message": "User registered successfully",
            "username": username, "email": email}


# 5. Running the FastAPI Application
# To run the FastAPI application, save the code and run it with Uvicorn:
# bash
# uvicorn main:app --reload

# 6. Testing the API
# You can test the API using tools like Postman or Curl.
# Here’s how you can test it:
```

```
# Register User Request-
# POST request to -> http://127.0.0.1:8000/register/:

# json
# {
#     "username": "JaneDoe",
#     "email": "janedoe@example.com"
# }
```

Explanation

1. Event Class: The `UserRegisteredEvent` class represents the data associated with a user registration event, encapsulating the username and email.

2. Event Bus: The `EventBus` class manages the subscription of listeners to specific events and the publishing of events to these listeners.

3. Event Handlers: `WelcomeEmailHandler` and `RegistrationLogger` are listeners that react to the `UserRegisteredEvent`. They define how to handle the event when it occurs.

- WelcomeEmailHandler: Sends a welcome email to the user.

- RegistrationLogger: Logs the registration details.

4. FastAPI Application: The FastAPI application exposes an endpoint for user registration, which creates an event and publishes it to the event bus, triggering the subscribed listeners.

Benefits

1. Loose Coupling: Components are decoupled from one another, as they interact through events rather than direct calls. This improves modularity and maintainability.

2. Scalability: The system can easily scale, allowing new event handlers to be added without modifying existing code.

3. Asynchronous Processing: Event-driven architectures can support asynchronous processing, improving performance and responsiveness.

4. Flexibility: New behaviours can be added by simply adding new event handlers without changing the core logic.

Application Requirements Recap

Your application involves the following:

1. **Data Processing**: Load CSV data into PySpark dataframes, process it based on business logic, and then write it to JSON.

2. **Data Serialization and Persistence**: Insert JSON data into PostgreSQL.

3. **API Layer**: Provide a REST API for business users to access processed data.

Evaluation of Recommended Design Patterns

1. Builder Pattern for Data Processing Layer

   - Usefulness: Useful if you need to construct complex objects or configurations in a step-by-step manner.

   - Recommendation: Keep it if your data processing logic involves multiple configurations or complex objects.

2. **Factory Pattern for Data Processing Layer**

   - Usefulness: Useful for creating objects without specifying the exact class of the object that will be created, especially if there are multiple types of processing strategies.

   - Recommendation: Consider keeping it if your processing involves different types of data sources or processing strategies that can benefit from encapsulation.

3. **Strategy Pattern for Data Processing Layer**

   - Usefulness: Useful for defining a family of algorithms, encapsulating each one, and making them interchangeable.

   - Recommendation: Keep it if your application needs to apply different processing strategies dynamically.

4. **Repository Pattern for Data Serialization and Persistence Layer**

   - Usefulness: Useful for abstracting the data access logic, providing a clear separation of concerns.

   - Recommendation: Keep it as it helps with database interactions and improves testability.

5. **Data Mapper Pattern for Data Serialization and Persistence Layer**

   - Usefulness: Useful if you want to keep your domain model separate from the database schema.

- Recommendation: Consider removing it if you are using an ORM like SQLAlchemy, which already provides this functionality.

6. **Adapter Pattern for Data Serialization and Persistence Layer**

   - Usefulness: Useful for integrating incompatible interfaces or systems.

   - Recommendation: Remove it if there are no integrations or interface incompatibilities.

7. **Controller Pattern for API Layer**

   - Usefulness: Useful for separating the application logic from the request handling logic in your API.

   - Recommendation: Keep it as it enhances organization and readability.

8. **Dependency Injection Pattern for API Layer**

   - Usefulness: Useful for managing dependencies and enhancing testability.

   - Recommendation: Keep it for better manageability of services and components.

9. **DTO (Data Transfer Object) Pattern for API Layer**

   - Usefulness: Useful for transferring data between processes and reducing the number of method calls.

   - Recommendation: Keep it if you need to structure the data exchanged between the API and clients.

10. **Singleton Pattern for Cross-Cutting**

    - Usefulness: Useful for ensuring that a class has only one instance and provides a global point of access to it.

    - Recommendation: Consider removing it unless you have specific components that require a single instance (e.g., configuration manager).

11. **Observer Pattern for Cross-Cutting**

    - Usefulness: Useful for defining a one-to-many dependency between objects.

    - Recommendation: Consider removing it unless you have components that need to react to specific events.

12. **Circuit Breaker Pattern for Cross-Cutting**

- Usefulness: Useful for handling failures gracefully in distributed systems, particularly for external service calls.

- Recommendation: Consider removing it unless you are integrating with unreliable external services.

13. **Facade Pattern for Application Core Design Patterns**

- Usefulness: Useful for providing a simplified interface to a complex subsystem.

- Recommendation: Keep it if you have a complex set of operations that can benefit from a unified interface.

14. **Pipeline Pattern for Application Core Design Patterns**

- Usefulness: Useful for processing data through a series of steps where the output of one step is the input for the next.

- Recommendation: Keep it if your data processing involves multiple stages that need to be executed sequentially.

15. **Event-Driven Pattern for Application Core Design Patterns**

- Usefulness: Useful for decoupling components and enabling asynchronous communication.

- Recommendation: Consider removing it unless you have events that need to trigger actions across different components.

Summary of Recommendations

Keep:

Builder, Factory, Strategy, Repository, Controller, Dependency Injection, Facade, Pipeline, Singleton.

Remove:

DTO, Data Mapper (if using ORM), Adapter, Singleton, Observer (unless necessary), Circuit Breaker (unless necessary), Event-Driven (unless necessary).