Developing a RESTful API with FastAPI [Part 3]

> **Objective: : In this lecture, we will learn how to use Pydantic for data serialization and input validation in FastAPI, focusing on preparing clean, structured data for machine learning APIs. By the end, you'll be able to serialize inputs, format predictions, and handle nested data structures for seamless API communication.**

Data Serialization with Pydantic

Data serialization is crucial when building APIs, especially for machine learning, as it helps convert data between formats like Python objects, JSON, or dictionaries. This is especially useful when:

- Accepting requests from API users (input serialization).
- Returning predictions or results in a structured format (output serialization).

What is Serialization?

Serialization is the process of converting a data structure (like a Python object) into a format that can be easily stored or transmitted (e.g., JSON). Deserialization is the reverse—converting incoming data (e.g., JSON) into Python objects.

Why is Serialization Important in ML APIs?

1. **Input Validation:** Ensure incoming data (like feature vectors) matches expected formats.
2. **Output Standardization:** Consistently format predictions for easy consumption by clients.
3. **Interoperability:** Serialize data into a universally accepted format (e.g., JSON) for API communication.

Serialization with Pydantic

Pydantic provides methods to handle serialization:

- **`.dict()`**: Converts the model instance to a Python dictionary.
- **`.json()`**: Converts the model instance to a JSON string.

- **Custom Field Serialization**: Use Pydantic's `@field_serializer` to customize how fields are serialized.

Concrete ML-Focused Examples

Example 1: Serializing Input Features

Suppose you have an ML model that predicts house prices based on features like `size`, `location`, and `number of rooms`.

script_name: example1.py

```
from pydantic import BaseModel

class HouseFeatures(BaseModel):
    size: float  # In square meters
    location: str
    num_rooms: int

# Input data
features = HouseFeatures(size=120.5, location="Downtown",
num_rooms=3)

# Serialize to dictionary
print(features.dict())

# Serialize to JSON
print(features.json())
```

Output:

```
{'size': 120.5, 'location': 'Downtown', 'num_rooms': 3}
{"size": 120.5, "location": "Downtown", "num_rooms": 3}
```

Explanation:

- `.dict()` converts the model instance into a dictionary suitable for internal processing.
- `.json()` prepares the data for transmission over APIs.

Example 2: Serializing Model Predictions

After processing input features, your ML model generates predictions. Let’s say it predicts a `price` for the house.

script_name: example2.py

```
class Prediction(BaseModel):
    price: float  # Predicted price in dollars

# Model prediction
prediction = Prediction(price=250000.75)

# Serialize prediction
print(prediction.dict())
print(prediction.json())
```

Output:

```
{'price': 250000.75}
{"price": 250000.75}
```

Explanation: This allows you to send the prediction as a JSON response to the API client.

Example 3: Nested Serialization

In many ML scenarios, data structures can be complex, involving nested objects. For instance, consider a case where we return both predictions and metadata about the request.

script_name: example3.py

```
from datetime import datetime

class Metadata(BaseModel):
    request_id: str
    timestamp: datetime

class Response(BaseModel):
    prediction: float
    metadata: Metadata

# Nested data
response = Response(
    prediction=250000.75,
    metadata=Metadata(request_id="12345",
timestamp=datetime.now())
)

# Serialize response
print(response.dict())
print(response.json())
```

Output:

```
{
  'prediction': 250000.75,
  'metadata': {
    'request_id': '12345',
    'timestamp': '2025-01-22T12:34:56.789000'
  }
}
{"prediction": 250000.75, "metadata": {"request_id": "12345",
"timestamp": "2025-01-22T12:34:56.789000"}}
```

Explanation:

- Nested serialization works seamlessly with `.dict()` and `.json()`.
- This is perfect for creating structured responses for ML APIs.

Custom Field Serialization

You can customize how fields are serialized using Pydantic's `@field_serializer`. This is useful when ML models produce raw outputs that need transformation (e.g., probabilities formatted as percentages).

Example: Formatting Probabilities

Imagine your ML model predicts probabilities for different classes. You want to return these as percentages.

script_name: example4.py

```
from pydantic import BaseModel, field_serializer

class Prediction(BaseModel):
    class_name: str
    probability: float  # Raw probability between 0 and 1

    @field_serializer("probability")
    def format_probability(cls, value):
        return f"{value * 100:.2f}%"  # Convert to percentage

# Prediction output
prediction = Prediction(class_name="Cat", probability=0.87)

# Serialize prediction
print(prediction.dict())
print(prediction.json())
```

Output:

```
{'class_name': 'Cat', 'probability': '87.00%'}
{"class_name": "Cat", "probability": "87.00%"}
```

Explanation: The `@field_serializer` decorator ensures probabilities are automatically formatted during serialization.

Example: Redacting Sensitive Information

If your API includes sensitive information (e.g., user IDs), you might want to exclude it during serialization.

script_name: example5.py

```
class UserPrediction(BaseModel):
    user_id: int  # Sensitive information
    prediction: float

    class Config:
        fields = {'user_id': {'exclude': True}}  # Exclude
user_id from serialization

# User prediction
user_prediction = UserPrediction(user_id=101, prediction=95.5)

# Serialize prediction
```

```
print(user_prediction.dict())
print(user_prediction.json())
```

Output:

```
{'prediction': 95.5}
{"prediction": 95.5}
```

Explanation: Using the `Config` class, we exclude `user_id` from serialized output, safeguarding sensitive data.

Real-World Application: Serializing Input and Output for ML API

Let’s put it all together in a simple ML API workflow:

1. **Input Serialization:** Validate and process the input features.
2. **Output Serialization:** Format predictions for the client.

script_name: example6.py

```
class InputFeatures(BaseModel):
    size: float
    location: str
    num_rooms: int

class OutputPrediction(BaseModel):
    price: float
    request_id: str
    timestamp: datetime

# Input from client
input_data = {"size": 120.5, "location": "Downtown", "num_rooms":
3}

# Deserialize input
features = InputFeatures(**input_data)
print("Validated Input:", features.dict())

# Generate prediction
prediction = OutputPrediction(
    price=250000.75,
    request_id="12345",
    timestamp=datetime.now()
)

# Serialize output
print("Serialized Output:", prediction.json())
```

Output:

```
Validated Input: {'size': 120.5, 'location': 'Downtown',
'num_rooms': 3}
Serialized Output: {"price": 250000.75, "request_id":
"12345", "timestamp": "2025-01-22T12:34:56.789000"}
```

Key Takeaways

- `.dict()` and `.json()` make serialization seamless.
- **Custom field serializers** allow for tailored data formatting.
- Pydantic ensures validation and serialization are tightly integrated, reducing errors in ML workflows.

Exercise

Challenge 1: E-Commerce Product Catalog

Create a script called challenge1.py containing your solutions and push to the "fastapi-project/part_3" folder in your git branch.

Task:

1. Define a `Product` model with the following fields:
    - `name` (string): The product name.
    - `price` (float): The product price in USD.
    - `category` (string): The product category (e.g., "Electronics", "Books").
    - `discount` (float, optional): Discount on the product, default is `0.0`.
2. Serialize the following product data into JSON:

```
{
    "name": "Laptop",
    "price": 1500.0,
    "category": "Electronics"
}
```

3. Add a custom serializer to ensure the `price` field always includes two decimal places (e.g., `1500.00`).

Challenge 2: Movie Recommendation API

Create a script called challenge2.py containing your solutions and push to the "fastapi-project/part_3" folder in your git branch.

Task:

1. Define a `Movie` model with these fields:
    - `title` (string): The movie title.
    - `genre` (list of strings): Genres the movie belongs to.
    - `rating` (float): The movie's rating between 0 and 10.

- `release_date` (date): When the movie was released.

2. Write a function to:
    - Deserialize JSON input to create a `Movie` object.
    - Serialize the object into a dictionary format.
3. Add a custom serializer to format the `release_date` as `DD-MM-YYYY`.

Example Input:

```
{
    "title": "Inception",
    "genre": ["Sci-Fi", "Thriller"],
    "rating": 9.0,
    "release_date": "2010-07-16"
}
```

Challenge 3: Weather Data API

Create a script called challenge3.py containing your solutions and push to the "fastapi-project/part_3" folder in your git branch.

Task:

1. Define a `WeatherReport` model with these fields:
    - `city` (string): The name of the city.
    - `temperature` (float): The temperature in Celsius.
    - `humidity` (integer): Percentage humidity.
    - `condition` (string): Weather condition (e.g., "Sunny", "Rainy").
2. Extend the model to include nested data for the time of the report:
    - Add a `ReportMetadata` model with:
        - `timestamp` (datetime): When the report was generated.
        - `source` (string): Source of the weather data (e.g., "NOAA").
3. Serialize a `WeatherReport` object (with nested metadata) into JSON.
4. Add a custom serializer to convert `temperature` from Celsius to Fahrenheit in the serialized output.

Example Input:

```
{
    "city": "New York",
    "temperature": 20.0,
    "humidity": 60,
```

```
    "condition": "Sunny",
    "metadata": {
        "timestamp": "2025-01-22T15:30:00",
        "source": "NOAA"
    }
}
```

Challenge 4: Fitness Tracker API

Create a script called challenge4.py containing your solutions and push to the "fastapi-project/part_3" folder in your git branch.

Task:

1. Define a `Workout` model with:
    - `type` (string): Type of workout (e.g., "Running", "Cycling").
    - `duration` (float): Duration in minutes.
    - `calories_burned` (float): Calories burned.
2. Add a validator to ensure `duration` is greater than 0.
3. Serialize a batch of workouts into JSON format.
4. Add a custom serializer to convert `calories_burned` into kilocalories (1 calorie = 0.001 kcal).

Example Input:

```
[
    {"type": "Running", "duration": 45.0, "calories_burned": 300},
    {"type": "Cycling", "duration": 60.0, "calories_burned": 500}
]
```

Expected Output:

```
[
    {"type": "Running", "duration": 45.0, "calories_burned": "0.30 kcal"},
    {"type": "Cycling", "duration": 60.0, "calories_burned": "0.50 kcal"}
]
```