FastAPI: обработка форм; JSON, схемы Pydantic | Я.Шпора

Обработках данных из форм

Для работы с полями формы необходима библиотека python-multipart

```
pip install python-multipart==0.0.5
```

Поля формы обозначаются при помощи класса `Form`, он ведёт себя аналогично классам `Path` и `Query`:

```
from fastapi import FastAPI, Form

app = FastAPI()


@app.post('/login')
def login(
        username: str = Form(...),
        password: str = Form(...),
):
    ...
```

Для передачи файлов используется класс `File`, аннотированный как `UploadFile`.

```
# form.py
from fastapi import FastAPI, File, Form, UploadFile

app = FastAPI()


@app.post('/login')
def login(
        username: str = Form(...),
        password: str = Form(...),
        some_file: UploadFile = File(...)
):
    ...
```

Обработка JSON, полученного в теле запроса

Эндпоинты FastAPI могут несколькими способами обработать JSON, полученный в теле POST-запроса из формы:

1. В функции, обрабатывающей запрос, можно аннотировать параметр как коллекцию — список, кортеж или множество, не используя классы `Query` или `Form`.

   ```
   @app.post("/product")
   def product(prices: list[float]):
   ```

```
    ...
```

2. При помощи класса `Body`, который ведёт себя аналогично уже рассмотренным классам `Path`, `Query`, `Form`. Можно почитать про него в документации.
3. При помощи классов из библиотеки Pydantic — обычно их называются «схемами».

Библиотека Pydantic

Pydantic — библиотека для десериализации и валидации данных, основанная на работе стандартных аннотаций Python. Эта библиотека устанавливается вместе с библиотекой FastAPI.

Все классы Pydantic наследуются от класса `pydantic.BaseModel`. Опишем Pydantic-класс (схему) `Product`, на основе которой FastAPI будет выполнять сериализацию и валидацию данных.

```
class Product(BaseModel):
    title: str
    prices: list[float]
    in_stock: bool = True
    discount: Optional[float]


@app.post("/product")
# Вместо множества параметров в функцию передаём объект класса Person,
# он содержит все необходимые поля.
```

```
def product(product: Product) -> dict[str, str]:
    ...
```

Теперь при POST-запросе к эндпоинту */product* JSON-объект из тела запроса будет валидирован и десериализован в Python-объект класса `Product`.

Валидация полей в схемах Pydantic

В схемах Pydantic дополнительные свойства полей описываются при помощи класса `Field`:

```
# Дополнительно к BaseModel импортируем класс Field.
from pydantic import BaseModel, Field

...

class Product(BaseModel):
    title: str = Field(
        ..., min_length=2, max_length=20,
        title='Полное название', description='Можно вводить в любом регистре'
    )
    prices: list[float] = Field(..., gt=4, le=100_000)
    in_stock: bool = Field(True, alias='is-staff')
    discount: Optional[float] = Field(None, gt=2, lt=51)
```

В моделях Pydantic можно описать подкласс Config, в котором прописываются настройки сразу для всех полей или для класса в целом.

```
class Product(BaseModel):
    ...

    class Config:
        title = 'Товар'
        min_anystr_length = 2  # Ограничение минимальной длины строки в классе.
```

В схеме Pydantic можно описать собственные валидаторы — для отдельных полей или для всей модели сразу.

Для валидации отдельных полей функция-валидатор оборачивается декоратором `@validator` из модуля `pydantic`. В декоратор передаётся аргумент — имя того поля, которое нужно проверить: `@validator('имя_поля')`. В случае успешной валидации функция должна вернуть `value` — значение валидируемого поля.

Пример валидатора, который не пропускает товар с названием «кукуруза»:

```
# Дополнительно импортируем декоратор для валидатора.
from pydantic import BaseModel, Field, validator
...

class Product(BaseModel):
    title: str = Field(
        ..., min_length=2, max_length=20,
        title='Полное название', description='Можно вводить в любом регистре'
```

```
    )
    ...

    @validator('title')
    def title_cannot_be_corn(cls, value:str):
        if value.lower() == 'кукуруза':
            raise ValueError('Название товара не может быть "кукуруза"')
        return value
```

Валидаторами Pydantic можно проверить условие, которое должно выполняться сразу в нескольких полях. Для этого используется декоратор `@root_validator`.

```
from pydantic import BaseModel, Field, root_validator, validator

...


class Product(BaseModel):
    ...

    @root_validator
    # К названию параметров функции-валидатора нет строгих требований.
    # Первым аргументом передается класс, вторым — словарь со значениями всех полей.
```

```
    def using_different_languages(cls, values):
        ...
```

Если в проекте одновременно применяются и валидаторы отдельных полей, и `@root_validator`, то по умолчанию сначала выполняются валидаторы отдельных полей, а потом — корневые.

Примеры запросов для документации

В описание моделей Pydantic можно добавить собственные примеры запросов — и они отобразятся в документации Swagger. Примеры добавляются во вложенном классе `Config` модели с помощью атрибута `schema_extra`.

```
class Product(BaseModel):
    ...
    class Config:
        ...
        schema_extra = {
            'example': {
                'title': 'Тостер',
                'prices': [10.18, 42, 229.01 ],
                'in_stock': True,
                'discount': 9.99
            }
        }

     ...
```

Значение для тестового запроса можно задать и для каждого поля отдельно, передав в класс `Field` атрибут `example`:

```
discount: Optional[float] = Field(None, gt=4, le=99, example=20.5)
```

Атрибут `example` можно передать и в классы `Path` и `Query`:

```
title: str = Path(..., example='Ivan')
```

При этом значение атрибута `example` будет автоматически отображаться в поле ввода в Swagger.

Настройки проекта FastAPI

Pydantic используется для задания настроек FastAPI-проекта. Обычно они размещаются в файле *app/core/config.py*:

```
from pydantic import BaseSettings


class Settings(BaseSettings):
    # Задаём имя приложения:
    app_title: str = 'First and Best Project'

    class Config:
        # Задаём имя файла с переменными окружения:
        env_file = '.env'
```

```
settings = Settings()
```

Файлы окружения (env-файлы) используются для хранения конфигурационных параметров и переменных окружения, которые могут быть использованы в приложениях и скриптах.

Название параметров в env-файле должно совпадать с названием атрибутов класса настроек без учета регистра. Например, `app_title` в классе настроек и `APP_TITLE` в env-файле.

Импорт и применение установленных настроек в *app/main.py*:

```
from app.core.config import settings


app = FastAPI(title=settings.app_title)
```