# Harvest x i4talent

## MLOps

Homework

i4talent DATA SCIENCE

# Homework for upcoming training

In order to get the most out of the upcoming MLOps training, make sure you are comfortable working with Fast API and Pydantic.


Data validation using Python type hints

# Fast API: An REST API development Framework for Python.

Fast API is an alternative to Django and Flask with a few advantages. We will be using Fast API to develop end points to use and monitor models during the training.

# Pydantic: data validation and parsing made easy!

Pydantic is a Python library designed to simplify data validation and parsing.

By leveraging Python's type annotations, Pydantic allows you to define data models with validation rules, enabling you to ensure data integrity and handle data from various sources effortlessly.

Fast API is built of Pydantic, so it is a good idea to understand a bit more about Pydantic before learning to work with Fast API.

# Pydantic

# The BaseModel

BaseModel is main way to interact with Pydantic. We create our own Pydantic model by defining a class that inherits from BaseModel and specifying some (typed) attributes.

We can dan create instances of this class the same way we would for normal Python classes.

The main takeaway here is that Pydantic actually enforces datatypes for attributes: Passing a string to id will actually fail the initialization instead of just rolling with it like normal Python would (resulting probably in errors later).

```python
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name = 'Jane Doe'
```

# Data validation and custom validators

Sometimes just using normal Python typing is not enough. There are a lot of custom validation options however.

Pydantic provide validation rules out of the box, such as required fields, minimum and maximum values, regular expressions and more.

Python 3.7 and above

```python
from pydantic import BaseModel, ValidationError, validator


class UserModel(BaseModel):
    name: str
    username: str
    password1: str
    password2: str

    @validator('name')
    def name_must_contain_space(cls, v):
        if ' ' not in v:
            raise ValueError('must contain a space')
        return v.title()

    @validator('password2')
    def passwords_match(cls, v, values, **kwargs):
        if 'password1' in values and v != values['password1']:
            raise ValueError('passwords do not match')
        return v

    @validator('username')
    def username_alphanumeric(cls, v):
        assert v.isalnum(), 'must be alphanumeric'
        return v
```

# Data parsing and serialization

Pydantic makes it possible to parse JSON objects into Python objects and the other way around; serialize Pydantic models back into JSON, making it easy to interact with external APIs for example.

# Models can be nested

Pydantic allows you to create nested models, allowing you to represent complex data structures and enforce validation at multiple levels.

When data is parsed by a Pydantic model with attributes that are also Pydantic models, data assigned to those attributes are thus parsed by the inner model as well.

For an example, see: https://docs.pydantic.dev/latest/concepts/models/#nested-models

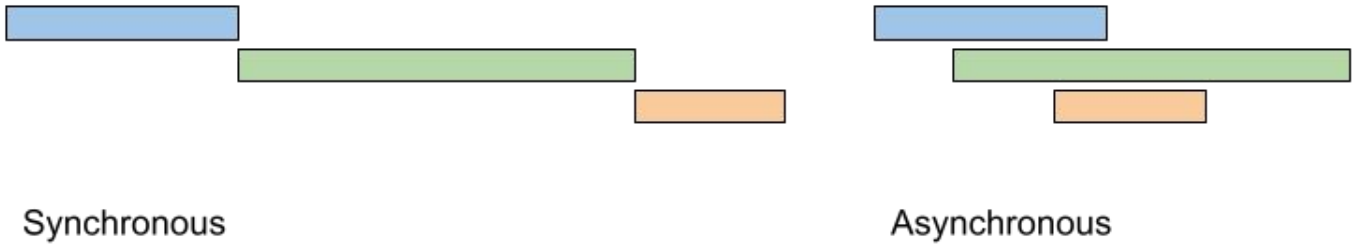
# Model configuration

Behaviour of pydantic can be controlled via the model_config attribute or Config Class (deprecated).

See https://docs.pydantic.dev/latest/concepts/config/

# Fast API

# FastAPI is asynchronous




It is able to handle requests asynchronously, meaning work on a request can start and sometimes finish, even if a previous request is not yet done. This makes it much more efficient, especially if there is waiting involved like for example getting data from a database.

# Developing with FastAPI is easy and efficient

Automatic OpenAPI documentation: Swagger page is generated for you.

Hot reloading: You can have it update the API when a file changes without restarting

Data validation with Pydantic

# You only need a few lines of code

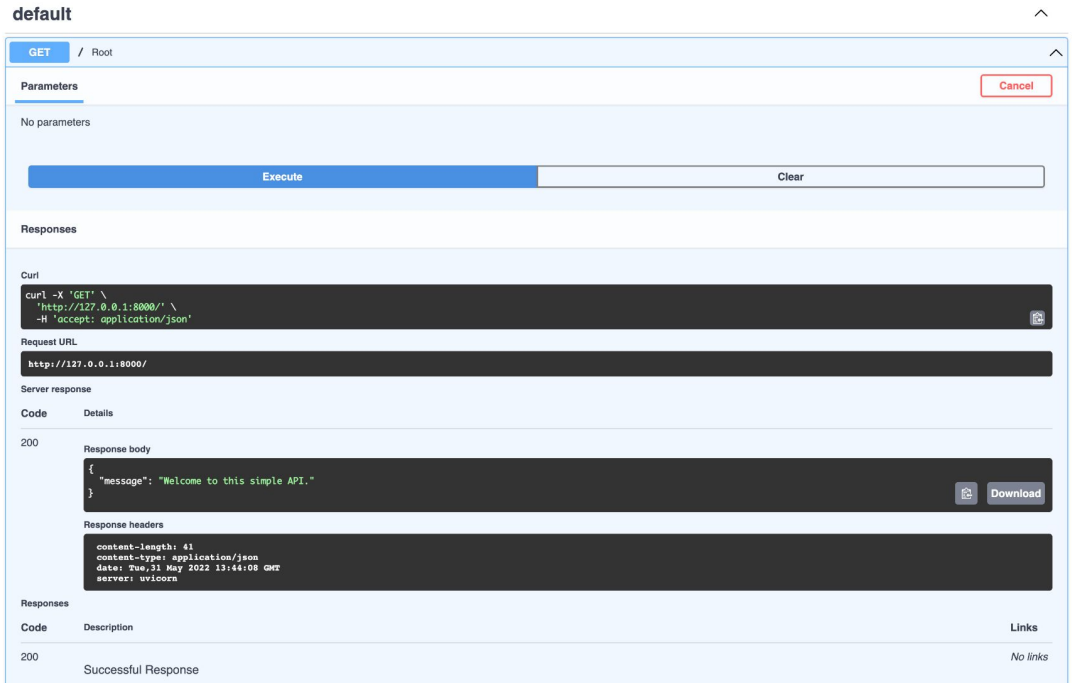
```python
from fastapi import FastAPI


app = FastAPI()


@app.get("/")
async def root():
    return {"message": "Welcome to the simple API."}

```

Run with uvicorn, a lightweight asynchronous webserver:

*uvicorn main:app --reload*

i4talent DATA SCIENCE

# Interactive documentation at http://127.0.0.1:8000/docs#



# *Routers* easily enable a more structured approach

user_data_router.py

```python
from fastapi import APIRouter, Depends, Path

from model_api.dataloaders import DataLoader
from model_api.dependencies import get_data
from model_api.api_classes import ViewResponseModel

router = APIRouter(prefix="/users",
                   tags=["users"],
                   dependencies=[Depends(get_data)],
                   responses={404: {"description": "Not Found"}})


@router.get("/")
async def get_users(data: DataLoader = Depends(get_data)):
    return {"message": data.get_users()}


@router.get("/{user}", response_model=list[ViewResponseModel])
async def get_user_view_data(user: int = Path(..., description="The user ID", ge=1),
                             data: DataLoader = Depends(get_data)):
    return data.query_data(user=str(user)).to_dict("records")

```

main.py

```python
from fastapi import FastAPI

from model_api.routers import user_data_router


app = FastAPI()

app.include_router(user_data_router.router)


@app.get("/")
async def root():
    return {"message": "Welcome to this simple API."}

```

https://fastapi.tiangolo.com/tutorial/bigger-applications/

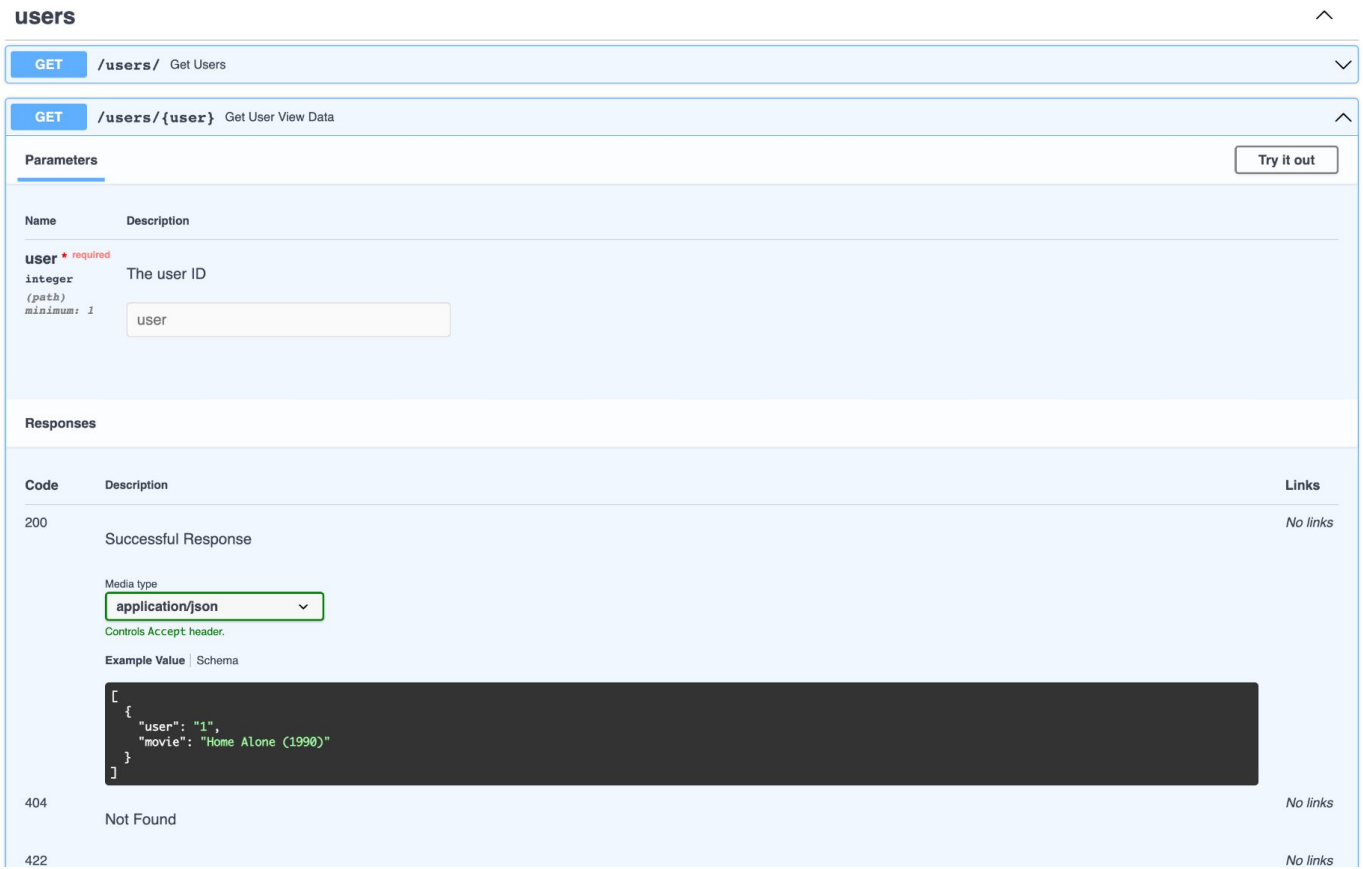
# *Dependencies* allow for easy sharing of data, objects and functions between endpoints

dependencies.py

```python
from model_api.dataloaders import DataLoader
from model_api.predictors import TensorflowPredictor


predictor = TensorflowPredictor(model_path="./model/index")
data_loader = DataLoader(predictor_model=predictor)


async def get_predictor() -> TensorflowPredictor:
    return predictor


async def get_data() -> DataLoader:
    return data_loader
```

```python
@router.get("/")
async def get_users(data: DataLoader = Depends(get_data)):
    return {"message": data.get_users()}
```

https://fastapi.tiangolo.com/tutorial/dependencies/


users
GET /users/ Get Users
GET /users/{user} Get User View Data
Parameters
Try it out
Name
Description
user * required
integer
(path)
minimum: 1
The user ID
user
Responses
Code
Description
Links
200
Successful Response
No links
Media type
application/json
Controls Accept header.
Example Value | Schema
[
{
"user": "1",
"movie": "Home Alone (1990)"
}
]
404
Not Found
No links
422
No links

# The template you are about to work on already has some things taken care of

It contains modules for loading user and movie data. More on what we will use the data for will become clear during the training, but you can try to figure it out if you are curious.

On first time startup of the API, a tensorflow dataset will be downloaded and processed into a local SQLite database. This is a one time setup, but it can take a while.

You will use this database to create some endpoints to practice working with Fast API. If you are having trouble finishing all assignments, don't worry too much. A version of the code with the endpoints implemented will be provided during the training.

However, do get some practice in, as this will make the rest of the training go a lot more smoothly.

# Some links

Repository: https://github.com/i4talent/fastapi_harvest

Follow the instructions in README.md to start the API.

Pydantic documentation: https://docs.pydantic.dev/latest/

Fast API documentation: https://fastapi.tiangolo.com/

# Assignments

Extend the *movie_data_router* in the *routers* subpackage. Make sure the path *"/movies/"* returns the full contents of the movies table in the database.

Take a look at *src/model_api/dataloaders/dataloader.py* to figure out what functions you can use to retrieve data.

# Assignments

Add a path to movie_data_router with a numeric parameter n. The path must return n random movies.

Bonus: Make the endpoint require n to be a positive integer.

# Assignments

Add a path to movie_data_router with two query parameters min_rating and max_rating, that allows users to query movies based on whether the average rating falls between these two values. Return at least the title and average rating.

Bonus: Make the endpoint require the query arguments to be between 0 and 5.

You can use the following SQL Query to get the average ratings:

```
select M.movie_title, avg(R.user_rating) as rating
from movies M join ratings R
on M.movie_id=R.movie_id
group by M.movie_title
having rating >= ? and rating <= ?
```

# Assignments

Users should be able to add additional ratings to the database when they watched a new movie.
Add a path to user_data_router to send new view data as a request body to the API and save it to the database. A method for adding new data is already present in *dataloader.py*. To validate the incoming data, a Pydantic model should be created.

Bonus: Make sure supplied users and movies are validated with the database. You can use a custom Pydantic validator for this.