

# **GET and POST requests for AI**

**DEPLOYING AI INTO PRODUCTION WITH FASTAPI**



**Matt Eckerle**

Software and Data Engineering Leader

# Our instructor



## Matt Eckerle

- Software and Data Engineering Leader
- Enterprise Data Manager at Inari
- Using FastAPI for ML since 2019

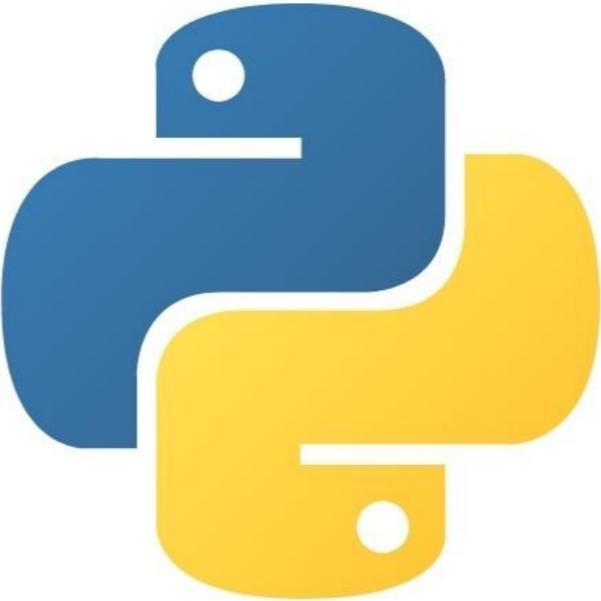
# Course overview

- FastAPI fundamentals
- Request handling and integration
- Input validation and security
- Create and maintain production-ready APIs



# Before we start

- ✓ Basic Python: functions, classes, modules, data structures



# Before we start

- ☑ Basic python: functions, classes, modules, data structures
- ☑ HTTP & REST API concepts



# Before we start

✓ Basic Python: functions, classes, modules, data structures

✓ HTTP & REST API concepts

✓ Using the FastAPI framework

- Handling GET and POST requests
- Using Pydantic models

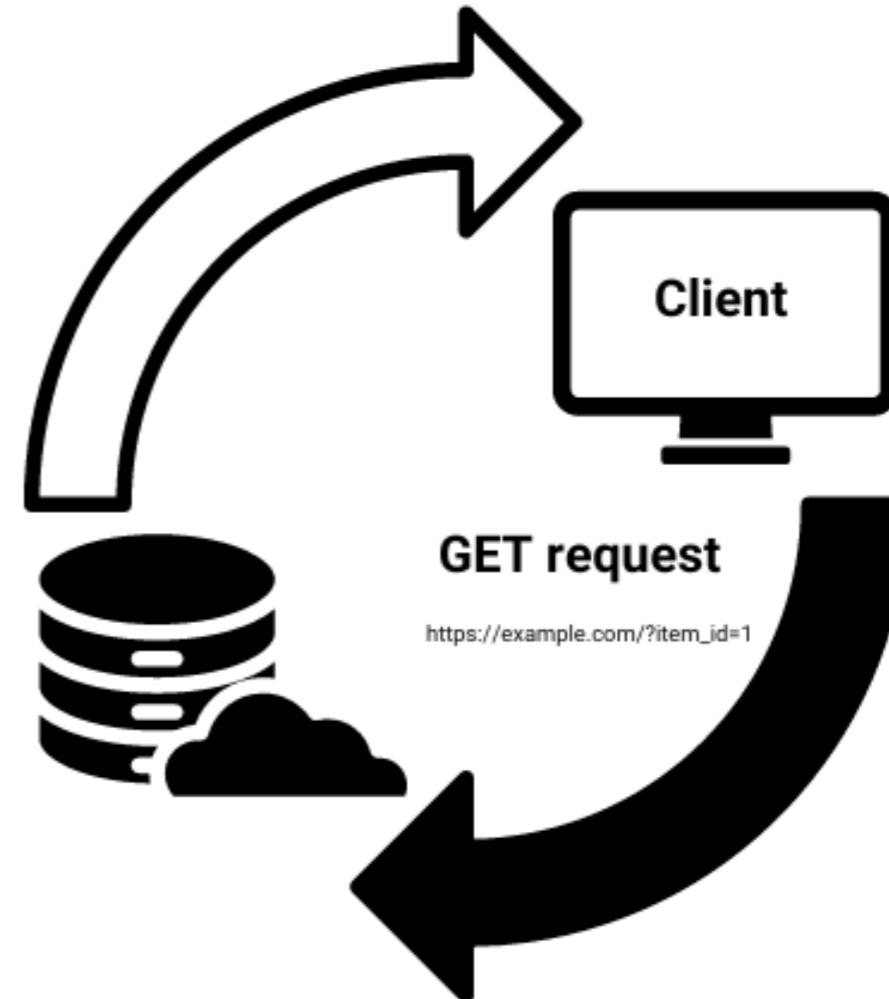
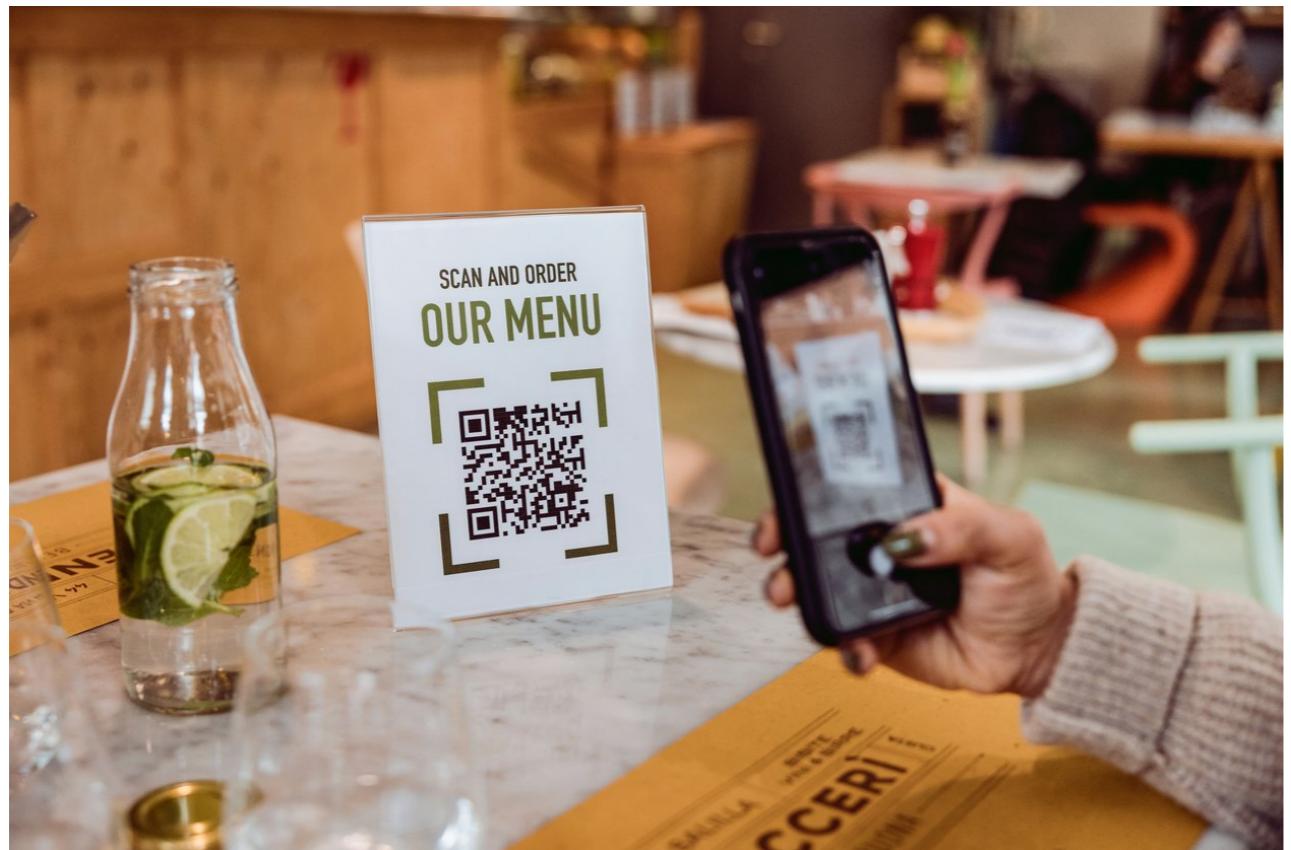
✓ Machine Learning basics



# GET requests explained

- Used to retrieve data
- Path parameter - information in URL
- Doesn't change server state

GET `https://example.com/?item_id=1`



# Implementing GET with path parameters

```
from fastapi import FastAPI

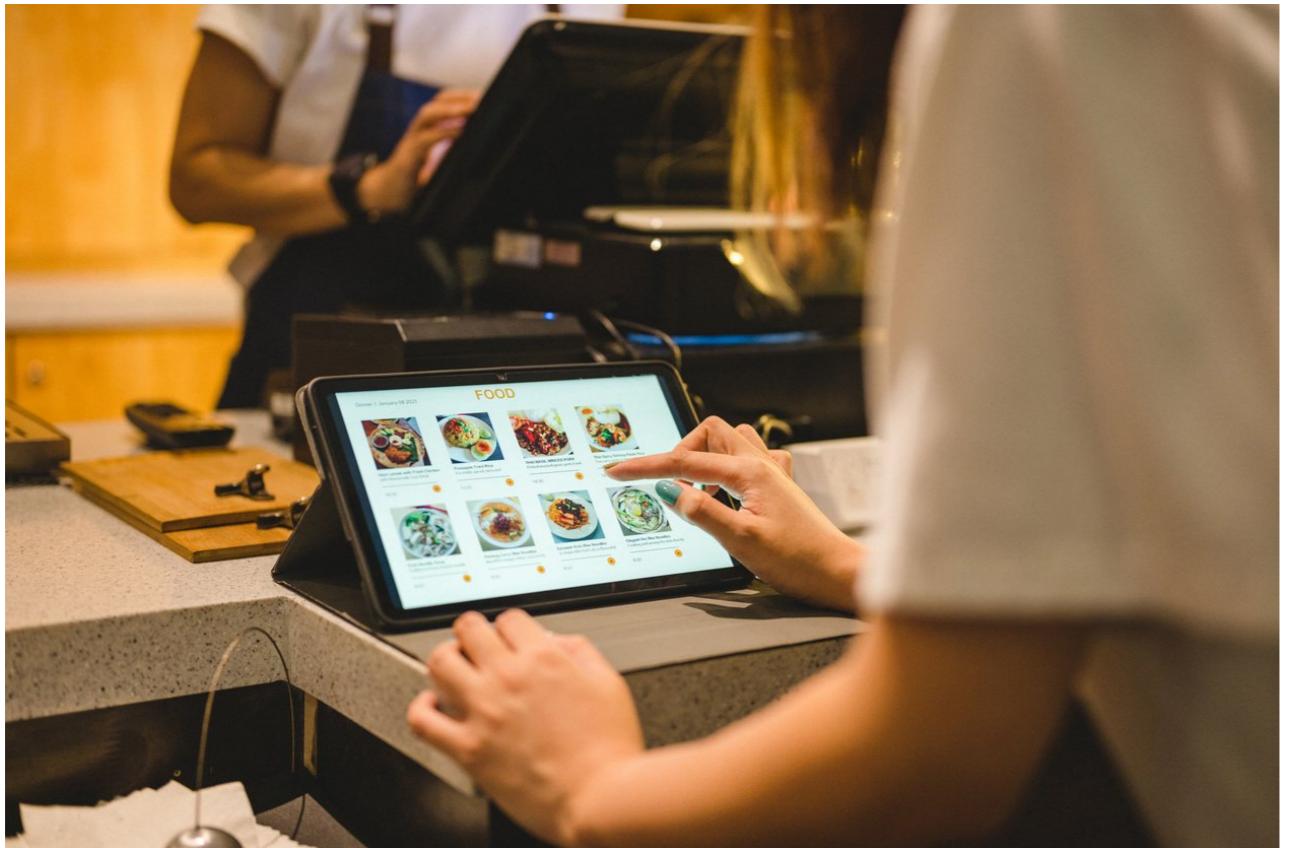
app = FastAPI()

@app.get("/item/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

- `@app.get` decorator defines endpoint.
- {item\_id} is a path parameter.
- Type hint (int) for auto validation.

# POST requests explained

- Used to send data to server
- Data in request body (usually JSON)
- Can change server state



POST <https://example.com>



# Implementing POST with JSON data

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()
db = {}

class Item(BaseModel):
    name: str
    price: float

@app.post("/items", status_code=201)
def create_item(item: Item):
    db[item.name] = item.model_dump()
    return {"message":
            f"Created {item.name}"}
```

- Pydantic model defines data structure
- `@app.post` for POST endpoint
- Automatic JSON parsing and validation

# HTTP status codes

- 200 OK: Default for success
- 404 Not Found: Resource not found
- 201 Success: Object created



# Raising HTTP exceptions

```
from fastapi import FastAPI, HTTPException
app = FastAPI()

@app.get("/item/{item_id}")
async def read_item(item_id: int):
    if item_id == 4242: #An invalid order number
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id}

@app.post("/items")
async def create_item(item: Item):
    # Simulating item creation
    return {"message": f"Created {item.name}"}, 201
```

# **Let's practice!**

**DEPLOYING AI INTO PRODUCTION WITH FASTAPI**

# FastAPI prediction with a pre-trained model

DEPLOYING AI INTO PRODUCTION WITH FASTAPI



**Matt Eckerle**

Software and Data Engineering Leader

# Setting up the environment

## Required libraries:

- `FastAPI` : framework for building APIs with Python
- `uvicorn` : fast ASGI server that runs Python web apps
- `joblib` : for loading the model

```
from fastapi import FastAPI  
import uvicorn  
import joblib  
# Create the FastAPI app instance  
app = FastAPI()
```

# Loading the pre-trained penguin classifier

- Trained on the Palmer Penguins dataset
- Predicts penguin species based on 4 features: culmen length, culmen depth, flipper length, and body mass
- Output: Adelie, Chinstrap, or Gentoo

```
import joblib

# Load the pre-trained model
model = joblib.load('penguin_classifier.pkl')
# Check data type of model to verify model loading
print(type(model))
```

```
<class 'skLearn.pipeline.Pipeline'>
```

<sup>1</sup> <https://huggingface.co/SIH/penguin-classifier-sklearn>

# Uvicorn

- ASGI (Asynchronous Server Gateway Interface) server
- Built by and for Python

```
uvicorn main:app \
    --host 0.0.0.0 \
    --port 8080
```

```
import uvicorn
uvicorn.run(app,
            host="0.0.0.0",
            port=8080)
```



# Creating the prediction endpoint

```
# FastAPI prediction endpoint  
  
@app.post("/predict")  
  
def predict(culmen_length_mm, culmen_depth_mm,  
            flipper_length_mm, body_mass_g):  
  
    features = [[culmen_length_mm, culmen_depth_mm,  
                flipper_length_mm, body_mass_g]]  
  
    prediction = model.predict(features)[0]  
    return {"predicted_species": prediction}
```

# Running the application

```
if __name__ == "__main__":
    uvicorn.run(
        app,
        host="0.0.0.0",
        port=8080)
```

Save all the code in a python file - `your_api_script.py`

```
$ python3 your_api_script.py
```

```
INFO:     Started server process [276]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
```

# Testing the API

```
curl \  
  -X POST "http://localhost:8080/predict" \  
  -H "Content-Type: application/json" \  
  -d '{"culmen_length_mm": 39.1,  
        "culmen_depth_mm": 18.7,  
        "flipper_length_mm": 181,  
        "body_mass_g": 3750}'
```

```
{  
  "prediction": "Adelie",  
  "confidence": 0.87  
}
```

# **Let's practice!**

**DEPLOYING AI INTO PRODUCTION WITH FASTAPI**

# Request and response models

DEPLOYING AI INTO PRODUCTION WITH FASTAPI



**Matt Eckerle**

Software and Data Engineering Leader

# Request and response structures in APIs

## Request structure

- Data sent by the client to the API
- Typically includes:
  - HTTP method (GET, POST, etc.)
  - Headers
  - Query parameters
  - Request body

```
http://localhost:8000/users/?  
email=john.doe@example.com
```

- `:8000` : The port number. FastAPI typically uses 8000 by default.
- `/users/` : The path to the specific endpoint we're accessing.
- `?` : Indicates the start of the query parameters.
- ``email=john.doe@example.com`` : The query parameter. In this case, we're passing an email address to the endpoint.

# Response structure

- Data sent back by the API to the client
  - JSON response payload with user information
- Typically includes:
  - Status code (200 OK, 404 Not Found, etc.)
  - Headers
  - Response body (data requested or operation result)

```
HTTP/1.1 200 OK
date: Fri, 18 Oct 2024 12:34:56 GMT
server: uvicorn
content-length: 76
content-type: application/json

{
    "username": "johndoe",
    "email": "john.doe@example.com",
    "age": 30
}
```

# Pydantic model

- Creation of User
- Use of Pydantic

```
from pydantic import BaseModel
```

```
class User(BaseModel):  
    username: str  
    email: str  
    age: int
```

- Inherits from BaseModel
- Defines attributes with type annotations
- Automatic validation based on types

# Validation error

```
from pydantic import ValidationError
try:
    invalid_user = User(username="john_doe", email="john.doe@example.com",
                         age="thirty")
    print("Invalid User:", invalid_user)
except ValidationError as e:
    print("Validation Error:", e)
```

```
Validation Error: 1 validation error for User
age
Input should be a valid integer, unable to parse string as an integer
[type=int_parsing, input_value='thirty', input_type=str]
```

# Using Pydantic models in FastAPI

```
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()

class User(BaseModel):
    username: str
    email: str
    age: int

@app.post("/users", response_model=User)
async def create_user(user: User):
    return user
```

- Model used as `response_model`
- Model used as parameter type hint
- Automatic request validation
- De/serialization of request/response
- Generate API docs

# Request and response formats

```
curl -X 'POST' \  
'http://localhost:8000/users/' \  
-H 'accept: application/json' \  
-H 'Content-Type: application/json' \  
-d '{  
    "username": "john_doe",  
    "email": "john.doe@example.com",  
    "age": 30  
}'
```

```
{  
    "username": "john_doe",  
    "email": "john.doe@example.com",  
    "age": 30  
}
```

# **Let's practice!**

**DEPLOYING AI INTO PRODUCTION WITH FASTAPI**