FastAPI

AI Basics

FastAPI Cheat Sheet

```
from fastapi import *
```

#Python

in/sumitkhanna

Python

Follow me on
Sumit Khanna for more updates

FastAPI Comprehensive Cheat Sheet and Guide

Introduction

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. The key features include automatic interactive API documentation, support for data validation, and the ability to handle request and response bodies. This guide aims to provide a detailed overview of FastAPI's capabilities with real-life use cases and examples.

Quick CheatSheet of Functions

Feature/Function	Brief Explanation
`FastAPI`	The main class to create a FastAPI application instance.
`@app.get()`	Define a route to handle GET requests.
`@app.post()`	Define a route to handle POST requests.
`@app.put()`	Define a route to handle PUT requests.
`@app.delete()`	Define a route to handle DELETE requests.
`@app.patch()`	Define a route to handle PATCH requests.
`@app.options()`	Define a route to handle OPTIONS requests.
`@app.head()`	Define a route to handle HEAD requests.
`@app.trace()`	Define a route to handle TRACE requests.
`Path`	Declare a path parameter.
`Query`	Declare a query parameter.

Feature/Function	Brief Explanation
`Body`	Declare a request body parameter.
`Form`	Declare form data parameters.
`File`	Declare file upload parameters.
`HTTPException`	Raise an HTTP error.
`Depends`	Declare dependencies.
`Security`	Security handling functions.
`BackgroundTasks`	Add background tasks to the request.
`Cookie`	Declare cookie parameters.
`Header`	Declare header parameters.
`Response`	Send custom responses.
`Request`	Handle incoming requests.
`APIRouter`	Create route groups.
`status`	HTTP status codes.
`Middleware`	Add middleware to the application.
`CORS`	Handling Cross-Origin Resource Sharing.
`WebSocket`	Define WebSocket endpoints.

FastAPI Application

FastAPI Instance

```
from fastapi import FastAPI

app = FastAPI()
```

The `FastAPI` instance is the main application. You create an instance of it and use it to define your routes and other configurations.

Route Handlers

GET Request

```
@app.get("/")
def read_root():
    return {"message": "Hello World"}
```

The `@app.get()` decorator defines a route to handle GET requests. In this case, it returns a simple JSON response.

POST Request

```
@app.post("/items/")
def create_item(item: dict):
    return item
```

The `@app.post()` decorator defines a route to handle POST requests. The example demonstrates how to handle request bodies.

PUT Request

```
@app.put("/items/{item_id}")
def update_item(item_id: int, item: dict):
    return {"item_id": item_id, **item}
```

The `@app.put()` decorator defines a route to handle PUT requests. This is used for updating resources.

DELETE Request

```
@app.delete("/items/{item_id}")
def delete_item(item_id: int):
    return {"item_id": item_id}
```

The `@app.delete()` decorator defines a route to handle DELETE requests. It deletes a resource by `item_id`.

Path Parameters

```
@app.get("/users/{user_id}")
def read_user(user_id: int):
    return {"user_id": user_id}
```

Path parameters are declared using curly braces in the route path and are automatically converted to the specified type.

Query Parameters

```
from fastapi import Query

@app.get("/items/")
def read_items(q: str = Query(None, min_length=3)):
    return {"q": q}
```

Query parameters are optional and can be declared with default values using the `Query` class.

Request Body

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

@app.post("/items/")
def create_item(item: Item):
    return item
```

Request bodies are declared using Pydantic models. These models provide data validation and serialization.

Form Data

```
from fastapi import Form

@app.post("/login/")
def login(username: str = Form(...), password: str = Form(...)):
    return {"username": username}
```

Form data is handled using the `Form` class. This is typically used for HTML forms.

File Uploads

```python
from fastapi import File, UploadFile

@app.post("/uploadfile/")
def create_upload_file(file: UploadFile = File(...)):
    return {"filename": file.filename}
```

File uploads are handled using the `File` and `UploadFile` classes.

HTTP Exceptions

```python
from fastapi import HTTPException

@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id}
```

HTTP exceptions can be raised using the `HTTPException` class to return specific HTTP status codes and error messages.

Dependencies

```python
from fastapi import Depends

def common_parameters(q: str = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
def read_items(commons: dict = Depends(common_parameters)):
    return commons
```

Dependencies can be declared using the `Depends` class, allowing for shared logic between multiple route handlers.

Security

```
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.get("/users/me")
def read_users_me(token: str = Depends(oauth2_scheme)):
    return {"token": token}
```

Security dependencies can be implemented using the `Security` module, such as OAuth2 authentication.

Background Tasks

```
from fastapi import BackgroundTasks

def write_log(message: str):
    with open("log.txt", "a") as log:
        log.write(message)

@app.post("/send-notification/")
def send_notification(email: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, email)
    return {"message": "Notification sent"}
```

Background tasks can be added to a request using the `BackgroundTasks` class.

Cookie Parameters

```
from fastapi import Cookie

@app.get("/items/")
def read_items(ads_id: str = Cookie(None)):
    return {"ads_id": ads_id}
```

Cookie parameters can be declared using the `Cookie` class.

Header Parameters

```python
from fastapi import Header

@app.get("/items/")
def read_items(user_agent: str = Header(None)):
    return {"User-Agent": user_agent}
```

Header parameters can be declared using the `Header` class.

Custom Responses

```python
from fastapi.responses import JSONResponse

@app.get("/items/{item_id}")
def read_item(item_id: str):
    return JSONResponse(content={"item_id": item_id}, status_code=200)
```

Custom responses can be sent using classes from the `responses` module.

Handling Requests

```python
from fastapi import Request

@app.post("/items/")
async def create_item(request: Request):
    json_body = await request.json()
    return json_body
```

Incoming requests can be accessed and handled using the `Request` class.

Routing

```python
from fastapi import APIRouter

router = APIRouter()

@router.get("/users/", tags=["users"])
def read_users():
    return [{"username": "user1"}, {"username": "user2"}]

app.include_router(router)
```

Routing can be organized using the `APIRouter` class to create modular route groups.

HTTP Status Codes

```
from fastapi import status

@app.post("/items/", status_code=status.HTTP_201_CREATED)
def create_item(item: Item):
    return item
```

HTTP status codes can be used from the `status` module to set response codes.

Middleware

```
from starlette.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Middleware can be added to the application to handle cross-cutting concerns like CORS.

WebSockets

```
from fastapi import WebSocket

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")
```

WebSocket endpoints can be defined to handle real-time communication.

Advanced Functions and Usage

Dependency Injection with Classes

```
class CommonQueryParams:
    def __init__(self, q: str = None, skip

: int = 0, limit: int = 100):
        self.q = q
        self.skip = skip
        self.limit = limit

@app.get("/items/")
def read_items(commons: CommonQueryParams = Depends(CommonQueryParams)):
    return commons
```

Dependencies can be injected using classes, allowing for more complex shared logic.

Response Models

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

@app.post("/items/", response_model=Item)
def create_item(item: Item):
    return item
```

Response models can be used to automatically serialize and validate responses.

Handling Form and File Data

```
from fastapi import Form, File, UploadFile

@app.post("/upload/")
async def create_upload_file(file: UploadFile = File(...), description: str = Form(...)):
    return {"filename": file.filename, "description": description}
```

Combining form data and file uploads in a single request.

Handling Different Content Types

```
from fastapi.responses import HTMLResponse

@app.get("/html", response_class=HTMLResponse)
def get_html():
    return """
    <html>
        <head>
            <title>FastAPI</title>
        </head>
        <body>
            <h1>Hello, FastAPI</h1>
        </body>
    </html>
    """
```

Returning different content types like HTML using `response_class`.

CORS Settings

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://example.com"],
    allow_credentials=True,
    allow_methods=["GET"],
    allow_headers=["*"],
)
```

Configuring CORS settings to control access from different origins.

Model Inference

```
import tensorflow as tf
from fastapi import File, UploadFile

model = tf.keras.models.load_model('path/to/model.h5')

@app.post("/predict/")
async def predict(file: UploadFile = File(...)):
    contents = await file.read()
    image = tf.image.decode_image(contents)
    image = tf.image.resize(image, [224, 224])
    image = tf.expand_dims(image, 0)
    predictions = model.predict(image)
    return {"predictions": predictions.tolist()}
```

Example of using FastAPI for model inference, handling file uploads, and making predictions.
