FastAPI Cheat Sheet

A comprehensive cheat sheet for FastAPI, a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints.

Core Concepts

Basic Setup

Installation:

```
pip install fastapi uvicorn
```

Basic FastAPI App:

```
from fastapi import FastAPI

app = FastAPI()

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

Running the app:

```
uvicorn main:app --reload
```

Path Parameters

Basic Path Parameter

```
from fastapi import FastAPI

app = FastAPI()

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

Path Order Matters

```
@app.get("/users/me")
async def read_user_me():
    return {"user_id": "the current user"}

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

Query Parameters

Basic Query Parameters

```
from typing import Optional
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
async def read_item(q: Optional[str] = None):
    if q:
        return {"item": q}
    return {"items": "All items"}
```

Required Query Parameters

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: str, needy: str):
    item = {"item_id": item_id, "needy": needy}
    return item
```

Request Body and Data Validation

Request Body

Using Pydantic models to define request body.

```
from typing import Optional
from fastapi import FastAPI
from pydantic import BaseModel

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

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    item_dict = item.dict()
    if item.tax:
        price_with_tax = item.price + item.tax
        item_dict.update({"price_with_tax": price_with_tax})
    return item_dict
```

Data Validation

FastAPI uses Pydantic for data validation. Type hints are automatically validated and serialized.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, validator

class User(BaseModel):
    id: int
    name: str
    signup_ts: Optional[datetime] = None
    friends: List[int] = []

    @validator('id')
    def id_must_be_positive(cls, value):
        if value <= 0:
            raise ValueError('id must be positive')
        return value

app = FastAPI()

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

Handling Errors

HTTP Exceptions

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}

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

Custom Exception Handlers

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

class UnicornException(Exception):
    def __init__(self, name: str):
        self.name = name

app = FastAPI()

@app.exception_handler(UnicornException)
async def unicorn_exception_handler(request: Request, exc: UnicornException):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} did something. There goes a unicorn :("},
    )

@app.get("/unicorns/{unicorn_name}")
async def read_unicorn(unicorn_name: str):
    if unicorn_name == "yolo":
        raise UnicornException(name="Yolo")
    return {"unicorn_name": unicorn_name}
```

Dependencies and Security

Dependency Injection

Using dependencies for reusable logic.

```
from typing import Optional
from fastapi import FastAPI, Depends

app = FastAPI()

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

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

@app.get("/users/")
async def read_users(commons: dict = Depends(common_parameters)):
    return commons
```

Security

OAuth2 with Password (and hashing)

```
from fastapi import Depends, FastAPI,
HTTPException, status
from fastapi.security import
OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext

# Example (simplified)

app = FastAPI()


@app.post("/token", response_model=Token)
async def login_for_access_token(form_data:
OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(form_data.username,
form_data.password)
    if not user:
        raise HTTPException(

status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or
password",
            headers={"WWW-Authenticate":
"Bearer"},
        )
    access_token_expires =
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username},
        expires_delta=access_token_expires
    )
    return {"access_token": access_token,
"token_type": "bearer"}
```

API Keys

```
from fastapi import Depends, FastAPI,
HTTPException
from fastapi.security import APIKeyHeader

X_API_KEY = APIKeyHeader(name="X-API-Key")

async def get_api_key(api_key_header: str =
Depends(X_API_KEY)):
    if api_key_header == "YOUR_API_KEY":
        return api_key_header
    else:
        raise HTTPException(status_code=400,
detail="Invalid API Key")

app = FastAPI()

@app.get("/items/", dependencies=
[Depends(get_api_key)])
async def read_items():
    return [{"item": "Foo"}, {"item": "Bar"}]
```

Advanced Features

Middleware

Adding middleware to process requests and responses.

```
from fastapi import FastAPI, Request
from starlette.middleware.base import
BaseHTTPMiddleware

app = FastAPI()

class CustomMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request,
call_next):
        # Code to execute before the request
        response = await call_next(request)
        # Code to execute after the request
        return response

app.add_middleware(CustomMiddleware)
```

Background Tasks

Running tasks after returning a response

```
from fastapi import
BackgroundTasks, FastAPI

app = FastAPI()

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

log.write(message)

@app.post("/log")
async def
log_message(message: str,
background_tasks:
BackgroundTasks):

background_tasks.add_task
(write_log, message)
    return {"message":
"Message logged in
background"}
```

Using `async` background tasks

```
import asyncio
from fastapi import
BackgroundTasks, FastAPI

app = FastAPI()

async def
write_log(message: str):
    await
asyncio.sleep(1)  #
Simulate some work
    with open("log.txt",
mode="a") as log:

log.write(message)

@app.post("/log")
async def
log_message(message: str,
background_tasks:
BackgroundTasks):

background_tasks.add_task
(write_log, message)
    return {"message":
"Message logged in
background"}
```

File Uploads

Handling file uploads.

```
from typing import List
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")
async def create_file(file: bytes =
File(...)):
    return {"file_size": len(file)}

@app.post("/uploadfiles/")
async def create_upload_files(files:
List[UploadFile] = File(...)):
    return {"filenames": [file.filename for
file in files]}
```