FastAPI Interview Q&A:

1. What is FastAPI, and why is it considered a modern web framework?

- **Answer:** FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. It's considered modern because:
    - It leverages Python type hints for data validation and serialization.
    - It automatically generates interactive API documentation (Swagger UI and ReDoc).
    - It's built on top of Starlette for ASGI and Pydantic for data validation, providing high performance.
    - It has built in dependency injection.

2. Explain how FastAPI utilizes Python type hints.

- **Answer:** FastAPI uses Python type hints for:
    - **Data validation:** It automatically validates incoming request data against the specified types.
    - **Serialization:** It serializes response data into JSON based on the type hints.
    - **Automatic documentation:** It generates API documentation based on the type hints, making it easy to understand and use the API.
    - Example:

Python:
```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

- In this example, `item_id: int` enforces that the `item_id` path parameter must be an integer.

3. What is Pydantic, and how does FastAPI use it?

- **Answer:** Pydantic is a Python library for data validation and settings management using Python type annotations. FastAPI uses Pydantic for:
    - Defining request and response data models.
    - Validating incoming data against these models.
    - Serializing data into JSON.

    - It provides robust data validation, ensuring that the API receives and processes data in the expected format.

4. How do you handle path parameters and query parameters in FastAPI?

- **Answer:**
    - **Path parameters:** Defined within the path itself, enclosed in curly braces `{}`. They are required.
    - **Query parameters:** Passed after the path, separated by a `?`, in the form `key=value`. They are optional by default.
    - Example:

Python:
```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int, q: str = None, skip: int = 0, limit: int = 10):
    return {"item_id": item_id, "q": q, "skip": skip, "limit": limit}
```

- In this example, `item_id` is a path parameter, and `q`, `skip`, and `limit` are query parameters.

5. Explain FastAPI's dependency injection system.

- **Answer:** FastAPI's dependency injection system allows you to:
    - Declare dependencies that your path operations rely on.
    - Automatically resolve and inject these dependencies.
    - Reuse dependencies across multiple path operations.
    - It promotes code reusability, modularity, and testability.
    - Example:

Python:
```
from fastapi import FastAPI, Depends

async def common_parameters(q: str = None, skip: int = 0, limit: int = 10):
    return {"q": q, "skip": skip, "limit": limit}

app = FastAPI()

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

- In this example, `common_parameters` is a dependency that's injected into the `read_items` path operation.

6. How do you handle request bodies in FastAPI?

- **Answer:** Request bodies are handled using Pydantic models. You define a Pydantic model representing the expected structure of the request body and then declare it as a parameter in your path operation.
  - Example:

Python:

```
from fastapi import FastAPI, Body
from pydantic import BaseModel

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

app = FastAPI()

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

7. How do you handle file uploads in FastAPI?

- **Answer:** FastAPI provides `UploadFile` to handle file uploads. You can declare an `UploadFile` parameter in your path operation to receive uploaded files.
  - Example:

Python:

```
from fastapi import FastAPI, UploadFile, File

app = FastAPI()

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

8. What are middleware in FastAPI, and how are they used?

- **Answer:** Middleware are functions that process requests and responses before they reach your path operations or after they leave. They can be used for:
    - Logging.
    - Authentication.
    - CORS handling.
    - Modifying requests or responses.
    - Example:

Python:

```
from fastapi import FastAPI
import time

app = FastAPI()

@app.middleware("http")
async def add_process_time_header(request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    return response
```

9. How do you handle error handling and exceptions in FastAPI?

- **Answer:** FastAPI provides several ways to handle errors:
    - **HTTP exceptions:** Use `HTTPException` to return standard HTTP error responses.
    - **Exception handlers:** Define custom exception handlers to handle specific exceptions.
    - Example:

Python:

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 0:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id}
```

10. How do you test a FastAPI application?

- **Answer:** FastAPI provides `TestClient` for testing. You can use it to send requests to your API and assert the responses.
    - Example:

Python:

```
from fastapi.testclient import TestClient
from .main import app #assuming your app is in main.py

client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}
```

- You would generally use pytest with fastapi.