FastAPI Interview Questions and Answers

1 . What is FastAPI?

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.8+ based on standard Python type hints. It is designed to be easy to use, fast to develop with, and to provide efficient performance. FastAPI is built on top of Starlette and Pydantic.

Key features of FastAPI include :

* **Fast :** As the name suggests, FastAPI is designed to be fast. It leverages Python type hints and asynchronous programming to achieve high performance.

* **Automatic Documentation :** FastAPI generates OpenAPI and JSON Schema documentation automatically based on the Python type hints used in the code. This documentation is interactive and can be explored through Swagger UI and ReDoc.

* **Type Checking :** FastAPI uses Python type hints for request and response validation. This not only helps with automatic documentation but also enables the use of tools like Pydantic for data validation and serialization.

* **Asynchronous Support :** FastAPI fully supports asynchronous programming, allowing you to write asynchronous routes and take advantage of the performance benefits of asynchronous I/O operations.

Quick Links

Interview Questions
S/W Technology
Civil, Mech
ECE, EEE
More Technologies

MCQ (or) Quiz
S/W Technology
Civil, Mech
ECE, EEE
Aeronautical

Example Programs
C Language, C++, Java, PHP, Python

Articles
Marketing Management

Tech Updates
Tech Articles

Tools

Compailers

* **Dependency Injection System :** FastAPI has a powerful dependency injection system that makes it easy to manage dependencies in a clean and organized way.

* **Security Features :** It includes features for handling security, including support for OAuth2, API key authentication, and more.

* **WebSocket Support :** FastAPI supports WebSocket communication, allowing real-time bidirectional communication between clients and the server.

* **Compatibility with Standard Python Type Hints :** FastAPI leverages Python type hints for request and response validation. This provides a clear contract for the API and allows for better tooling support.

* **Compatibility with Other Web Frameworks :** FastAPI is designed to be easy to integrate with other web frameworks, allowing you to use it alongside existing applications or services.

* **Built-in Support for OAuth2 and JWT :** FastAPI provides built-in support for OAuth2 and JSON Web Tokens (JWT) for handling authentication and authorization.

2 . Explain the asynchronous features of FastAPI

FastAPI leverages asynchronous programming to provide high-performance web applications and APIs. Asynchronous programming allows tasks to be executed concurrently without waiting for each to complete, making better use of system resources and improving overall application responsiveness. Here are some key aspects of the asynchronous features in FastAPI:

Async and Await Syntax : FastAPI uses Python's async and await syntax to define asynchronous functions. These functions can be used in route handlers, dependencies, and other parts of the application.

```
from fastapi import FastAPI

app = FastAPI()

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

Asynchronous Request Handlers : FastAPI allows you to define asynchronous request handlers using the async def syntax. This is particularly useful for handling requests concurrently, especially when dealing with I/O-bound operations such as database queries or external API calls.

```
from fastapi import FastAPI

app = FastAPI()
```

Color Picker	HTML
Interest Calculator	C & CPP
EMI Calculator	PHP
Vehicle EMI Calculator	Python

```
@app.get("/async_endpoint")
async def async_endpoint():
    result = await some_async_function()
    return {"result": result}?
```

Dependency Injection with Async Dependencies : FastAPI's dependency injection system supports asynchronous dependencies. Dependencies marked with async def can perform asynchronous operations, such as fetching data from a database or making asynchronous API calls.

```
from fastapi import Depends, FastAPI

app = FastAPI()

async def get_some_data():
    # Asynchronous operations here
    return {"data": "Some asynchronous data"}

@app.get("/")
async def read_data(data: dict = Depends(get_some_data)):
    return data?
```

Background Tasks : FastAPI provides a mechanism called BackgroundTasks that allows you to run asynchronous functions in the background after a response has been sent to the client. This is useful for performing non-blocking tasks after handling a request.

```
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("/send-notification/{email}")
async def send_notification(
    email: str, background_tasks: BackgroundTasks
):
    message = f"message to {email}"
```

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

WebSocket Support : FastAPI supports asynchronous WebSocket communication. You can define WebSocket routes using asynchronous functions, enabling real-time bidirectional communication between clients and the server.

```
from fastapi import FastAPI, WebSocket

app = FastAPI()

@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}")
```

By embracing asynchronous programming, FastAPI is able to handle a large number of simultaneous connections efficiently, making it suitable for building scalable and performant web applications and APIs.

3 . How do you define a route in FastAPI?

In FastAPI, routes are defined using decorators. For example, to define a route that returns a list of users, you would use the `@get decorator` :

```
@get(“/users”)
def list_users():
return [{“username”: “jane”}, {“username”: “joe”}]?
```

4 . Can you explain the key differences between FastAPI and Flask and why you would choose FastAPI over Flask?

FastAPI and Flask are both Python web frameworks, but they differ significantly. FastAPI is built on Starlette for web routing and Pydantic for data validation, which allows it to offer high performance compared to Flask. It also supports modern functionalities like async and await keywords.

FastAPI automatically generates interactive API documentation using OpenAPI and JSON Schema standards, a feature not present in Flask. This makes it easier to test and debug APIs during development.

FastAPI's use of Python type hints leads to better editor support, error checking, and refactoring capabilities. In contrast, Flask lacks this feature, making code maintenance more challenging.

The choice between FastAPI and Flask depends on the project requirements. For applications requiring high performance, modern asynchronous features, automatic API documentation, or extensive use of data validation, FastAPI would be the preferred choice over Flask due to its advanced features and superior performance.

5 . What is the role of Pydantic in FastAPI?

Pydantic plays a crucial role in FastAPI by providing data validation and settings management using Python type annotations. It ensures that incoming data matches the expected types, reducing runtime errors. Pydantic models define how requests and responses should be structured, enabling automatic request body parsing, validation, serialization, and documentation.

6 . What are some advantages of using FastAPI over Flask?

FastAPI and Flask are both popular web frameworks for building web applications and APIs in Python, but they have different design philosophies and features. Here are some advantages of using FastAPI over Flask:

* **Performance :** FastAPI is designed to be highly performant, thanks to its use of asynchronous programming. Asynchronous support allows FastAPI to handle a large number of concurrent requests efficiently, making it well-suited for applications with high performance requirements. Flask, on the other hand, is synchronous by default.

* **Automatic API Documentation :** FastAPI automatically generates interactive API documentation based on the Python type hints used in your code. This documentation is accessible through Swagger UI and ReDoc, making it easy for developers to understand and test the API without additional manual documentation efforts. Flask, while having documentation tools, may require more manual intervention for similar features.

* **Type Hints and Validation** : FastAPI leverages Python type hints to automatically validate requests and responses. This not only helps in catching errors early in the development process but also improves code readability. Flask, being a more traditional framework, doesn't enforce type hints as strictly.

* **Dependency Injection :** FastAPI provides a built-in dependency injection system that makes it easy to manage dependencies in a clean and organized way. This can be particularly useful for handling reusable components and ensuring proper separation of concerns. Flask, while having support for extensions, may not provide as built-in and structured a system for dependency injection.

* **Asynchronous Database Operations :** FastAPI's asynchronous support allows you to perform asynchronous database operations, which can significantly improve the efficiency of applications that involve database queries. Flask is synchronous by default, and while there are ways to integrate asynchronous code, it may not be as seamless as in FastAPI.

* **WebSocket Support :** FastAPI has built-in support for handling WebSocket communication, enabling real-time

WebSocket Support : FastAPI has built-in support for handling WebSocket communication, enabling real-time bidirectional communication between clients and the server. Flask, while having third-party extensions for WebSocket support, doesn't provide native support out of the box.

* **Security Features :** FastAPI includes features for handling security, such as built-in support for OAuth2, API key authentication, and other security-related functionalities. Flask, while having a robust ecosystem, may require additional third-party packages for certain security features.

* **Data Validation with Pydantic :** FastAPI encourages the use of Pydantic models for request and response data. Pydantic provides powerful data validation and serialization capabilities, ensuring that the data sent and received by your API is well-formed. While Flask has support for data validation, it may not be as integrated and based on Python type hints.

7 . How does FastAPI leverage type hints for automatic API documentation?

FastAPI leverages Python type hints to automatically generate comprehensive and interactive API documentation. Here's how it works:

* **Type Hints in Function Signatures :** When defining routes in FastAPI, you use Python type hints in the function signatures to declare the expected data types for request parameters, request bodies, and response objects.

* **Pydantic Models for Request and Response Bodies :** FastAPI encourages the use of Pydantic models, which are Python classes with type hints provided. These models define the structure and validation rules for request and response bodies.

* **Automatic Validation and Documentation :** FastAPI uses the provided type hints to automatically validate incoming requests and generate OpenAPI and JSON Schema documentation. This documentation includes details about the expected request parameters, request bodies, response objects, and their data types.

* **Swagger UI and ReDoc Integration :** FastAPI provides two interactive web interfaces, Swagger UI and ReDoc, that are automatically generated based on the OpenAPI documentation. These interfaces allow developers to explore and test the API interactively. Swagger UI is accessible at `http://localhost:8000/docs`, and ReDoc at `http://localhost:8000/redoc`.

* **Validation and Documentation for Query Parameters :** Query parameters are also automatically documented based on the provided type hints. The documentation includes information about the parameter name, its type, whether it's required or optional, default values, etc.

8 . Can you explain how you would implement authentication and authorization in FastAPI?

FastAPI provides a security module to implement authentication and authorization. For authentication, OAuth2PasswordBearer is used which requires a URL that the client will use for token retrieval. The get_current_user function uses Depends to inject dependencies, where it decodes the token and fetches user data. If invalid, HTTPException is raised.

For authorization, FastAPI offers Security Scopes. Each route can have a list of scopes as dependencies. When a request comes in, FastAPI checks if the current user has required scopes. If not, an error is returned.

Here's a code snippet :

```
from fastapi import Depends, FastAPI, HTTPException, Security
from fastapi.security import OAuth2PasswordBearer, SecurityScopes
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
def get_current_user(security_scopes: SecurityScopes, token: str = Depends(oauth2_scheme)):
    # decode token and fetch user data here
    raise HTTPException(status_code=403, detail="Not authenticated")
app = FastAPI()
@app.get("/items/", dependencies=[Depends(Security(get_current_user, scopes=["items:read"]))])
async def read_items():
    return [{"item": "Foo", "value": "Bar"}]?
```

9 . How does FastAPI take advantage of Python 3.6 type declarations?

FastAPI utilizes Python 3.6 type declarations to provide several benefits. It uses these annotations for data validation, serialization, and documentation while reducing the amount of code required. FastAPI leverages Pydantic models that use type hints to perform automatic request body JSON parsing, form data handling, and query parameter handling. This results in cleaner, more maintainable code. Additionally, it generates interactive API documentation automatically using OpenAPI standards based on these type declarations.

10 . How would you handle exception handling and custom error responses in FastAPI?

FastAPI provides built-in exception handling. To handle exceptions, use the HTTPException class from fastapi.exceptions module. This class accepts status_code and detail parameters to define HTTP status code and error message respectively.

For custom error responses, create a subclass of HTTPException and override its attributes. You can also customize the validation error response body by creating a route operation function that raises RequestValidationError from fastapi.exceptions and catch it in an exception handler.

Here's a coding example :

```
from fastapi import FastAPI, HTTPException
app = FastAPI()
@app.exception_handler(HTTPException)
async def http_exception_handler(request, exc):
    return JSONResponse(
        status_code=exc.status_code,
        content={"message": f"Oops! {exc.detail}"},
```

```
    )
@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")?
```

11 . What is dependency injection in FastAPI?

Dependency injection in FastAPI is a mechanism that allows you to declare and manage the dependencies required by your route functions or other parts of your application. Dependencies are components or services that your application relies on, such as database connections, authentication services, or external API clients. FastAPI's dependency injection system helps organize and handle these dependencies in a clean and modular way.

Here's how dependency injection works in FastAPI :

Dependency Declaration : You declare dependencies using Python functions, which FastAPI refers to as "dependency functions." These functions can be asynchronous (async def) or synchronous (def), depending on the nature of the dependency.

```
from fastapi import Depends, FastAPI

app = FastAPI()

def get_db():
    # Function to get a database connection
    # ...

def get_current_user(token: str = Depends(get_token)):
    # Function to get the current user based on the token
    # ...

@app.get("/items/")
async def read_items(db: Session = Depends(get_db), current_user: User = Depends(get_current_
    # Route function that depends on the database and current user
    # ...?
```

Dependency Injection in Route Functions : In the example above, the read_items route function has dependencies on a database connection (db) and the current user (current_user). FastAPI automatically injects these dependencies into the function when it is called.

Dependency Resolvers : FastAPI uses dependency resolvers to resolve and inject dependencies into route functions. Dependency resolvers are responsible for obtaining the required dependency instances before executing the route function.

Order of Execution : Dependencies are executed in the order they are declared in the function signature. FastAPI ensures that dependencies with shared sub-dependencies are only executed once, and their results are reused.

Reusable Components : By using dependency injection, you can create reusable components for common functionality. For example, you can define a `get_db` dependency that provides a database session, and then use it in multiple route functions.

Automatic Validation of Dependencies : FastAPI automatically validates and handles the dependencies based on the type hints provided in the dependency functions. If a required dependency cannot be resolved, FastAPI will raise an exception and return an error response.

12 . Explain the role of Pydantic in FastAPI

Pydantic plays a crucial role in FastAPI as it is used for data validation, serialization, and the automatic generation of interactive API documentation.

Here's an explanation of the key roles Pydantic plays in FastAPI :

* **Data Validation :** Pydantic is a data validation library for Python that is heavily used in FastAPI. It allows you to define data models using Python classes with type hints. These models not only serve as documentation for your data structures but also automatically validate incoming request data.

* **Automatic Documentation Generation :** Pydantic models, being a part of the FastAPI application, are leveraged to automatically generate detailed and interactive API documentation. FastAPI uses the type hints and validation rules in the Pydantic models to produce OpenAPI and JSON Schema documentation.

* **Data Serialization :** Pydantic models also play a role in serializing data for responses. When you return an instance of a Pydantic model from a route function, FastAPI automatically serializes it to JSON, ensuring that the response adheres to the structure defined in the model.

* **Validation Errors Handling :** When validation fails (e.g., due to incorrect data types or missing required fields), Pydantic raises validation errors. FastAPI handles these errors and automatically generates appropriate error responses, making it easy to communicate validation issues back to the client.

Pydantic's integration with FastAPI enhances the development experience by providing a clear and consistent way to define, validate, and document data structures within your application. It ensures that your API documentation is always up-to-date and that data validation is applied consistently throughout your FastAPI application.

13 . Can you give me an example of how to use HTTP methods with routes in FastAPI?

You can use `HTTP` methods with routes in FastAPI by specifying the methods argument with a list of methods when you

add a route. For example, if you want to add a route that can be accessed with the `GET` and `POST` methods, you would do the following:

```
@app.get("/my-route", methods=["GET", "POST"])
def my_route():
return "Hello, world!"?
```

14 . How does FastAPI integrate with SQLAlchemy for database connection and ORM?

FastAPI integrates with `SQLAlchemy` through the use of Pydantic models and dependency injection. First, a `SQLAlchemy` model is defined for the database structure. Then, a Pydantic model is created to handle data validation and serialization. FastAPI uses these models in route functions to interact with the database.

Dependency injection allows for reusable dependencies that manage database sessions. A common pattern involves creating a function that yields a session, then closes it after request handling. This function can be included as a parameter in route functions, providing them with a session instance.

For SQL queries, `SQLAlchemy's ORM` is used directly within the route functions. The session provided by the dependency handles transactions, ensuring changes are committed or rolled back appropriately.

15 . How would you handle file uploads in FastAPI?

FastAPI provides a simple way to handle file uploads using the File and UploadFile classes. To upload a file, you would define an endpoint that includes a parameter of type UploadFile. This parameter will be treated as a "form data" parameter.

Here's a basic example :

```
from fastapi import FastAPI, UploadFile, File
app = FastAPI()
@app.post("/files/")
async def create_file(file: UploadFile = File(...)):
    return {"filename": file.filename}?
```

In this code, file: `UploadFile = File(...)` declares a new parameter of type `UploadFile`. The `File` function is a "special function" used to declare it.

The uploaded file is stored in memory up to a limit, and then passed to a temporary file stored on disk. You can access the file with `.file`, get metadata like filename or content type with `.filename` and `.content_type`.

16 . How would you implement rate limiting in FastAPI?

Rate limiting in FastAPI can be implemented using a middleware, such as SlowApi. This is an ASGI middleware for rate limiting based on the standard Python library ratelimit. To use it, you need to install SlowApi and its dependencies, then import and add it to your FastAPI application.

Here's a basic example :

```
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)
@app.get("/home", dependencies=[Depends(limiter.limit("5/minute"))])
async def home(request: Request):
    return {"Hello": "World"}?
```

In this code, we're setting a limit of 5 requests per minute for the "`/home`" endpoint. The key_func parameter determines how to identify different clients, here we're using the client's IP address.

17 . What is the purpose of the BackgroundTasks class in FastAPI?

The BackgroundTasks class in FastAPI is used to schedule background tasks that should be executed after a response has been sent to the client. This allows you to perform non-blocking or time-consuming operations without delaying the response to the client.

The purpose of the BackgroundTasks class can be summarized as follows :

Non-blocking Operations : When handling a request, there are situations where you want to perform additional operations in the background without blocking the response to the client. For example, you might want to log information, send emails, or perform other tasks that do not need to be completed before responding to the client.

Usage in Route Functions : In a FastAPI route function, you can declare a parameter of type BackgroundTasks. `FastAPI` will automatically create an instance of the `BackgroundTasks` class and inject it into your route function.

```
from fastapi import FastAPI, BackgroundTasks

app = FastAPI()

def write_log(message: str):
    # Simulate a time-consuming task (e.g., writing to a log file)
```

```
    with open("log.txt", mode="a") as log:
        log.write(message)

@app.post("/send-notification/{email}")
async def send_notification(email: str, background_tasks: BackgroundTasks):
    message = f"message to {email}"
    background_tasks.add_task(write_log, message)
    return {"message": "Message sent in the background"}?
```

In this example, the `send_notification` route function takes an email parameter and a `background_tasks` parameter of type `BackgroundTasks`. The function sends a message and schedules the `write_log` function to run in the background.

Adding Tasks to the Background : You can add tasks to the background using the `add_task` method of the `BackgroundTasks` instance. Each task added will be executed asynchronously after the response has been sent to the client.

Execution After Response : The tasks scheduled with `BackgroundTasks` are executed in the order they are added, and they are executed after the response to the client has been sent. This ensures that the client receives a quick response, and any additional, potentially time-consuming tasks are performed in the background.

Here's an overview of how it works :

```
from fastapi import FastAPI, BackgroundTasks

app = FastAPI()

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

@app.post("/send-notification/{email}")
async def send_notification(email: str, background_tasks: BackgroundTasks):
    message = f"message to {email}"
    background_tasks.add_task(write_log, message)
    return {"message": "Message sent in the background"}?
```

In this example, when a POST request is made to /send-notification/{email}, the response is sent immediately with

In this example, when a POST request is made to `/send-notification/{email}`, the response is sent immediately with the message "`Message sent in the background`", and the `write_log` function is executed in the background to write to a log file. This ensures a responsive API while still allowing for additional background processing.

18 . Can you explain how you would set up unit tests for a FastAPI application?

To set up unit tests for a `FastAPI` application, you would first install pytest and requests libraries. Then, create a test file in your project directory named '`test_main.py`'. In this file, import the necessary modules including FastAPI's `TestClient`, `pytest`, and your main app.

Next, instantiate the TestClient with your `FastAPI` application as an argument. This client will be used to simulate HTTP requests in your tests.

For each endpoint in your application, write a function that sends a request to it using the `TestClient` and checks the response. Use pytest's assert statements to verify the status code, headers, and body of the response match what is expected.

Remember to isolate each test case by mocking external dependencies and resetting any changes made during the test. Pytest fixtures can help manage setup and teardown tasks.

19 . Could you demonstrate a case where you would prefer to use HTTP protocol directly instead of FastAPI's dependency injection?

FastAPI's dependency injection system is highly efficient for managing dependencies and reducing code repetition. However, there are cases where using HTTP protocol directly might be preferred. One such case could be when dealing with low-level network operations or custom protocols.

For instance, if we need to implement a WebSocket server that communicates via a specific binary protocol, FastAPI's dependency injection may not provide the necessary control over the raw data stream. In this scenario, it would be more appropriate to use an ASGI server like Uvicorn or Hypercorn directly along with Python's built-in asyncio library for handling asynchronous I/O operations.

Here's a simplified example of how you might set up a WebSocket server using Uvicorn:

```
import uvicorn
from starlette.websockets import WebSocket
async def app(scope, receive, send):
    websocket = WebSocket(scope, receive, send)
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
```

```
        # Process data here...
        await websocket.send_text(f"Processed: {data}")
if __name__ == "__main__":
    uvicorn.run(app, host="127.0.0.1", port=8000)?
```

In this example, we have direct access to the incoming data stream and can process it as needed without any abstraction layers introduced by FastAPI's dependency injection system.

20 . How does FastAPI handle serialization and validation of data?

FastAPI uses Pydantic for data serialization and validation. Pydantic models define the shape of incoming or outgoing data, ensuring type correctness. When a request is received, `FastAPI` validates the data against the model's schema using Python's built-in typing system.

If the data doesn't match the schema, `FastAPI` automatically sends an error response detailing the issue. For valid data, it serializes into `JSON` format for `HTTP` responses. This process also works in reverse for incoming `JSON` data, deserializing it into Python objects.

21 . Explain the concept of routers in FastAPI.

In FastAPI, routers are a way to organize and modularize your API by grouping related endpoints together. Routers allow you to create separate sets of routes that are logically grouped based on their functionality. This helps in maintaining a clean and organized codebase, making it easier to manage and scale your FastAPI application.

Here are key concepts related to routers in FastAPI:

Creating a Router : You create a router using the `APIRouter` class from FastAPI. This class allows you to define routes, dependencies, and other aspects of your API within a specific router instance.

```
from fastapi import APIRouter

router = APIRouter()?
```

Defining Routes in a Router : You can define routes within a router using the same decorators (`@app.get, @app.post, etc.`) that you use at the application level. The difference is that you use the decorators on the router instance instead of the main `FastAPI` application instance.

```
@router.get("/items/")
def read_items():
    return {"message": "Read items"}?
```

Mounting a Router : To include a router in your main FastAPI application, you use the `app.include_router` method. This mounts the routes defined in the router to a specific path in the application.

```
from fastapi import FastAPI

app = FastAPI()

app.include_router(router, prefix="/v1")?
```

In this example, all **routes** defined in the `router` will be accessible under the `/v1` path.

Dependency Injection in Routers : Routers support dependency injection in the same way as route functions. You can define dependencies within a router, and FastAPI will handle the injection of dependencies when processing requests to routes in that router.

```
from fastapi import APIRouter, Depends

router = APIRouter()

def get_db():
    # Function to get a database connection
    # ...

@router.get("/items/")
def read_items(db: Session = Depends(get_db)):
    return {"message": "Read items"}?
```

Organizing Code : Routers help in organizing your code by allowing you to group related endpoints together. This is particularly useful as your application grows, and you have multiple sets of functionalities.

Sub-routers : You can create sub-routers by instantiating additional `APIRouter` instances within a router. This allows for hierarchical organization of routes within your application.

```
from fastapi import APIRouter

parent_router = APIRouter()
child_router = APIRouter()

@parent_router.get("/parent/")
```

```python
@parent_router.get("/parent/")
def parent_route():
    return {"message": "Parent route"}

@child_router.get("/child/")
def child_route():
    return {"message": "Child route"}

parent_router.include_router(child_router, prefix="/child")?
```

In this example, the `/child/` route is a sub-route of the `/parent/` route.

Using routers helps in creating modular and maintainable APIs by allowing you to separate concerns and organize your code in a structured manner. It's especially useful when building larger applications with multiple sets of functionality.

22 . What are the advantages of using FastAPI over Django for building APIs?

Both FastAPI and Django are powerful Python web frameworks, but they have different design philosophies and are suited for different use cases. Here are some advantages of using FastAPI over Django for building APIs:

* **Performance :** FastAPI is designed to be high-performance, leveraging asynchronous programming to handle a large number of concurrent requests efficiently. This makes it well-suited for applications with high performance requirements. Django, while powerful, may have more overhead and is traditionally synchronous.

* **Automatic API Documentation :** FastAPI automatically generates interactive API documentation based on the Python type hints used in the code. This documentation is accessible through Swagger UI and ReDoc, providing developers with an easy way to understand and test the API. Django requires additional tools or manual documentation efforts.

* **Type Hints and Validation :** FastAPI uses Python type hints for request and response validation. This not only helps in catching errors early in the development process but also improves code readability. Django typically relies on serializers and forms for data validation.

* **Asynchronous Support :** FastAPI fully supports asynchronous programming, allowing you to write asynchronous routes and take advantage of the performance benefits of asynchronous I/O operations. Django supports asynchronous views as of version 3.1, but its core design is synchronous.

* **Dependency Injection System :** FastAPI has a built-in dependency injection system that makes it easy to manage and inject dependencies in a clean and organized way. Django, while having support for dependency injection, may not have it as seamlessly integrated as FastAPI.

* **Minimal Boilerplate Code :** FastAPI reduces the amount of boilerplate code required to build APIs, especially with automatic data validation and documentation. This can lead to faster development cycles and cleaner, more maintainable code.

* **WebSocket Support :** FastAPI has built-in support for handling WebSocket communication, enabling real-time bidirectional communication between clients and the server. Django channels provides WebSocket support for Django but may involve additional configuration and setup.

* **Single-File Mode :** FastAPI supports a "single-file" mode, allowing you to define routes directly in the main application file without the need for separate views, serializers, and other components. This can be convenient for small projects or quick prototyping.

* **Simplicity and Explicitness :** FastAPI is designed to be simple, explicit, and easy to use. It encourages the use of Python type hints and follows the principle of least surprise. Django, being a full-fledged web framework, may come with more features and abstractions, which could be overkill for API-focused projects.

* **No ORM (Object-Relational Mapping) :** FastAPI does not come with its own ORM. While Django's ORM is powerful and feature-rich, in certain scenarios, using an ORM may be unnecessary or overcomplicated for API-only projects. FastAPI allows developers to choose their preferred database tools.

23 . What is `UTF8JSONResponse`?

`UTF8JSONResponse` is a class used in `FastAPI` to return responses that are encoded in `UTF-8` and formatted as JSON. This is useful for APIs that need to support international characters, as `UTF-8` is a widely used encoding that can represent most languages.

24 . What are OpenAPI specifications and why are they important?

`OpenAPI` specifications are important because they provide a standard, language-agnostic way of describing REST APIs. This allows developers to more easily understand how an API works, and also allows for tools to be built that can automatically generate code or documentation based on the OpenAPI specification.

25 . Do clients need to support JSON schema or OpenAPI specifications to interact with your API?

No, clients do not need to support JSON schema or OpenAPI specifications to interact with your API. However, doing so would certainly make things easier, as it would allow them to automatically generate code to interact with your API.

26 . What are some common error codes returned by FastAPI?

Some common error codes that may be returned by `FastAPI` include :

-400 : Bad Request
-401 : Unauthorized
-403 : Forbidden
-404 : Not Found
-405 : Method Not Allowed
-500 : Internal Server Error

27 . Explain the difference between path parameters and query parameters in FastAPI.

In FastAPI, both path parameters and query parameters are used to extract information from the URL, but they serve different purposes and have distinct syntax. Here's an explanation of the differences between path parameters and query parameters:

Path Parameters :

1. Positional in the URL : Path parameters are part of the URL path itself and are specified by enclosing them in curly braces `{}` within the route path. They are positional and must be included in the order they appear in the route definition.

```
from fastapi import FastAPI

app = FastAPI()

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

In this example, the `item_id` is a path parameter that is part of the URL path `/items/{item_id}`.

2. Required by Default : Path parameters are required by default. If a path parameter is defined in the route, it must be provided in the URL, and its type is automatically validated based on the type hint in the route function.

Query Parameters :

1. Appended to the URL : Query parameters are appended to the URL after a question mark `?` and are usually in the form of `key=value` pairs. They are optional and can be provided in any order.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
def read_item(query_param: str):
    return {"query_param": query_param}?
```

In this example, `query_param` is a query parameter that can be provided in the URL, e.g., `/items/?query_param=value`.

2. Optional and Specified with Defaults : Query parameters are optional by default. You can provide default values for query parameters in the route function to indicate a default value if the parameter is not provided in the URL.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
def read_item(query_param: str = "default_value"):
    return {"query_param": query_param}?
```

If the client does not provide a value for `query_param`, the default value "`default_value`" will be used.

Example :

```
from fastapi import FastAPI

app = FastAPI()

# Path parameter
@app.get("/items/{item_id}")
def read_item_by_id(item_id: int):
    return {"item_id": item_id}

# Query parameter with a default value
@app.get("/items/")
def read_items_with_query_param(query_param: str = "default_value"):
    return {"query_param": query_param}?
```

In the example above, `/items/{item_id}` expects a path parameter (`item_id`) in the URL path, while `/items/` expects an optional query parameter (`query_param`) with a default value.

28 . Can you describe how you would handle CORS (Cross-Origin Resource Sharing) in FastAPI?

FastAPI has a `built-in` middleware for managing CORS, which can be added to the application instance. To enable it, import `CORSMiddleware` from fastapi.middleware.cors and add it to your `FastAPI` app using the `.add_middleware()` method. You need to specify parameters like `allow_origins` (a list of origins that are allowed), `allow_credentials` (whether cookies can be supported), `allow_methods` (HTTP methods allowed), and `allow_headers` (which HTTP headers are permitted). Here's an example:

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
app = FastAPI()
origins = ["http://localhost:3000"]
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
@app.get("/")
async def main():
    return {"message": "Hello World"}?
```

29 . What features does FastAPI provide for form handling?

FastAPI provides robust form handling features. It uses Python type hints to validate incoming data, ensuring that it matches the expected format. This includes checking for required fields and validating field types. FastAPI also supports nested models for complex forms with sub-forms or lists of sub-forms.

Additionally, it allows for custom validation using Pydantic's `@validator` decorator, enabling more complex checks beyond simple type validation. Furthermore, FastAPI can automatically generate interactive documentation for your API including form parameters, making it easier for users to understand how to interact with your endpoints.

30 . How can you serve static files with FastAPI?

FastAPI can serve static files using the `StaticFiles` class from `starlette.staticfiles`. First, import FastAPI and `StaticFiles` from fastapi and `starlette.staticfiles` respectively. Then create an instance of `FastAPI` and mount a new instance of `StaticFiles` to it. The directory parameter in `StaticFiles` should point to your static files' location. For example :

```
from fastapi import FastAPI
from starlette.staticfiles import StaticFiles
app = FastAPI()
app.mount("/static", StaticFiles(directory="static"), name="static")?
```

In this code, "`/static`" is the path where your app will look for static files. "`directory`" is the folder with your static files.

31 . How would you deploy a FastAPI application to a production environment?

FastAPI deployment involves several steps. First, develop the FastAPI application locally and ensure it's functioning as expected. Next, containerize your app using Docker for consistency across environments. Create a Dockerfile in your project directory that includes instructions to build an image of your app.

Here is a basic example :

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
COPY ./app /app?
```

Then, build the Docker image with `docker build -t myimage .` and run it with `docker run -d --name mycontainer -p 80:80 myimage`.

For production, consider deploying on a cloud platform like AWS or Google Cloud. Use their respective services (ECS/Fargate for AWS, Kubernetes Engine for GCP) to manage your containers. Ensure you have set up proper logging and monitoring for your deployed application.

32 . Discuss the role of middleware in FastAPI.

Middleware in FastAPI plays a crucial role in request and response handling, allowing you to perform additional processing before and after route execution. Middleware functions intercept incoming requests and outgoing responses, providing a way to modify or augment them as they pass through the application.

Here's a discussion on the role and functionality of middleware in FastAPI:

1. Request Middleware :

* **Pre-Processing Requests :** Request middleware functions are executed before the route handler functions. They can inspect, modify, or validate incoming requests before they are passed to the corresponding route.

* **Authentication and Authorization :** Middleware can handle authentication and authorization tasks by validating tokens, checking user permissions, or enforcing access controls before allowing access to protected routes.

* **Logging and Metrics :** Middleware functions can log incoming requests, capture request metrics, or perform any other pre-processing tasks such as rate limiting or request filtering.

2. Response Middleware :

* **Post-Processing Responses :** Response middleware functions are executed after the route handler functions. They can modify or augment the response data generated by the route handlers before sending it back to the client.

* **Error Handling** : Middleware can handle errors and exceptions that occur during route execution, allowing you to customize error responses or perform additional error logging or recovery actions.

* **Response Compression or Transformation** : Middleware can compress response payloads, transform response data into different formats (e.g., JSON to XML), or add additional headers or metadata to the response before it's sent back to the client.

3. Implementing Middleware in FastAPI :

* **Using Request and Response Hooks** : FastAPI provides hooks for registering middleware functions that are executed before and after route handlers. You can use the `app.middleware` decorator to register middleware functions for request and response processing.

* **Custom Middleware Classes** : You can implement custom middleware classes by subclassing `starlette.middleware.base.BaseHTTPMiddleware`. This gives you more control over the middleware behavior and allows for more complex middleware logic.

Example :

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

app = FastAPI()

class CustomMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request, call_next):
        # Pre-processing logic
        # e.g., Authentication, Logging, Metrics

        response = await call_next(request)

        # Post-processing logic
        # e.g., Error Handling, Response Transformation

        return response

app.add_middleware(CustomMiddleware)?
```

In this example, `CustomMiddleware` is a custom middleware class that subclasses `BaseHTTPMiddleware`. The `dispatch` method intercepts incoming requests and outgoing responses, allowing you to perform pre-processing and post-processing tasks as needed.

33 . How do FastAPI's response models work and what benefits do they provide?

FastAPI's response models are Python classes that define the structure and data types of HTTP responses. They leverage Pydantic for data validation, serialization, and documentation. When a route function returns a Pydantic model, FastAPI automatically converts it into JSON, checks the data against the model's schema, and generates an OpenAPI schema.

The benefits include :

1. **Data Validation :** Ensures only valid data is returned.

2. **Serialization :** Converts complex data types to JSON.

3. **Documentation :** Auto-generates API docs based on the model.

4. **Code Reusability :** Models can be reused across different routes.

5. **Error Handling :** Automatically handles errors when data doesn't match the model.

34 . How does FastAPI integrate with GraphQL?

FastAPI integrates with GraphQL through third-party libraries that provide GraphQL support. One of the popular libraries for integrating GraphQL with FastAPI is tartiflette. Here's an overview of how you can use tartiflette to add GraphQL support to your FastAPI application:

1. **Install Dependencies :** You need to install the tartiflette library along with the aiohttp adapter to integrate it with FastAPI.

```
pip install tartiflette[aiohttp] uvicorn fastapi?
```

2. **Define GraphQL Schema :** Define your GraphQL schema using the GraphQL Schema Definition Language (SDL) or Python code. You can define your schema in a separate module or directly in your FastAPI application.

```
# schema.graphql
"""
type Query {
    hello: String!
}
"""


# schema.py
from tartiflette import Resolver, create_engine
```

```
@Resolver("Query.hello")
async def resolve_hello(parent, args, context, info):
    return "Hello, World!"?
```

3. **Create FastAPI Application :** Create a FastAPI application and mount the GraphQL endpoint using tartiflette. Use the `TartifletteApp` class provided by `tartiflette`.

```
from fastapi import FastAPI
from tartiflette import Engine
from schema import schema

app = FastAPI()

@app.post("/graphql")
async def graphql(request: Request):
    engine = Engine(schema)
    return await engine.handle_graphql_request(request)?
```

4. **Run the Application :** Run the FastAPI application using `uvicorn`.

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

5. **Access the GraphQL API :** You can now access your GraphQL API by sending POST requests to the `/graphql endpoint`.

```
curl -X POST -H "Content-Type: application/json" -d '{"query": "{ hello }"}' http://localhost
```

Additional Considerations :

* **Authentication and Authorization :** You may need to implement authentication and authorization mechanisms separately in your FastAPI application and integrate them with your GraphQL resolvers as needed.

* **Error Handling :** FastAPI provides robust error handling capabilities, but you may need to handle GraphQL-specific errors within your resolver functions.

* **Performance and Scalability :** Consider performance and scalability implications when designing and implementing your GraphQL API, especially if you expect high traffic or complex query patterns.

* **Schema Stitching and Federation :** For more complex scenarios, you can use schema stitching or federation techniques to compose multiple GraphQL schemas into a single cohesive API.

35 . What is the purpose of the status_code parameter in FastAPI route functions?

In FastAPI route functions, the status_code parameter is used to specify the HTTP status code to be returned in the response. This parameter allows you to customize the status code returned by your API endpoints based on the outcome of the request processing.

Here's how the `status_code parameter` is used in FastAPI route functions :

* **Default Status Code :** By default, FastAPI route functions return a status code of `200 OK` for successful responses.

* **Customizing Status Code :** You can specify a different status code using the `status_code` parameter in the route function decorator.

```
from fastapi import FastAPI, status

app = FastAPI()

@app.get("/items/", status_code=status.HTTP_201_CREATED)
def create_item():
    # Logic to create an item
    return {"message": "Item created successfully"}?
```

In this example, the `/items/` endpoint returns a status code of `201 Created` instead of the default `200 OK`.

* **Error Handling :** You can use the `status_code` parameter to specify the appropriate HTTP status code for error responses, such as `404 Not Found`, `400 Bad Request`, or `500 Internal Server Error`.

```
from fastapi import FastAPI, HTTPException, status

app = FastAPI()

@app.get("/items/{item_id}")
def read_item(item_id: int):
    # Logic to retrieve item from database
    item = ...
```

```
    if not item:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Item not found")

    return item?
```

In this example, if the requested item is not found, the route function raises an `HTTPException` with a status code of `404 Not Found`.

* **Semantic Meaning :** Specifying the appropriate status code enhances the semantic meaning of your API endpoints and communicates the outcome of the request to the client in a standardized way.

Using the `status_code` parameter in FastAPI route functions allows you to customize the HTTP status codes returned by your API endpoints, providing clear and consistent communication of the request outcomes to clients.

36 . How does FastAPI handle cookies and sessions?

FastAPI doesn't directly handle cookies and sessions, but it can be integrated with Starlette's SessionMiddleware for this purpose. To use cookies, FastAPI has a '`cookies`' parameter in path operation functions.

You declare the cookie name as a string argument to receive its value. For sessions, you add `SessionMiddleware` to your application, providing a secret key. This middleware uses signed cookies to store session data client-side. The data is cryptographically signed but not encrypted, so user can see contents but cannot modify them without invalidating signature.

37 . What are some of the security features provided by FastAPI and how can they be utilized?

FastAPI provides several security features. It supports `OAuth2` with Password and Bearer, a standard for user authentication, allowing secure access to resources. This is achieved by using Python-Jose to encrypt and verify tokens. FastAPI also offers `HTTPBasicAuth` for simpler cases where username and password are required.

Another feature is the automatic generation of interactive API documentation with login functionality. This allows users to authenticate directly from their browser while testing endpoints.

FastAPI's dependency system can be used to manage permissions effectively. By creating dependencies for different routes or groups of routes, you can control who has access to what data.

FastAPI also protects against common vulnerabilities like `Cross-Site Scripting (XSS)` and SQL Injection attacks by default. It uses Pydantic models which automatically validate incoming JSON requests, preventing malicious code from being executed.

38 . How can FastAPI be used with async and await, and what benefits does this provide?

FastAPI supports asynchronous request handling through Python's async and await keywords. This allows for concurrent processing of requests, improving application performance. When a FastAPI route is defined with an async function, it becomes a coroutine that can be paused and resumed, allowing other tasks to run in the meantime.

For instance :

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def read_root():
    return {"Hello": "World"}?
```

In this example, read_root is an asynchronous function. If it calls another async function with await, execution returns to the event loop, freeing up resources until the awaited function completes.

This non-blocking nature of **async I/O operations** means your app can handle more requests with fewer resources, as idle time waiting for **I/O (like network or disk access)** can be used to serve other requests. It also simplifies code by avoiding callback hell or threading complexities, making it easier to write and maintain.

39 . Discuss the role of the security parameter in FastAPI route functions.

In FastAPI, the security parameter in route functions is used to define security requirements for accessing the endpoint. It allows you to specify authentication and authorization mechanisms required to access the route. The security parameter is a list of security requirements, where each requirement can be a dependency or a security scheme.

Here's a detailed discussion on the role of the security parameter in FastAPI route functions:

1. Authentication and Authorization :

* **Authentication Schemes :** You can specify one or more authentication schemes required to authenticate the client making the request. Common authentication schemes include OAuth2, JWT (JSON Web Tokens), HTTP Basic Authentication, etc.

* **Authorization Dependencies :** The security requirements can include dependencies that handle user authentication and authorization. These dependencies can validate tokens, check user roles or permissions, and enforce access controls.

2. Defining Security Requirements :

* **Security Schemes :** FastAPI supports various security schemes, such as OAuth2PasswordBearer, OAuth2PasswordRequestForm, APIKey, HTTPBearer, etc. These schemes can be used as security requirements in the security parameter.

* **Custom Dependencies :** You can define custom dependencies that handle authentication and authorization logic according to your application's requirements. These dependencies can be reused across multiple routes to enforce

consistent security policies.

3. Role-based Access Control :

* **Role-based Requirements :** You can specify security requirements based on user roles or permissions. For example, you may require certain routes to be accessible only to users with specific roles, such as administrators or moderators.

40 . How does FastAPI support automatic API documentation generation?

FastAPI supports automatic API documentation generation using the OpenAPI standard. OpenAPI is a specification for building and documenting APIs, and FastAPI leverages this specification to automatically generate interactive API documentation. Here's how FastAPI achieves automatic API documentation generation:

1. **Pydantic Models :** FastAPI uses Pydantic models to define request and response data structures. These models include type hints and validation rules, which serve as the basis for generating documentation.

```
from pydantic import BaseModel

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

2. **Route Decorators :** FastAPI uses route decorators (@app.get, @app.post, etc.) to define API endpoints. These decorators include metadata such as route paths, parameter types, and response models, which are used for documentation generation.

```
from fastapi import FastAPI

app = FastAPI()

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

3. **Automatic Documentation Generation :** FastAPI automatically generates interactive API documentation based on the defined routes, request and response models, and other metadata. The generated documentation includes detailed

information about each endpoint, including route paths, HTTP methods, request parameters, response models, and example usage.

4. Swagger UI and ReDoc : FastAPI provides built-in support for Swagger UI and ReDoc, which are popular tools for viewing and interacting with OpenAPI documentation. When you run your FastAPI application and navigate to `/docs` or `/redoc` endpoints in your browser, you will see the automatically generated API documentation.

5. OpenAPI Schema : FastAPI generates an OpenAPI schema (formerly known as Swagger schema) based on the defined routes and models. This schema is a machine-readable JSON document that describes the API endpoints, request and response models, and other details required for documentation.

Example :

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

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

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

When you run this FastAPI application and navigate to `http://localhost:8000/docs` in your browser, you will see the automatically generated API documentation with interactive features for testing endpoints.