Top 50 FastAPI Job Interview Questions and Answer

FastAPI, a modern, high-performance web framework for Python, has rapidly gained popularity among developers due to its simplicity, efficiency, and developer-friendly features. Its ability to create robust and scalable APIs has made it a go-to choice for a wide range of applications. As the demand for skilled FastAPI developers continues to grow, understanding the framework's core concepts and potential interview questions becomes increasingly important.

This blog post aims to provide a comprehensive guide to the Top 50 FastAPI Interview Questions and Answers. By exploring key topics such as asynchronous programming, dependency injection, path operations, data validation, and advanced features, we'll equip you with the knowledge and confidence to excel in your FastAPI interviews. Whether you're a seasoned Python developer or new to the framework, this resource will serve as a valuable asset in your career journey.

FastAPI Interview Questions and Answers

1. Explain the concept of asynchronous programming in Python.

Asynchronous programming allows multiple tasks to run concurrently without blocking each other. In Python, it's implemented using coroutines and event loops.

2. How does FastAPI leverage asynchronous programming for performance?

FastAPI uses the `uvicorn` ASGI server, which is inherently asynchronous. This allows it to handle multiple requests concurrently without blocking the main thread, improving performance and scalability.

3. What are the benefits of using asynchronous programming in FastAPI?

- **Improved performance:** Handles more concurrent requests without blocking.
- **Non-blocking I/O:** Efficiently handles operations like database queries and network requests.
- **Better scalability:** Can handle higher loads without additional resources.

4. What is dependency injection in FastAPI?

Dependency injection is a design pattern where dependencies (objects needed by a class) are provided to the class from the outside, rather than the class creating them.

5. How does dependency injection simplify code organization and testing in FastAPI?

- **Loose coupling:** Classes become less dependent on specific implementations, making them easier to test and reuse.
- **Dependency management:** FastAPI's built-in dependency injection system handles the creation and management of dependencies.

6. Provide an example of using dependency injection in FastAPI.

Python

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

class MyDependency:
def init(self, message: str):
self.message = message

class Item(BaseModel):
name: str

app = FastAPI()

@app.get("/")
async def read_items(item: Item, my_dependency: MyDependency = Depends(MyDependency)):
return {"item": item, "message": my_dependency.message}
```

7. What are path operations in FastAPI?

Path operations define the endpoints of your API, specifying the HTTP method (GET, POST, PUT, DELETE, etc.) and the path that triggers the operation.

8. Discuss different types of path operations (GET, POST, PUT, DELETE, etc.)

- **GET:** Retrieves data from the server.
- **POST:** Sends data to the server to create a new resource.
- **PUT:** Updates an existing resource.
- **DELETE:** Removes a resource from the server.
- **PATCH:** Partially updates an existing resource.

9. Demonstrate how to create a path operation using FastAPI.

from fastapi import FastAPI

```
app = FastAPI()

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

10. Explain the importance of data validation in API development.

- **Data integrity:** Ensures that incoming data is valid and consistent.
- **Security:** Helps prevent malicious attacks like injection and cross-site scripting.
- **User experience:** Provides clear feedback to users when invalid data is submitted.

11. How does FastAPI use Pydantic for data validation?

Pydantic is a library that defines data structures and validates data against those structures. FastAPI uses Pydantic to validate request and response data.

12. Provide examples of validating request and response data in FastAPI.

Python

from fastapi import FastAPI, Request, HTTPException
from pydantic import BaseModel

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

app = FastAPI()

@app.post(“/items/”)
async def create_item(item: Item, request: Request):
# Validate request data using Pydantic
item.validate()

```
# ... process item data

return {"item_id": 123}
```

13. Explain the concept of WebSockets and their use cases.

WebSockets provide full-duplex communication between a client and server, allowing for real-time updates and bidirectional data transfer. They are commonly used for chat applications, online games, and real-time data visualizations.

14. Demonstrate how to implement WebSockets in FastAPI.

“python from fastapi import FastAPI, WebSocket, WebSocketDisconnect

app = FastAPI()

@app.websocket(“/ws/{client_id}”)

async def websocket_endpoint(websocket: WebSocket, client_id: int):

```
await websocket.accept()

try:

while True:

data = await websocket.receive_text()

await websocket.send_text(f"You sent: {data}")

except WebSocketDisconnect:

pass
```

15. Discuss challenges and best practices for WebSocket development.

- **Error handling:** Implement proper error handling mechanisms to prevent unexpected behavior.
- **Scalability:** Consider using libraries like `websockets` or `asgi-redis` for handling large numbers of concurrent WebSocket connections.
- **Security:** Protect against vulnerabilities like XSS and CSRF.

16. Explain background tasks and their use cases.

Background tasks are tasks that run asynchronously in the background, allowing your application to continue processing requests while performing long-running operations. They are useful for tasks like sending emails, processing large datasets, or triggering periodic updates.

17. Demonstrate how to create background tasks using FastAPI.

```
from fastapi import FastAPI, BackgroundTask
import time

app = FastAPI()

async def send_email(recipient: str):
# Simulate sending an email
time.sleep(2)
print(f"Sending email to {recipient}")

@app.get("/send-email/{recipient}")
async def send_email_endpoint(recipient: str):
background_task = BackgroundTask(send_email, recipient=recipient)
return {"message": "Email sent in the background"}
```

18. Discuss best practices for managing background tasks.

- **Error handling:** Implement proper error handling mechanisms to ensure background tasks are retried if they fail.
- **Task prioritization:** If multiple background tasks are running, consider implementing a priority system to ensure important tasks are executed first.
- **Task monitoring:** Use tools or libraries to monitor the status of background tasks and track their progress.

19. Discuss security considerations in FastAPI development.

- **Authentication:** Implement mechanisms to verify the identity of users (e.g., using tokens, OAuth).
- **Authorization:** Control access to resources based on user roles and permissions.
- **Input validation:** Validate user input to prevent attacks like injection and cross-site scripting.
- **Data encryption:** Encrypt sensitive data both at rest and in transit.
- **Security headers:** Use security headers like `Content-Security-Policy` and `HTTP Strict Transport Security` to protect against common`python from fastapi import FastAPI, Depends, HTTPException from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm from pydantic import BaseModel

app = FastAPI()

security = OAuth2PasswordBearer(tokenUrl=”token”)

class User(BaseModel): username: str password: str

async def get_current_user(security_token: str = Depends(security)): user = verify_token(security_token) # Replace with your token verification logic return user

@app.post(“/token”) async def login(form_data: OAuth2PasswordRequestForm = Depends()): user = authenticate_user(form_data.username, form_data.password) # Replace with your authentication logic if not user: raise HTTPException(status_code=400, detail=”Incorrect username or password”) access_token = create_access_token(user) return {“access_token”: access_token, “token_type”: “bearer”}

@app.get(“/items/”) async def read_items(current_user: User = Depends(get_current_user)): # Only authenticated users can access this endpoint return {“items”: [{“item_id”: 1, “name”: “Foo”}, {“item_id”: 2, “name”: “Bar”}]}

20. Provide examples of using security features in FastAPI.

- **Password hashing:** Use strong password hashing algorithms like bcrypt to store user passwords securely.
- **CORS:** Configure CORS settings to allow requests from specific domains.
- **Rate limiting:** Limit the number of requests a client can make within a certain time period to prevent abuse.
- **Data encryption:** Encrypt sensitive data using libraries like `cryptography`.
- **Security headers:** Set security headers like `Content-Security-Policy` and `HTTP Strict Transport Security` to protect against common web attacks.

21. Discuss the importance of testing in FastAPI development.

- **Quality assurance:** Ensures that your API code is reliable and works as expected.
- **Error prevention:** Helps identify and fix bugs before they are deployed to production.
- **Regression testing:** Verifies that changes to your code don’t introduce new bugs.
- **Documentation:** Test cases can serve as documentation for your API’s behavior.

22. Demonstrate how to write unit and integration tests for FastAPI applications.

import pytest
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

```
app = FastAPI()

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

@app.post("/items/")
async def create_item(item: Item):
return {"item_id": 123}

# Unit test

def test_create_item():
client = TestClient(app)
response = client.post("/items/", json={"name": "Foo", "price": 10.0})
assert response.status_code == 200
assert response.json() == {"item_id": 123}

# Integration test

def test_create_item_with_invalid_data():
client = TestClient(app)
response = client.post("/items/", json={"name": ""})
assert response.status_code == 422
assert response.json()["detail"][0]["msg"] == "required"
```

23. Explain best practices for testing FastAPI applications.

- **Test coverage:** Aim for high test coverage to ensure that all parts of your code are thoroughly tested.
- **Test isolation:** Write tests that are independent of each other to avoid unexpected interactions.
- **Test automation:** Use tools like pytest to automate the running of your tests.
- **Continuous integration:** Integrate your tests into your continuous integration pipeline to automatically run them with every code change.

24. Explain the concept of dependency injection in FastAPI.

Dependency injection is a design pattern where dependencies (objects needed by a class) are provided to the class from the outside, rather than the class creating them.

25. How do you handle request validation errors in FastAPI?

FastAPI automatically handles request validation errors and returns a detailed error message with a 422 status code when input data doesn't conform to the defined Pydantic model.

26. How can you secure FastAPI endpoints with OAuth2?

FastAPI provides built-in support for OAuth2 via `OAuth2PasswordBearer` and `OAuth2PasswordRequestForm` to secure routes. It requires creating an authentication flow that generates and validates JWT tokens.

27. How can you handle background tasks in FastAPI?

Background tasks can be handled using the `BackgroundTasks` class:

from fastapi import BackgroundTasks
def send_email(email: str):
# Email sending logic
@app.post("/send-email/")
def send_email_background(email: str, background_tasks: BackgroundTasks):
background_tasks.add_task(send_email, email)
return {"message": "Email will be sent"}

28. What are middleware in FastAPI, and how do you use them?

Middleware is a layer that processes requests before they reach your route and can handle responses before sending them to the client. Example of using middleware:

```
pythonCopy codefrom starlette.middleware.base import BaseHTTPMiddleware
app.add_middleware(BaseHTTPMiddleware, dispatch=my_custom_middleware)
```

29. How do you test FastAPI applications?

FastAPI supports testing with the `TestClient` from `starlette.testclient`:

```
pythonCopy codefrom fastapi.testclient import TestClient
client = TestClient(app)
def test_read_item():
    response = client.get("/items/")
    assert response.status_code == 200
```

30. How do you structure a FastAPI project?

A typical FastAPI project structure includes separate modules for routes, models, and services. For example:

```
markdownCopy code- app/
  - main.py
  - routes/
  - models/
  - services/
```

31. What are CORS, and how do you handle them in FastAPI?

CORS (Cross-Origin Resource Sharing) restricts which domains can interact with your API. You can enable CORS in FastAPI using the `CORSMiddleware`:

```
pythonCopy codefrom fastapi.middleware.cors import CORSMiddleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

32. How do you upload files in FastAPI?

You can upload files using the `File` and `UploadFile` classes:

```
pythonCopy codefrom fastapi import File, UploadFile
@app.post("/uploadfile/")
def upload_file(file: UploadFile = File(...)):
    return {"filename": file.filename}
```

33. How do you implement pagination in FastAPI?

Pagination is typically implemented by accepting `limit` and `skip` query parameters and returning a subset of the data:

```
pythonCopy code@app.get("/items/")
def get_items(skip: int = 0, limit: int = 10):
    return items[skip : skip + limit]
```

34. How do you use WebSockets in FastAPI?

WebSockets in FastAPI are supported via `@app.websocket()` decorator:

```
pythonCopy code@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    await websocket.send_text("Hello WebSocket")
```

35. What is the difference between `@app.get()` and `@app.post()` in FastAPI?

`@app.get()` handles HTTP GET requests, typically used to retrieve data, while `@app.post()` handles HTTP POST requests, used to submit data to the server.

36. How can you customize exception handling in FastAPI?

FastAPI allows custom exception handling using `@app.exception_handler()` decorator:

```
pythonCopy codefrom fastapi.exceptions import RequestValidationError
@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    return JSONResponse(status_code=422, content={"detail
```

37. How do you handle form data in FastAPI?

Form data can be handled using `Form` from FastAPI: `python from fastapi import Form @app.post("/login/") def login(username: str = Form(...), password: str = Form(...)): return {"username": username}`

38. How do you handle multiple query parameters in FastAPI?

You can handle multiple query parameters by defining them as function arguments. FastAPI will automatically parse and validate them: `python @app.get("/items/") def read_items(q: str = None, limit: int = 10): return {"q": q, "limit": limit}`

39. What are request and response objects in FastAPI?

The request object represents the incoming HTTP request, and the response object represents the outgoing HTTP response. You can use `Request` and `Response` classes from FastAPI: `python from fastapi import Request, Response @app.get("/custom-response/") def custom_response(request: Request, response: Response): response.headers["X-Custom-Header"] = "Custom value" return {"message": "Custom response"}`

40. How can you validate a specific field in a Pydantic model?

You can use Pydantic validators to validate fields: `python from pydantic import BaseModel, validator class Item(BaseModel): name: str price: float @validator('price') def`

```
price_must_be_positive(cls, value): if value <= 0: raise ValueError("Price must be positive") return value
```

41. What is the purpose of `Depends()` in FastAPI?

`Depends()` is used to declare dependencies for your path operations. It allows you to reuse logic such as database connections, authentication, etc.

42. How do you handle global dependencies in FastAPI?

Global dependencies can be applied across multiple routes by adding them to the `dependencies` argument when creating the FastAPI instance: `python app = FastAPI(dependencies=[Depends(common_dependency)])`

43. What are asynchronous dependencies, and how can you define them in FastAPI?

Asynchronous dependencies are defined with `async def` and can be injected using `Depends()` like regular dependencies. These are useful for async I/O operations such as database queries.

44. How can you handle rate limiting in FastAPI?

Rate limiting can be implemented using third-party libraries like `slowapi` or custom middleware that limits requests per user or IP: `python from slowapi import Limiter from slowapi.util import get_remote_address limiter = Limiter(key_func=get_remote_address) @app.get("/items/") @limiter.limit("5/minute") def get_items(): return {"message": "This is rate-limited"}`

45. How do you manage database connections in FastAPI?

Database connections can be managed using dependency injection. You can define a dependency that yields a database session: `python def get_db(): db = SessionLocal() try: yield db finally: db.close() @app.get("/items/") def read_items(db: Session = Depends(get_db)): return db.query(Item).all()`

46. How can you implement JWT-based authentication in FastAPI?

JWT-based authentication is implemented by using `OAuth2PasswordBearer` and creating JWT tokens upon successful login: `python from fastapi.security import OAuth2PasswordBearer oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token") @app.get("/users/me") def read_users_me(token: str = Depends(oauth2_scheme)): user = verify_token(token) return user`

47. What is the difference between synchronous and asynchronous endpoints in FastAPI?

Synchronous endpoints are blocking and may cause performance issues with high I/O operations, while asynchronous endpoints allow for non-blocking I/O operations, improving performance in these cases.

48. How do you handle exceptions globally in FastAPI?

Global exception handling can be achieved using custom exception classes and `@app.exception_handler()`: `python from fastapi import HTTPException class CustomException(Exception): def __init__(self, name: str): self.name = name @app.exception_handler(CustomException) def custom_exception_handler(request, exc: CustomException): return JSONResponse(status_code=418, content={"message": f"Custom Exception: {exc.name}"})`

49. How can you optimize the performance of FastAPI?

Performance optimizations can include using asynchronous endpoints for I/O-bound operations, caching responses, using efficient middleware, and employing load balancers like Nginx or Traefik.

50. How do you handle large file uploads in FastAPI?

Large file uploads can be handled by adjusting the ASGI server configuration, using `StreamingResponse`, or writing the file in chunks: `python from fastapi.responses import StreamingResponse @app.post("/upload/") async def upload_large_file(file: UploadFile = File(...)): async def iterfile(): yield from file.file return StreamingResponse(iterfile(), media_type="application/octet-stream")`

Conclusion

In this guide, we have explored the Top 50 FastAPI Interview Questions and Answers. By understanding the core concepts, advanced topics, and best practices, you are well-equipped to tackle any FastAPI-related interview question.

Key Points:

- **Master the fundamentals:** Understand asynchronous programming, dependency injection, path operations, and data validation.
- **Explore advanced topics:** Familiarize yourself with WebSockets, background tasks, security, and testing.
- **Practice consistently:** Regularly practice solving FastAPI-related problems to enhance your skills.
- **Stay updated:** Keep up with the latest developments and best practices in the FastAPI ecosystem.