

LEARNINGS FROM MIGRATING A FLASK APP TO FASTAPI

PyCon DE & PyData

Orell Garten

24.04.2025



[Presentation on GitHub](#)



LET'S START WITH SOME QUESTIONS

LET'S START WITH SOME QUESTIONS

Who has used flask?

LET'S START WITH SOME QUESTIONS

Who has used flask?

Who has used FastAPI?

LET'S START WITH SOME QUESTIONS

Who has used flask?

Who has used FastAPI?

Has anyone migrated an app?

/WHOAMI

/WHY

Are there any good reasons to migrate from flask to fastAPI?

/WHY

Are there any good reasons to migrate from flask to fastAPI?

Performance?

/WHY

Are there any good reasons to migrate from flask to fastAPI?

Performance?

Data Validation?

/WHY

Are there any good reasons to migrate from flask to fastAPI?

Performance?

Data Validation?

Async?

/WHY

Are there any good reasons to migrate from flask to fastAPI?

Performance?

Data Validation?

Async?

Developer Experience?

GOALS

Migration from Flask to FastAPI was not the main goal.

- We wanted:
 - Database integration
 - better code quality
 - Better testability

Migration to FastAPI was helping us achieve this.

DISCLAIMER

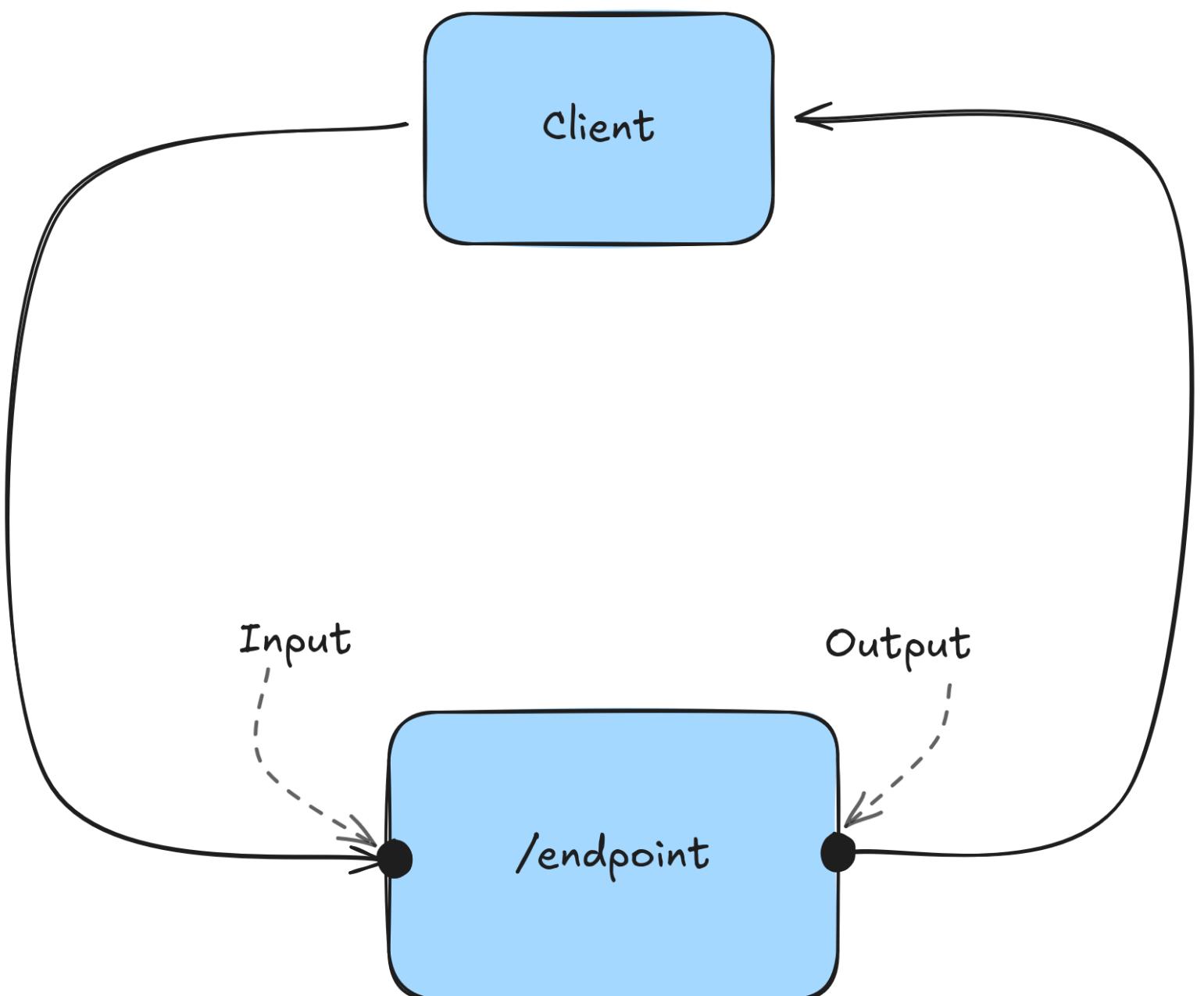
- This talk is about **my** experiences.
- Your experience might be different

/AGENDA

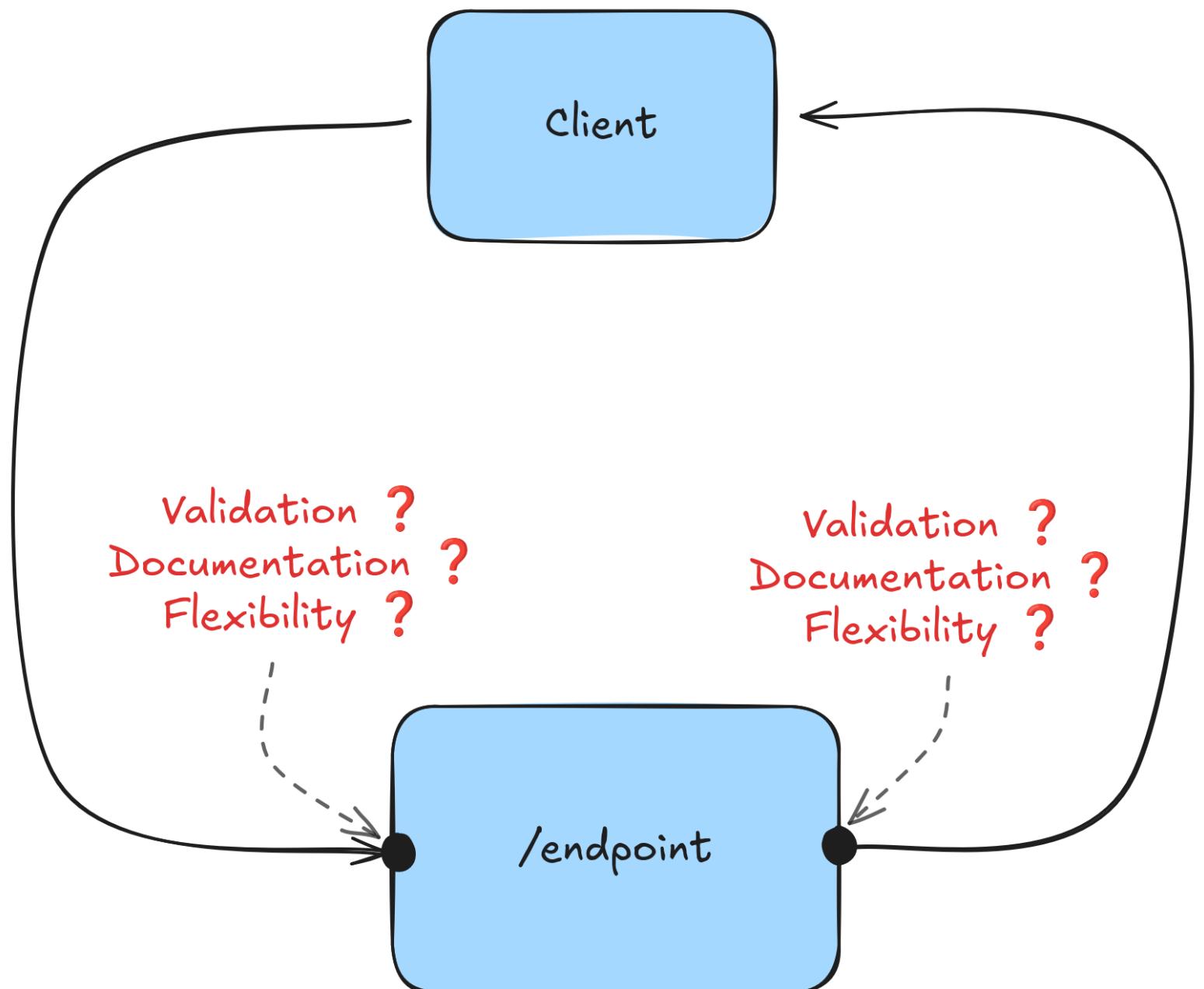
1. Data modeling
2. async is overrated
3. Problems you will encounter
4. Migration strategy
5. Q&A

DATA MODELING

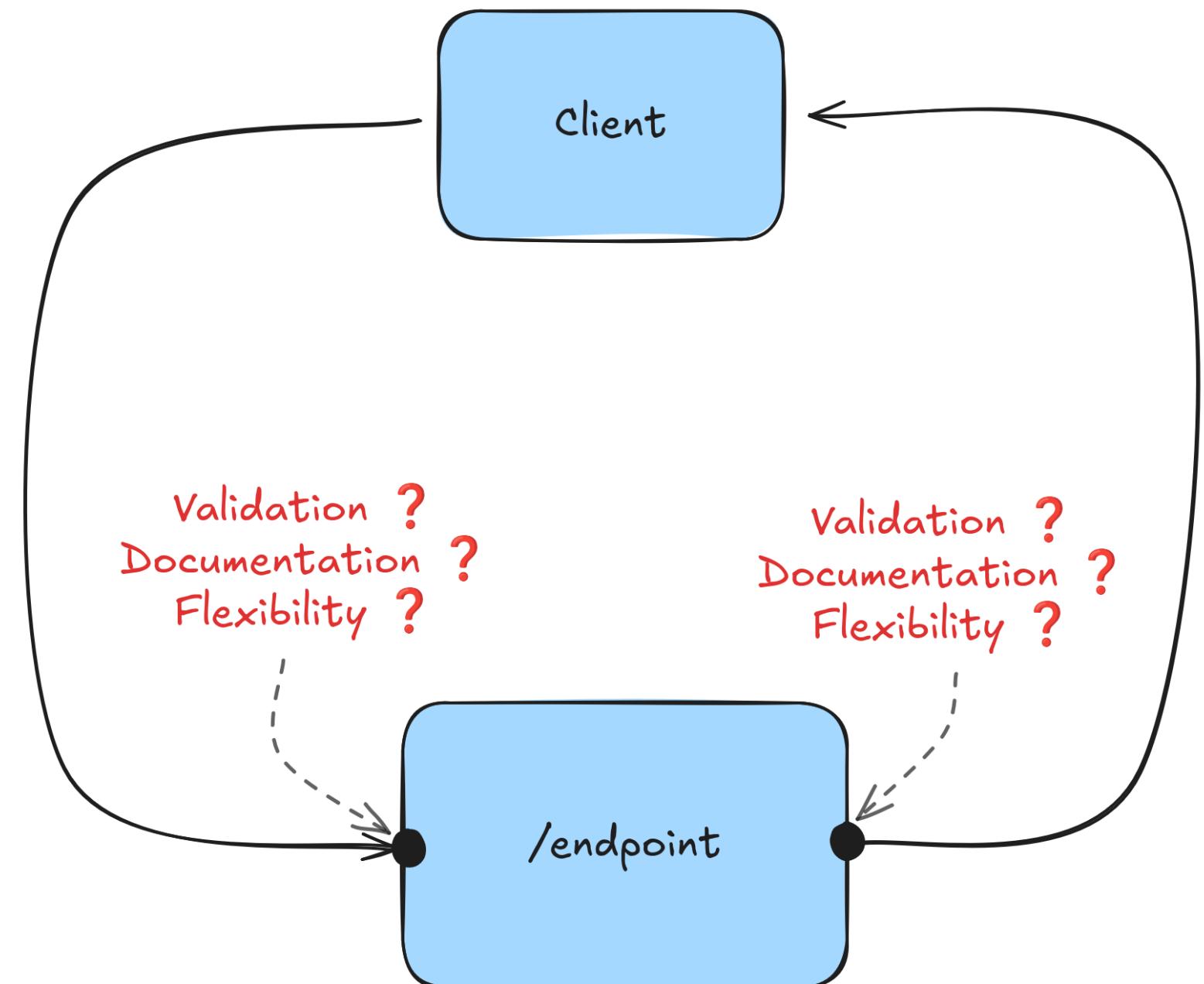
THE SITUATION



THE SITUATION



THE SITUATION



How do we deal with input and outputs in a good way?

INPUTS WITH FLASK

```
1 @app.route('/user', methods=['POST'])
2 def create_user():
3     data = request.get_json()
4     username = data.get('name')
5     email = data.get('email')
6     password = data.get('password')
7
8     # Create user in DB
9     db[1] = {"name": username, "email": email, "password": password}
10
11    return jsonify({'message': 'User created successfully', 'id': 1}), 201
```

INPUTS WITH FLASK

```
1 @app.route('/user', methods=['POST'])
2 def create_user():
3     data = request.get_json()
4     username = data.get('name')
5     email = data.get('email')
6     password = data.get('password')
7
8     # Create user in DB
9     db[1] = {"name": username, "email": email, "password": password}
10
11    return jsonify({'message': 'User created successfully', 'id': 1}), 201
```

INPUTS WITH FLASK

```
1 @app.route('/user', methods=['POST'])
2 def create_user():
3     data = request.get_json()
4     username = data.get('name')
5     email = data.get('email')
6     password = data.get('password')
7
8     # Create user in DB
9     db[1] = {"name": username, "email": email, "password": password}
10
11    return jsonify({'message': 'User created successfully', 'id': 1}), 201
```

INPUTS WITH FLASK

```
1 @app.route('/user', methods=['POST'])
2 def create_user():
3     data = request.get_json()
4     username = data.get('name')
5     email = data.get('email')
6     password = data.get('password')
7
8     # Create user in DB
9     db[1] = {"name": username, "email": email, "password": password}
10
11    return jsonify({'message': 'User created successfully', 'id': 1}), 201

$ curl -X 'POST' \
  'http://localhost:8000/user' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{"name": "Orell", "email": "hello@orellgarten.com", "password": "test"}'

{"id":1,"message":"User created successfully"}
```

OUTPUTS WITH FLASK

```
1 @app.route("/user/<int:id>", methods=['GET'])
2 def get_user(id: int):
3     # do something, e.g. create user in database
4     user = db.get(id)
5
6     return user
```

OUTPUTS WITH FLASK

```
1 @app.route("/user/<int:id>", methods=['GET'])
2 def get_user(id: int):
3     # do something, e.g. create user in database
4     user = db.get(id)
5
6     return user
```

OUTPUTS WITH FLASK

```
1 @app.route("/user/<int:id>", methods=['GET'])
2 def get_user(id: int):
3     # do something, e.g. create user in database
4     user = db.get(id)
5
6     return user
```

OUTPUTS WITH FLASK

```
1 @app.route("/user/<int:id>", methods=['GET'])
2 def get_user(id: int):
3     # do something, e.g. create user in database
4     user = db.get(id)
5
6     return user
```

```
$ curl -X 'GET' 'http://localhost:8000/user/1'
{"email": "hello@orellgarten.com", "name": "Orell", "password": "test"}
```

OUTPUTS WITH FLASK

```
1 @app.route("/user/<int:id>", methods=['GET'])
2 def get_user(id: int):
3     # do something, e.g. create user in database
4     user = db.get(id)
5
6     return user
```

```
$ curl -X 'GET' 'http://localhost:8000/user/1'
{"email": "hello@orellgarten.com", "name": "Orell", "password": "test"}
```

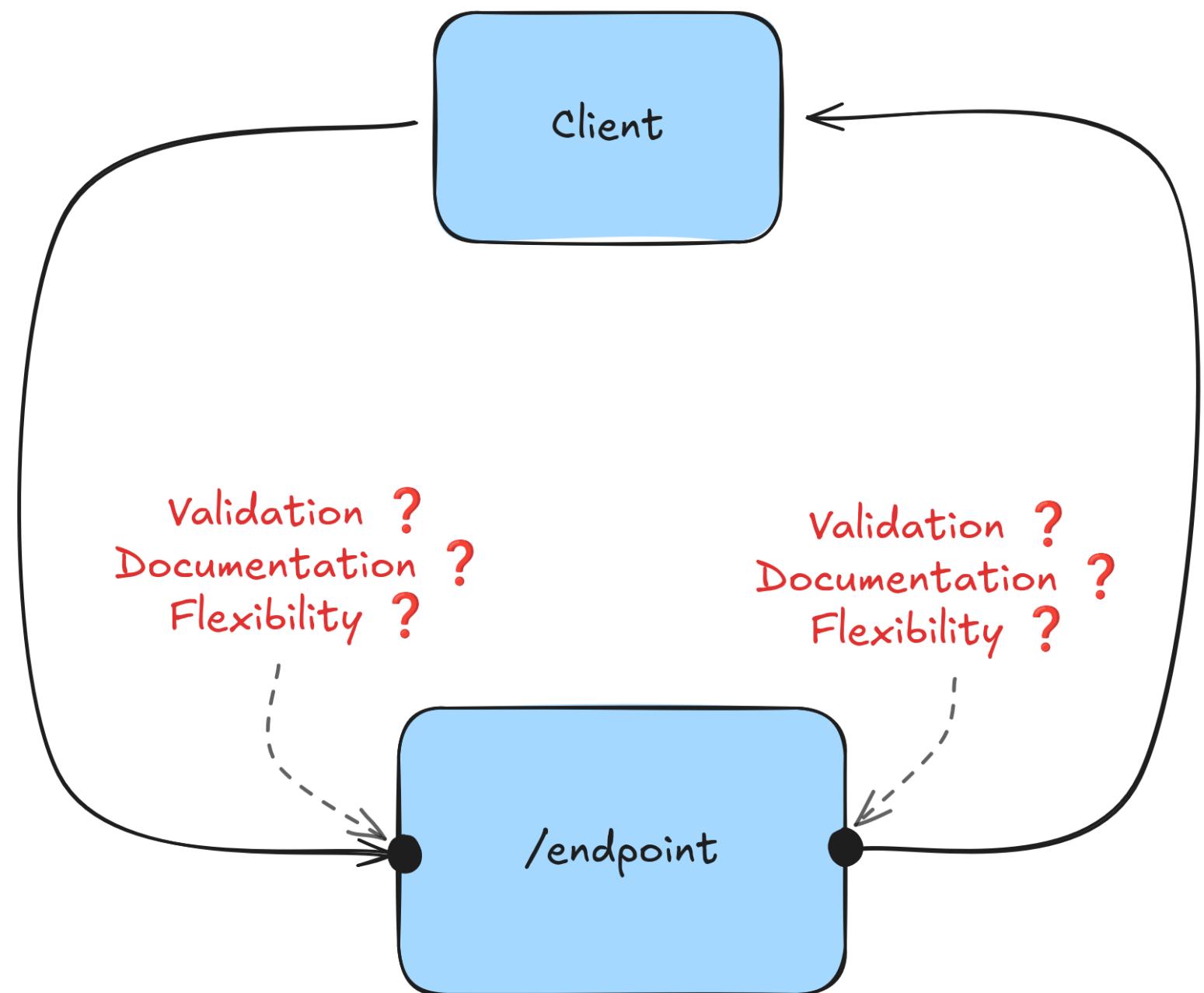
Oops, password in response!

FLASK IN/OUT

- Work with `request` object
- Typing not required

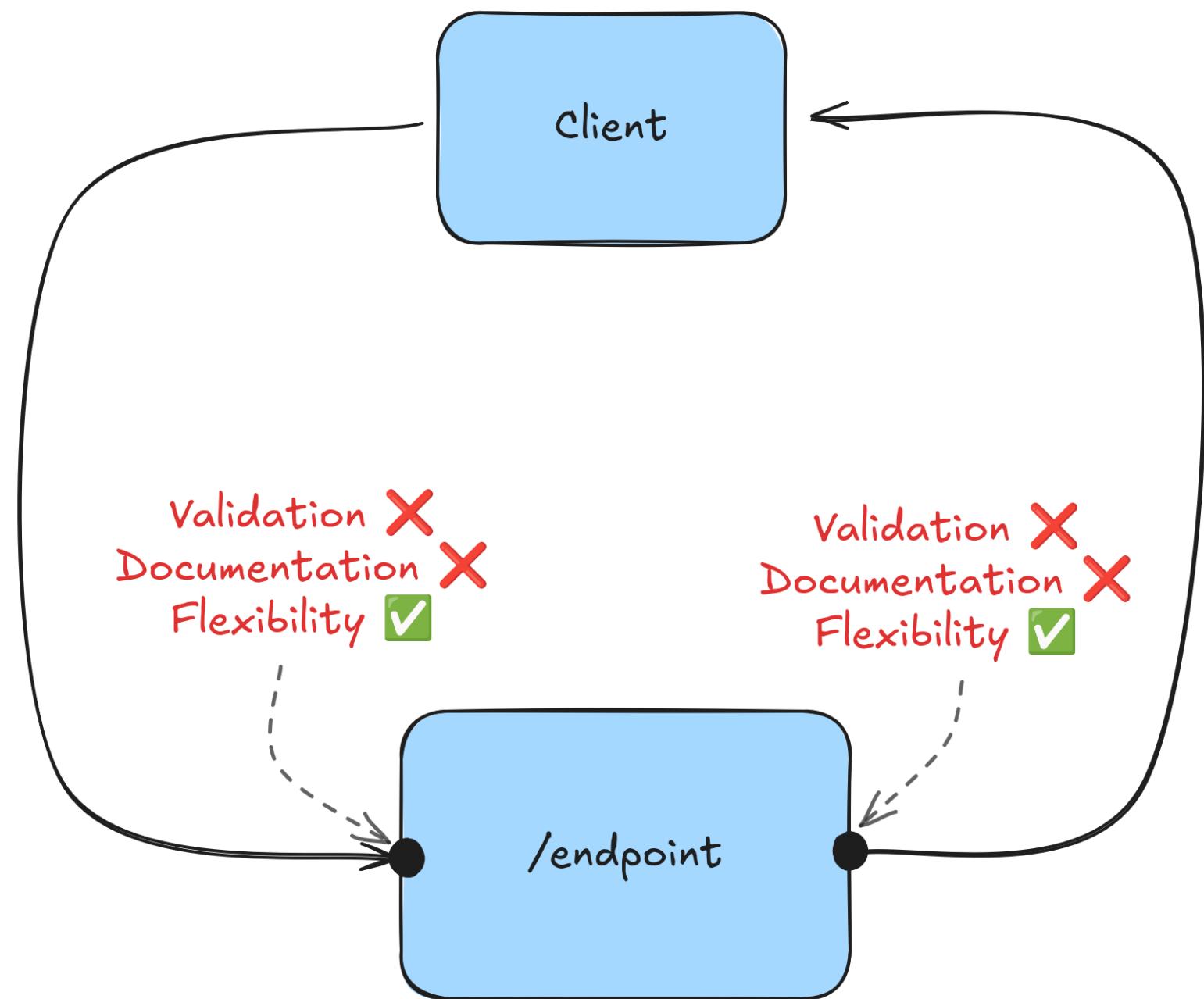
FLASK IN/OUT

- Work with `request` object
- Typing not required



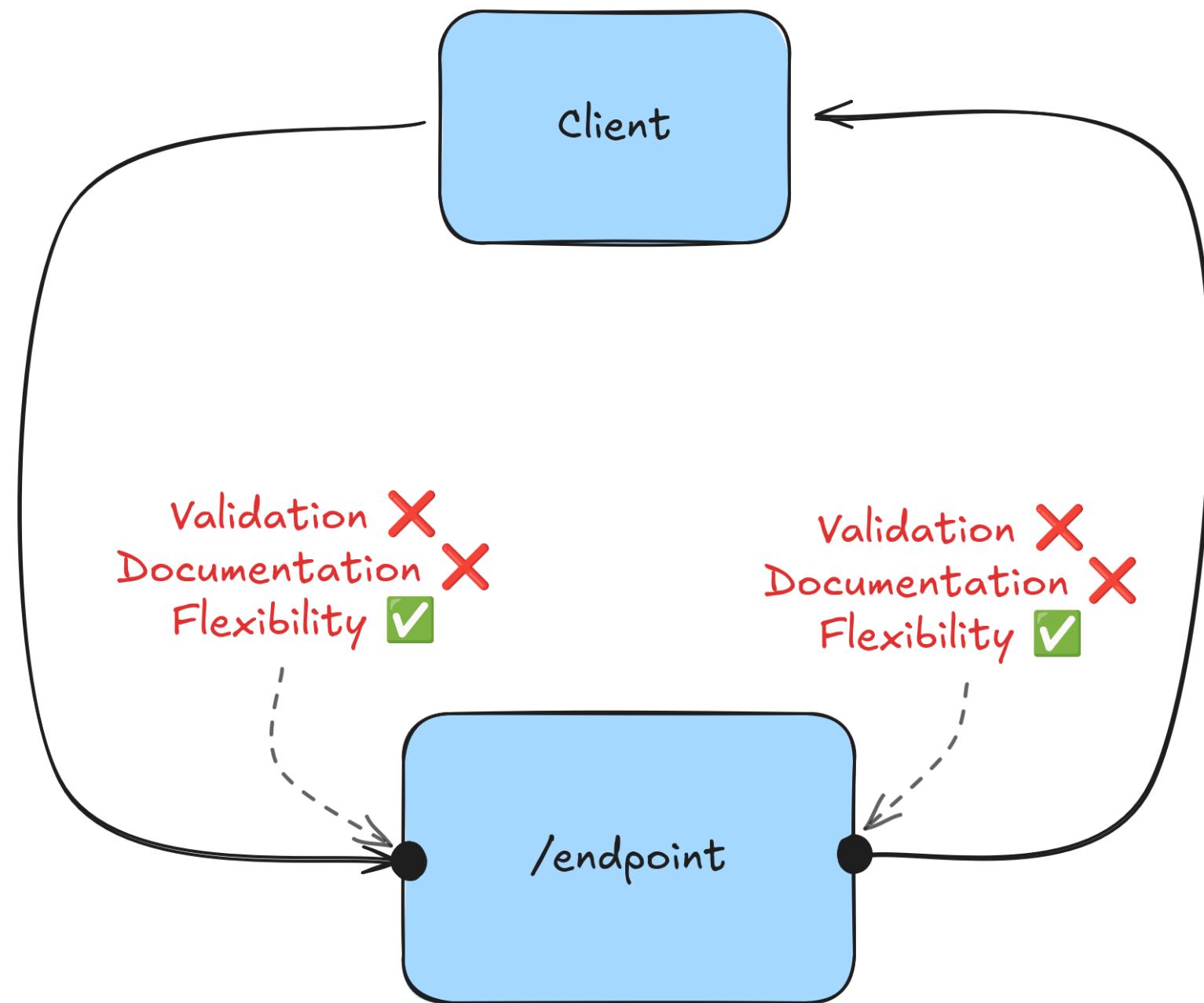
FLASK IN/OUT

- Work with `request` object
- Typing not required



FLASK IN/OUT

- Work with `request` object
- Typing not required



- No automatic input validation
- Manual data validation requires more boilerplate/libraries

INPUTS WITH FASTAPI

```
27 class UserIn(BaseModel):
28     name: str
29     email: str
30     password: str
31
32 @app.post("/user", status_code=status.HTTP_201_CREATED)
33 def create_user(
34     user: UserIn,
35     db: Annotated[Dict, Depends(get_db)]
36 ):
37     # do something, e.g. create user in database
38     db[1] = user
39
40     return {"message": "User created successfully", "id": 1}
```

INPUTS WITH FASTAPI

```
27 class UserIn(BaseModel):
28     name: str
29     email: str
30     password: str
31
32 @app.post("/user", status_code=status.HTTP_201_CREATED)
33 def create_user(
34     user: UserIn,
35     db: Annotated[Dict, Depends(get_db)]
36 ):
37     # do something, e.g. create user in database
38     db[1] = user
39
40     return {"message": "User created successfully", "id": 1}
```

INPUTS WITH FASTAPI

```
27 class UserIn(BaseModel):
28     name: str
29     email: str
30     password: str
31
32 @app.post("/user", status_code=status.HTTP_201_CREATED)
33 def create_user(
34     user: UserIn,
35     db: Annotated[Dict, Depends(get_db)]
36 ):
37     # do something, e.g. create user in database
38     db[1] = user
39
40     return {"message": "User created successfully", "id": 1}
```

INPUTS WITH FASTAPI

```
27 class UserIn(BaseModel):
28     name: str
29     email: str
30     password: str
31
32 @app.post("/user", status_code=status.HTTP_201_CREATED)
33 def create_user(
34     user: UserIn,
35     db: Annotated[Dict, Depends(get_db)]
36 ):
37     # do something, e.g. create user in database
38     db[1] = user
39
40     return {"message": "User created successfully", "id": 1}
```



```
$ curl -X 'POST' \
  'http://localhost:8000/user' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{"name": "Orell", "email": "hello@orellgarten.com", "password": "test"}'
{"message": "User created successfully", "id": 1}
```

OUTPUTS WITH FASTAPI

```
42 class UserOut(BaseModel):
43     name: str
44     email: str
45
46 @app.get("/user/{id}",
47         response_model=UserOut,
48         status_code=status.HTTP_200_OK
49     )
50 def get_user(
51     id: int,
52     db: Annotated[Dict, Depends(get_db)]
53 ):
54     # get user from database
55     user = db.get(id)
56
57     return UserOut.model_validate(user, from_attributes=True)
```

OUTPUTS WITH FASTAPI

```
42 class UserOut(BaseModel):
43     name: str
44     email: str
45
46 @app.get("/user/{id}",
47         response_model=UserOut,
48         status_code=status.HTTP_200_OK
49     )
50 def get_user(
51     id: int,
52     db: Annotated[Dict, Depends(get_db)]
53 ):
54     # get user from database
55     user = db.get(id)
56
57     return UserOut.model_validate(user, from_attributes=True)
```

OUTPUTS WITH FASTAPI

```
42 class UserOut(BaseModel):
43     name: str
44     email: str
45
46 @app.get("/user/{id}",
47         response_model=UserOut,
48         status_code=status.HTTP_200_OK
49     )
50 def get_user(
51     id: int,
52     db: Annotated[Dict, Depends(get_db)]
53 ):
54     # get user from database
55     user = db.get(id)
56
57     return UserOut.model_validate(user, from_attributes=True)
```

OUTPUTS WITH FASTAPI

```
42 class UserOut(BaseModel):
43     name: str
44     email: str
45
46 @app.get("/user/{id}",
47         response_model=UserOut,
48         status_code=status.HTTP_200_OK
49     )
50 def get_user(
51     id: int,
52     db: Annotated[Dict, Depends(get_db)]
53 ):
54     # get user from database
55     user = db.get(id)
56
57     return UserOut.model_validate(user, from_attributes=True)
```

```
$ curl -X 'GET' 'http://localhost:8000/user/1'
{"name": "Orell", "email": "hello@orellgarten.com"}
```

OUTPUTS WITH FASTAPI

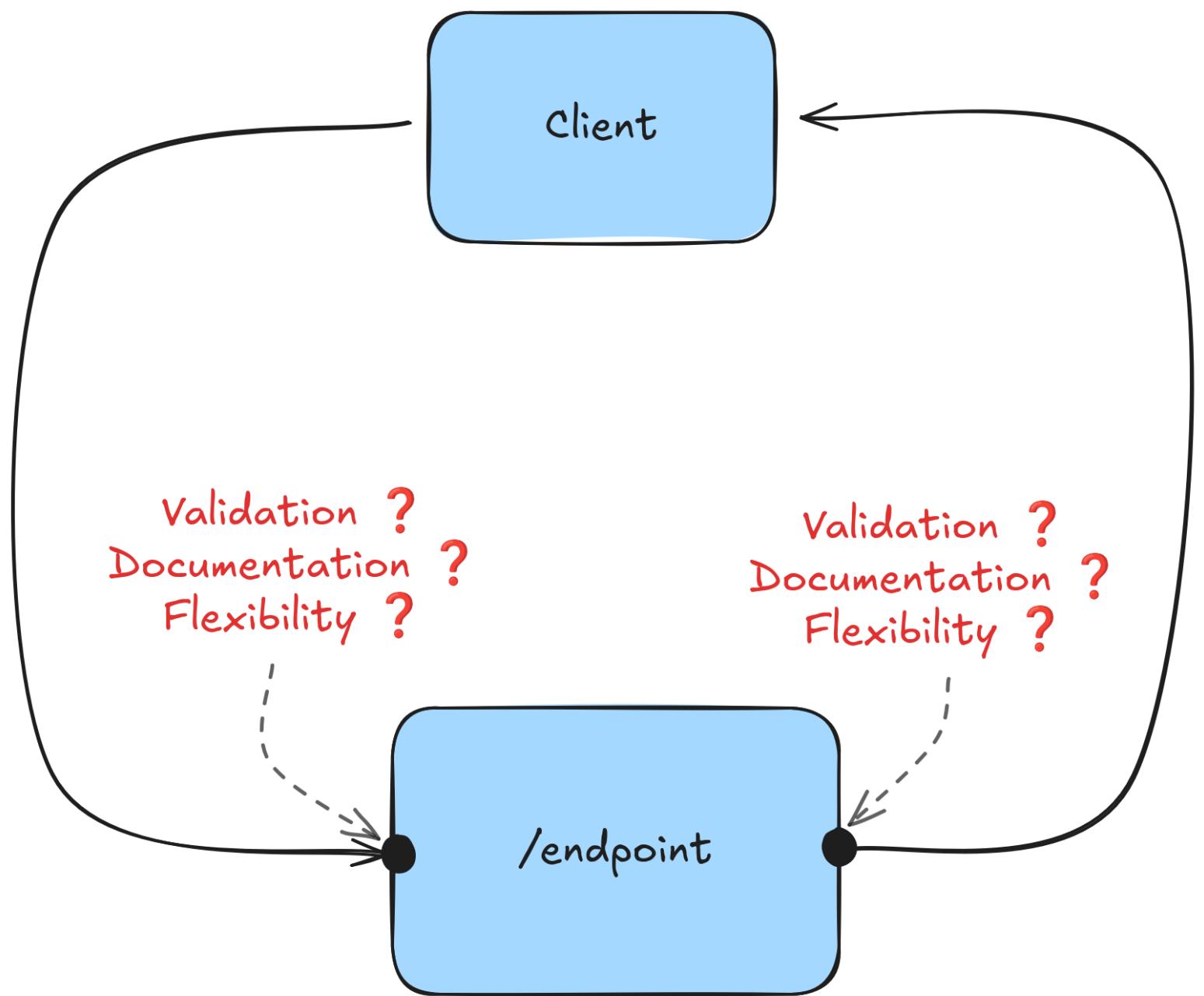
```
42 class UserOut(BaseModel):
43     name: str
44     email: str
45
46 @app.get("/user/{id}",
47         response_model=UserOut,
48         status_code=status.HTTP_200_OK
49     )
50 def get_user(
51     id: int,
52     db: Annotated[Dict, Depends(get_db)]
53 ):
54     # get user from database
55     user = db.get(id)
56
57     return UserOut.model_validate(user, from_attributes=True)
```

```
$ curl -X 'GET' 'http://localhost:8000/user/1'
{"name": "Orell", "email": "hello@orellgarten.com"}
```

Data Transfer Object: Password not leaked

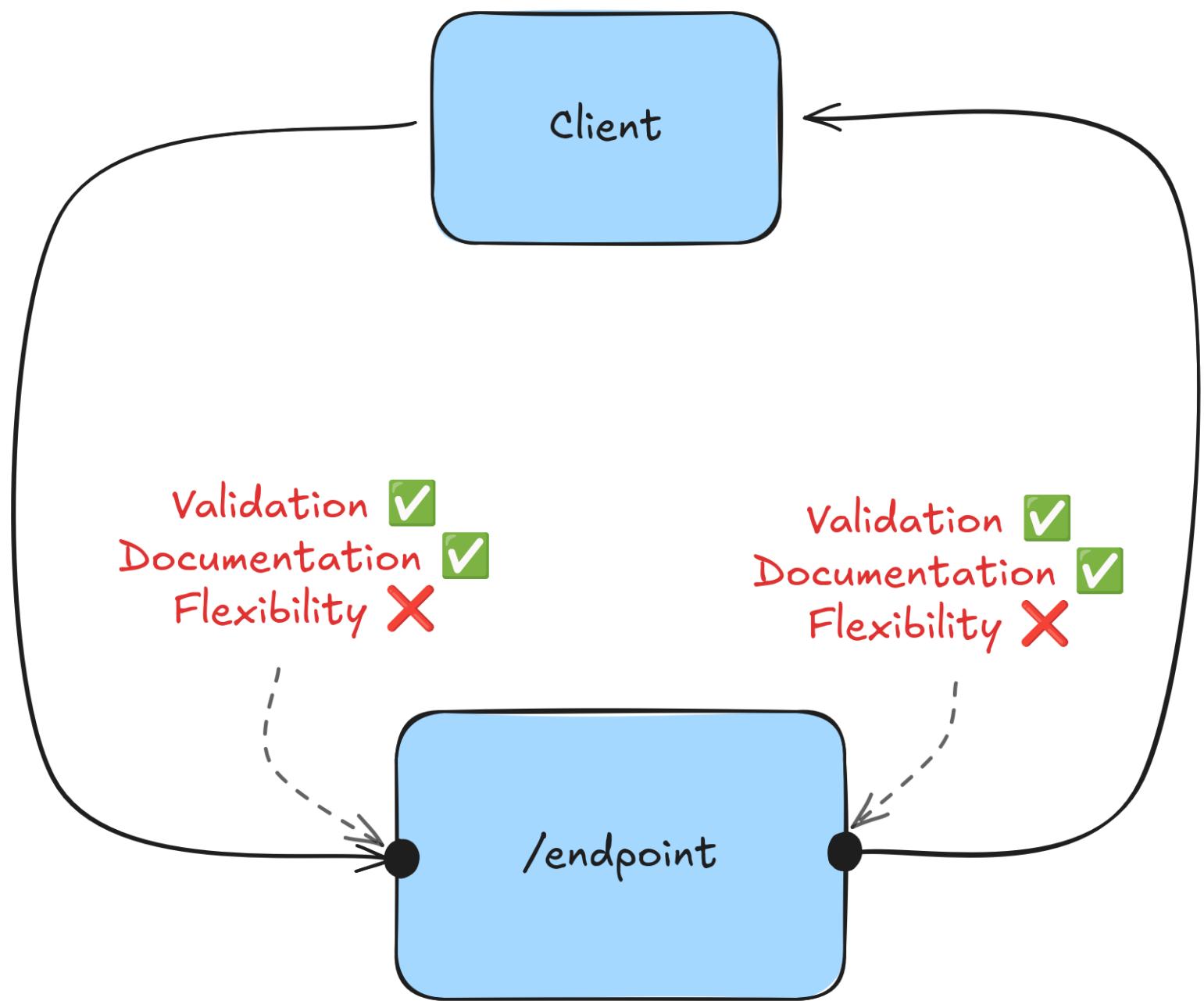
FASTAPI IN/OUT

- Works with Pydantic Models
- Automatic OpenAPI docs



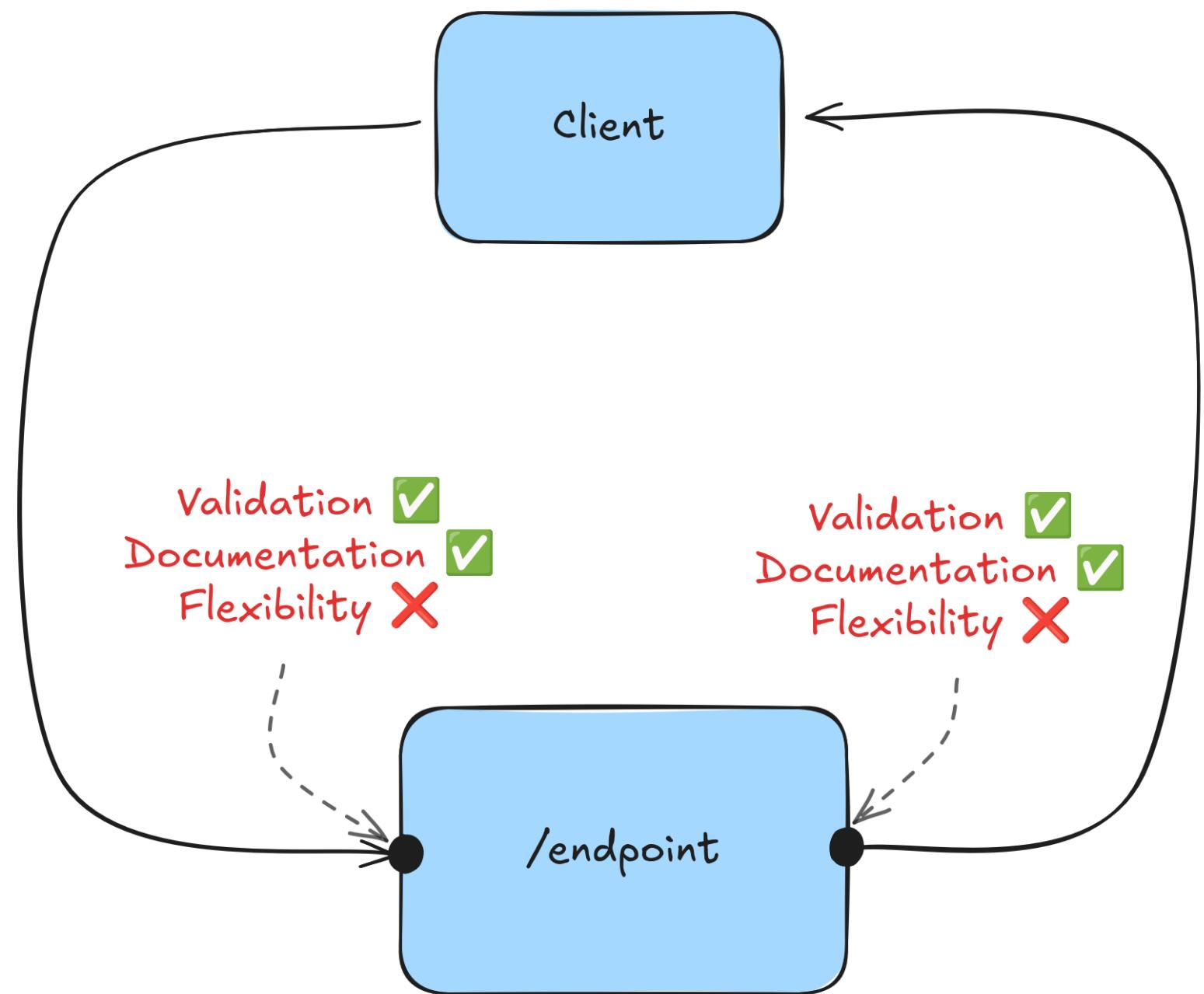
FASTAPI IN/OUT

- Works with Pydantic Models
- Automatic OpenAPI docs



FASTAPI IN/OUT

- Works with Pydantic Models
- Automatic OpenAPI docs



- Data modeling forces you to think about your application

LEARNING 1

Data modeling is not something you can do, it's something you must do.



async IS OVERRATED



ASYNC MEANS HIGH PERFORMANCE



ASYNC MEANS HIGH PERFORMANCE?

THE PROBLEM

Code in tutorials usually looks like this:

```
1 from fastapi import FastAPI  
2  
3 app = FastAPI()  
4  
5 @app.get("/")  
6 async def root():  
7     return {"message": "Hello World"}
```

THE PROBLEM

Code in tutorials usually looks like this:

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 async def root():
7     return {"message": "Hello World"}
```

ASYNC



- Single-threaded event loop
- Async runtime makes sure event loop is busy

THE PROBLEM

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 async def root():
7     time.sleep(5) # Simulate long-running blocking op
8     return {"message": "Hello World"}
```

What will happen to here?

THE PROBLEM

```
1 from fastapi import FastAPI  
2  
3 app = FastAPI()  
4  
5 @app.get("/")  
6 async def root():  
7     time.sleep(5) # Simulate long-running blocking op  
8     return {"message": "Hello World"}
```

What will happen to here?

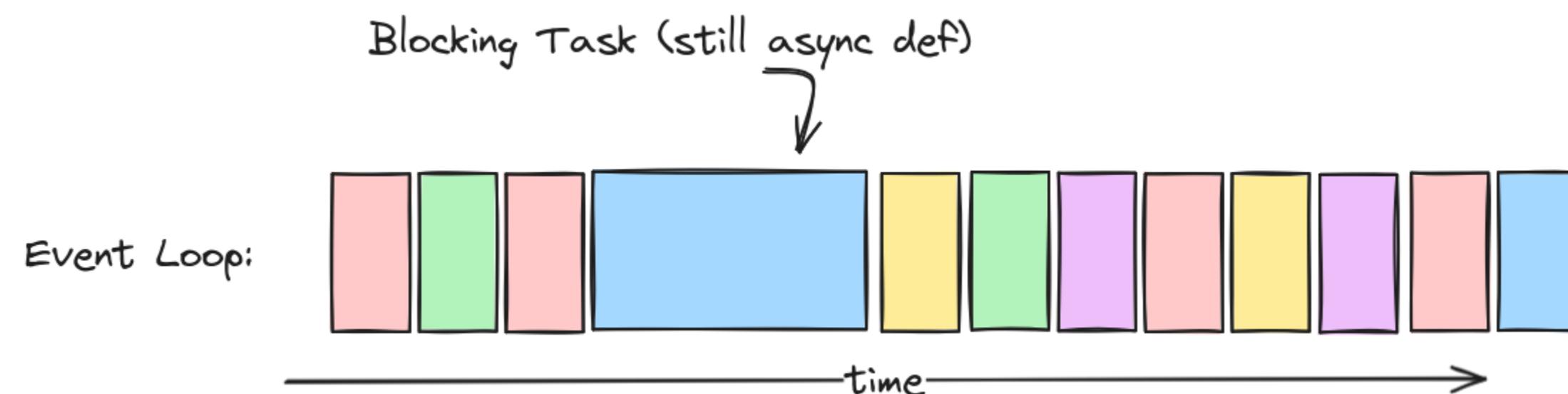
ASYNC



ASYNC



ASYNC



The entire API is unresponsive

THE SOLUTION

- Only use awaitable functions (coroutines)
- blocking functions need to be fast

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 async def root():
7     await asyncio.sleep(5) # long-running non-blocking operation
8     return {"message": "Hello World"}
9 ````
```

THE SOLUTION

- Only use awaitable functions (coroutines)
- blocking functions need to be fast

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 async def root():
7     await asyncio.sleep(5) # long-running non-blocking operation
8     return {"message": "Hello World"}
9 ````
```

THE OTHER SOLUTION

- Do not use `async` and fastAPI knows what to do:

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 def root():
7     time.sleep(5) # long-running blocking operation
8     return {"message": "Hello World"}
```

- This does not block the API
- Path operation runs in thread-pool and is awaited

THE OTHER SOLUTION

- Do not use `async` and fastAPI knows what to do:

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 def root():
7     time.sleep(5) # long-running blocking operation
8     return {"message": "Hello World"}
```

- This does not block the API
- Path operation runs in thread-pool and is awaited

LEARNING 2: `async` DOES NOT MEAN HIGH PERFORMANCE

REMARKS

- Async require a different mental model of programming
- Easy to make mistakes, e.g. call blocking functions by accident
- Async is not the silver bullet for performance

WHEN TO USE `async`?

- Only if you are sure it's non-blocking
- Middleware and dependencies
- Trivial code

LEARNING 3: IF IN DOUBT USE `def` INSTEAD OF `async def`.

PROBLEMS YOU WILL ENCOUNTER

OVERVIEW

- Problem 1) Global objects
- Problem 2) Server-side sessions
- Problem 3) Flask plugins

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

Not-untypical deep inside the call stack in a Flask application:

```
1 # core/processing/files.py
2 def enrich_file(file):
3
4     # TODO: don't use request here
5     tag = request.args.get('tag', 'unidentified')
6     timestamp = datetime.utcnow().isoformat()
7
8     # do more things
9
10    return file
```

PROBLEMS?

- `request` being used in modules relatively unrelated to actual API
- `tag` is a query parameter!

1) FLASK - GLOBAL OBJECTS

- request and g are global states available in the entire call-chain
- Find use of global state in your application
 - grep -nri request (request context)
 - grep -nri 'import g' for application context

1) FASTAPI - DEPENDENCY INJECTION

- Solution in FastAPI:
 - Pydantic data models for `flask.request` (Data Modeling)
 - Dependency injection for `flask.g`

1) FASTAPI - DEPENDENCY INJECTION

- Solution in FastAPI:
 - Pydantic data models for `flask.request` (Data Modeling)
 - Dependency injection for `flask.g`

```
1 def get_db():
2     with sessionmaker() as session:
3         yield session
4
5 @app.post("/user", status_code=status.HTTP_201_CREATED)
6 def create_user(
7     user: UserIn,
8     db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
9 ):
10     ...
```

1) FASTAPI - DEPENDENCY INJECTION

- Solution in FastAPI:
 - Pydantic data models for `flask.request` (Data Modeling)
 - Dependency injection for `flask.g`

```
1 def get_db():
2     with sessionmaker() as session:
3         yield session
4
5 @app.post("/user", status_code=status.HTTP_201_CREATED)
6 def create_user(
7     user: UserIn,
8     db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
9 ):
10     ...
```

1) FASTAPI - DEPENDENCY INJECTION

- Solution in FastAPI:
 - Pydantic data models for `flask.request` (Data Modeling)
 - Dependency injection for `flask.g`

```
1 def get_db():
2     with sessionmaker() as session:
3         yield session
4
5 @app.post("/user", status_code=status.HTTP_201_CREATED)
6 def create_user(
7     user: UserIn,
8     db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
9 ):
10     ...
```

1) FASTAPI - DEPENDENCY INJECTION

- Solution in FastAPI:
 - Pydantic data models for `flask.request` (Data Modeling)
 - Dependency injection for `flask.g`

```
1 def get_db():
2     with sessionmaker() as session:
3         yield session
4
5 @app.post("/user", status_code=status.HTTP_201_CREATED)
6 def create_user(
7     user: UserIn,
8     db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
9 ):
10     ...
```

2) SERVER-SIDE-SESSIONS

- Provided by the `flask-session` package
- Supports various backends for session storage
- Use like a key-value storage object
- Automatically sets cookies

2) SERVER-SIDE-SESSIONS

- Provided by the `flask-session` package
- Supports various backends for session storage
- Use like a key-value storage object
- Automatically sets cookies

No equivalent solution in FastAPI:

2) SERVER-SIDE-SESSIONS

- Provided by the `flask-session` package
- Supports various backends for session storage
- Use like a key-value storage object
- Automatically sets cookies

No equivalent solution in FastAPI:

- Session management needs to be done manually
- Ideally, you don't want server-side session anyway
- MSAL auth is session based

3) PLUGINS

Flask:

- Has plugins for everything
- Install a flask-specific Python package
- Understand how to use that package
- Connect the app and the plugin

3) PLUGINS

Flask:

- Has plugins for everything
- Install a flask-specific Python package
- Understand how to use that package
- Connect the app and the plugin

FastAPI:

- Very few plugins available
- Dependency injection to extent function
- "plug'n'play" with normal packages

MIGRATION STRATEGY

MIGRATION STRATEGY

Migration Strategy

1. Create test cases for all routes

Migration Strategy

1. Create test cases for all routes
2. Create endpoints in FastAPI app

MIGRATION STRATEGY

1. Create test cases for all routes
2. Create endpoints in FastAPI app
3. Identify locations where global state is used

MIGRATION STRATEGY

1. Create test cases for all routes
2. Create endpoints in FastAPI app
3. Identify locations where global state is used
4. Move that state to endpoint and explicitly call functions with required information

Migration Strategy

1. Create test cases for all routes
2. Create endpoints in FastAPI app
3. Identify locations where global state is used
4. Move that state to endpoint and explicitly call functions with required information
5. Identify input and output models and create Pydantic models

Migration Strategy

1. Create test cases for all routes
2. Create endpoints in FastAPI app
3. Identify locations where global state is used
4. Move that state to endpoint and explicitly call functions with required information
5. Identify input and output models and create Pydantic models
6. Implement fastAPI endpoint

THANK YOU & LET'S CONNECT



hello@orellgarten.com



linkedin.com/in/ogarten



Material