LEARNINGS FROM MIGRATING A FLASK APP TO FASTAPI

PyCon DE & PyData

Orell Garten

24.04.2025

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 testibility

Migration to FastAPI was helping us achieve this.

DISCLAIMER

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

/AGENDA

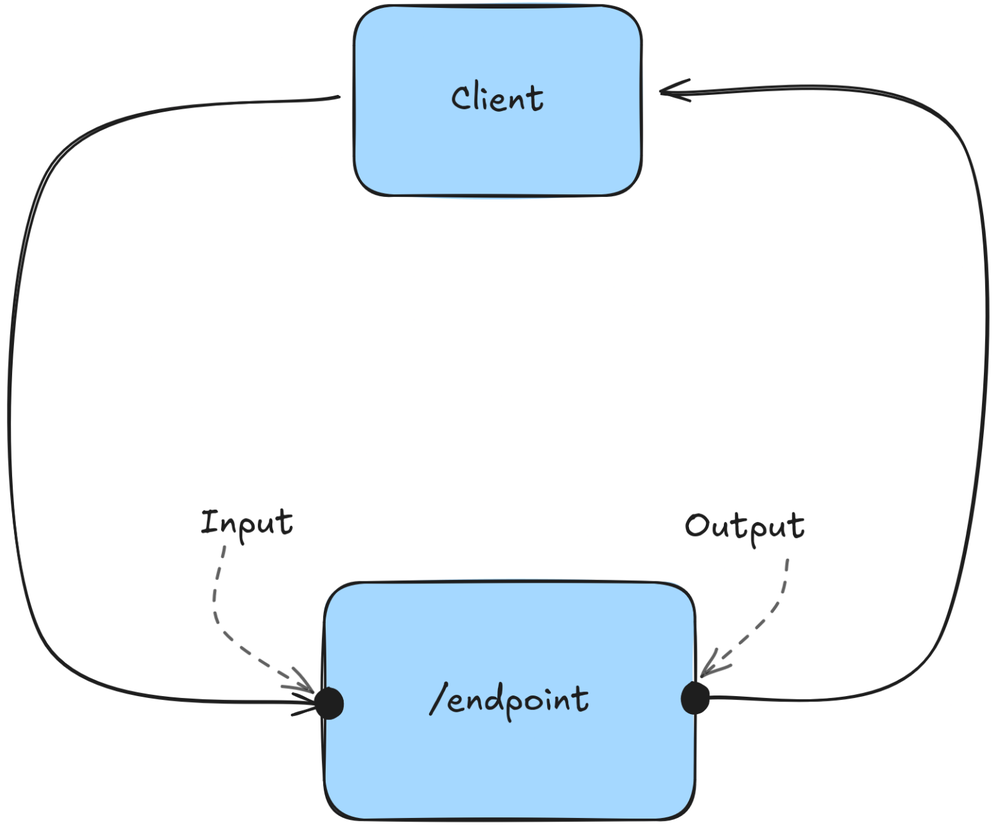
1. Data modeling
2. async is overrated
3. Problems you will encounter
4. Migration strategy
5. Q&A

MATERIAL

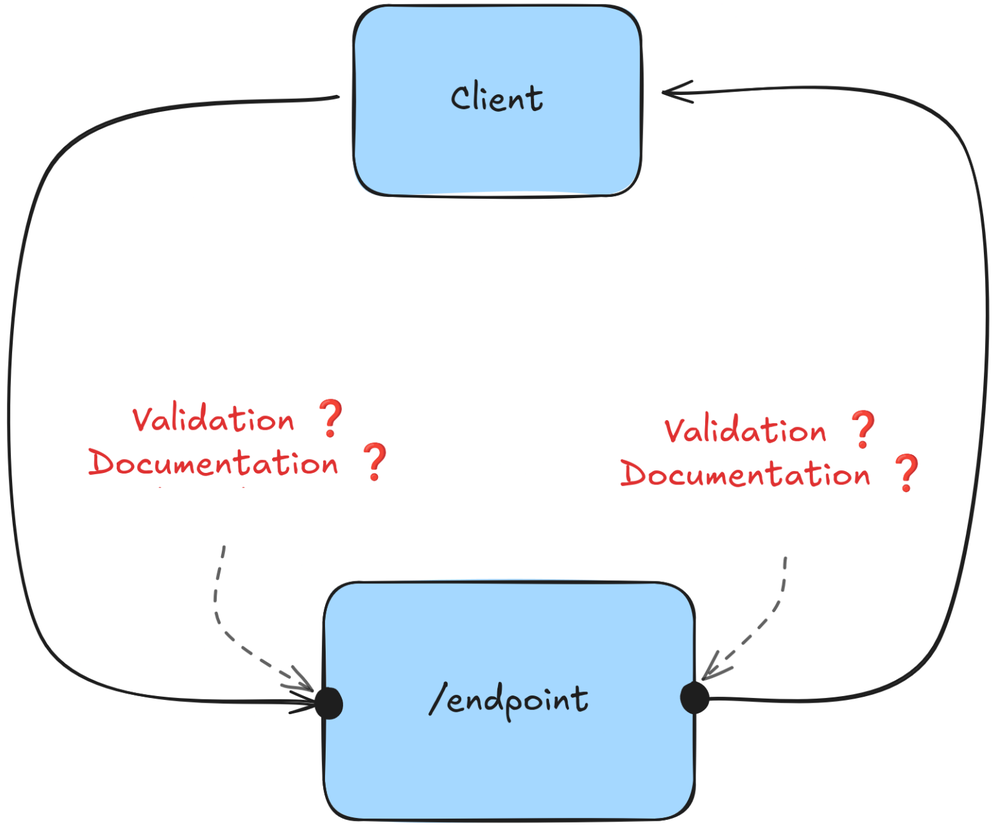
Material on github.com/orgarten

DATA MODELING

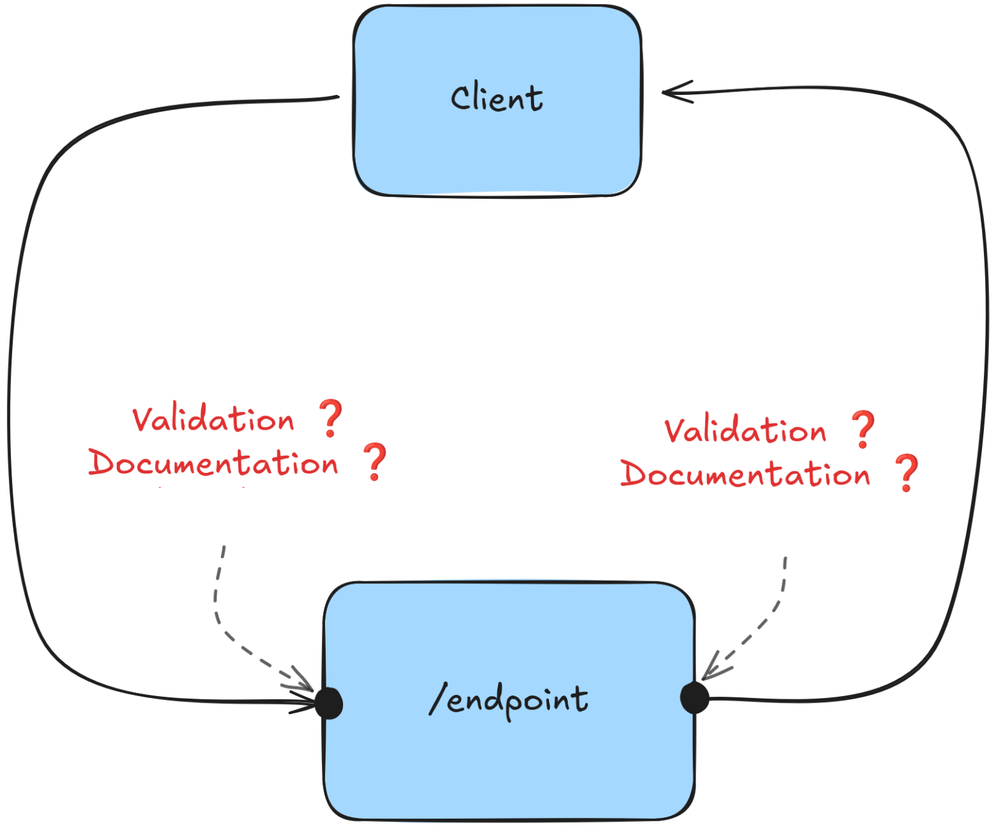
THE SITUATION



THE SITUATION



THE SITUATION



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

INPUTS WITH FLASK

```python
@app.route('/user', methods=['POST'])
def create_user():
    data = request.get_json()
    username = data.get('name')
    email = data.get('email')
    password = data.get('password')

    # Create user in DB
    db[1] = {"name": username, "email": email, "password": password}

    return jsonify({'message': 'User created successfully','id': 1}), 201
```

INPUTS WITH FLASK

```python
@app.route('/user', methods=['POST'])
def create_user():
    data = request.get_json()
    username = data.get('name')
    email = data.get('email')
    password = data.get('password')

    # Create user in DB
    db[1] = {"name": username, "email": email, "password": password}

    return jsonify({'message': 'User created successfully','id': 1}), 201
```

INPUTS WITH FLASK

```python
@app.route('/user', methods=['POST'])
def create_user():
    data = request.get_json()
    username = data.get('name')
    email = data.get('email')
    password = data.get('password')

    # Create user in DB
    db[1] = {"name": username, "email": email, "password": password}

    return jsonify({'message': 'User created successfully','id': 1}), 201
```

INPUTS WITH FLASK

```python
@app.route('/user', methods=['POST'])
def create_user():
    data = request.get_json()
    username = data.get('name')
    email = data.get('email')
    password = data.get('password')

    # Create user in DB
    db[1] = {"name": username, "email": email, "password": password}

    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

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

    return user
```

OUTPUTS WITH FLASK

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

    return user
```

OUTPUTS WITH FLASK

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

    return user
```

OUTPUTS WITH FLASK

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

    return user
```

OUTPUTS WITH FLASK

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

    return user
```

```
$ curl -X 'GET' 'http://localhost:8000/user/1'

{"email":"hello@orellgarten.com","name":"Orell","password":"test"}
```

OUTPUTS WITH FLASK

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

    return user
```

```
$ curl -X 'GET' 'http://localhost:8000/user/1'

{"email":"hello@orellgarten.com","name":"Orell","password":"test"}
```

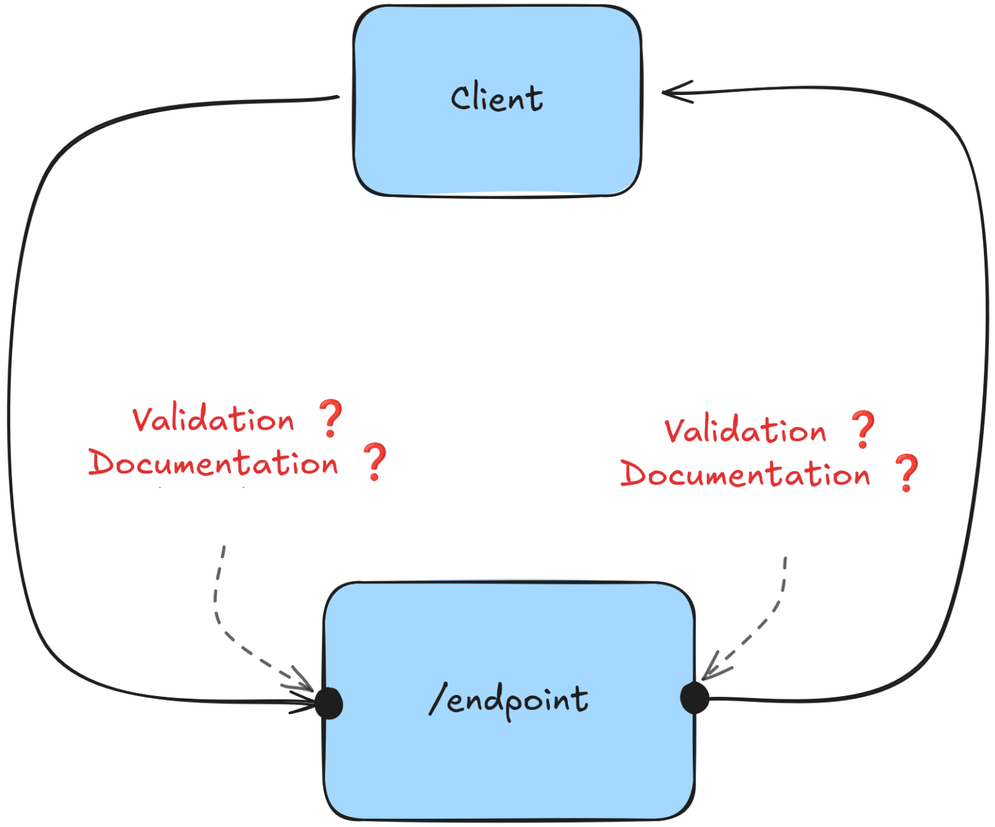
Oops, password in response!

FLASK IN/OUT

- Work with `request` object
- Typing not required

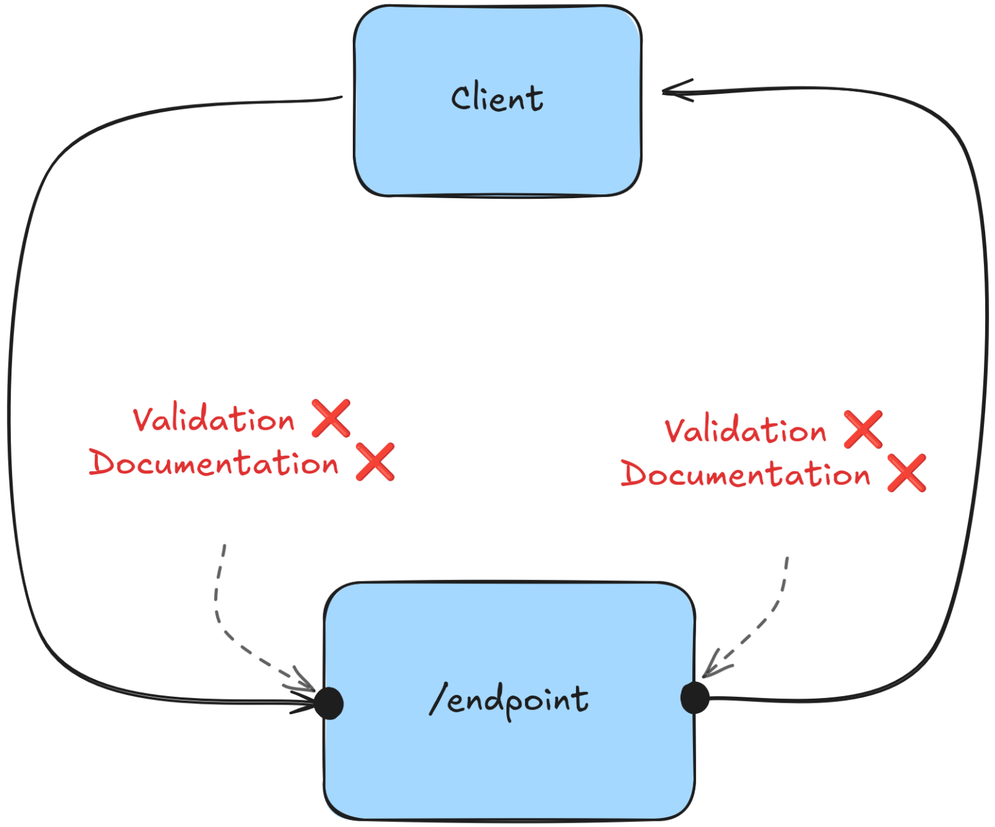
FLASK IN/OUT

- Work with `request` object
- Typing not required



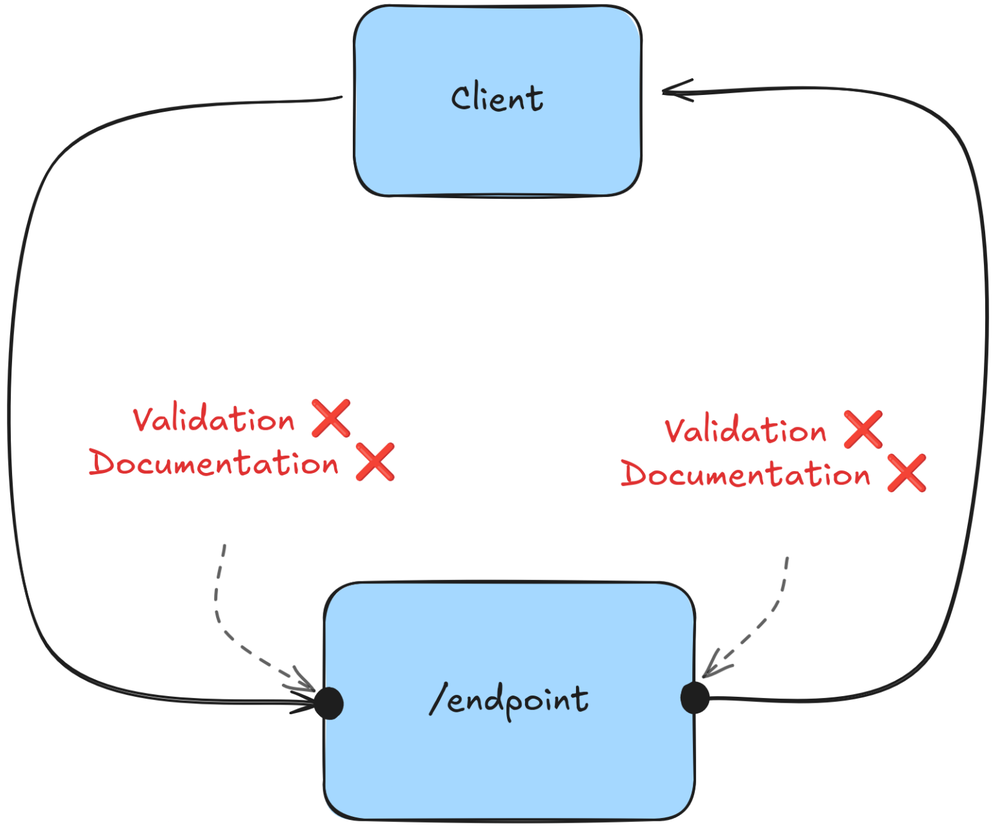
FLASK IN/OUT

- 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

```python
class UserIn(BaseModel):
    name: str
    email: str
    password: str

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # do something, e.g. create user in database
    db[1] = user

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

INPUTS WITH FASTAPI

```python
class UserIn(BaseModel):
    name: str
    email: str
    password: str

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # do something, e.g. create user in database
    db[1] = user

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

INPUTS WITH FASTAPI

```python
class UserIn(BaseModel):
    name: str
    email: str
    password: str

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # do something, e.g. create user in database
    db[1] = user

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

INPUTS WITH FASTAPI

```python
class UserIn(BaseModel):
    name: str
    email: str
    password: str

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # do something, e.g. create user in database
    db[1] = user

    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

```python
class UserOut(BaseModel):
    name: str
    email: str

@app.get("/user/{id}",
     response_model=UserOut,
     status_code=status.HTTP_200_OK
 )
def get_user(
    id: int,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # get user from database
    user = db.get(id)

    return UserOut.model_validate(user, from_attributes=True)
```

OUTPUTS WITH FASTAPI

```python
class UserOut(BaseModel):
    name: str
    email: str

@app.get("/user/{id}",
     response_model=UserOut,
     status_code=status.HTTP_200_OK
 )
def get_user(
    id: int,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # get user from database
    user = db.get(id)

    return UserOut.model_validate(user, from_attributes=True)
```

OUTPUTS WITH FASTAPI

```python
class UserOut(BaseModel):
    name: str
    email: str

@app.get("/user/{id}",
     response_model=UserOut,
     status_code=status.HTTP_200_OK
 )
def get_user(
    id: int,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # get user from database
    user = db.get(id)

    return UserOut.model_validate(user, from_attributes=True)
```

OUTPUTS WITH FASTAPI

```python
class UserOut(BaseModel):
    name: str
    email: str

@app.get("/user/{id}",
     response_model=UserOut,
     status_code=status.HTTP_200_OK
 )
def get_user(
    id: int,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # get user from database
    user = db.get(id)

    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

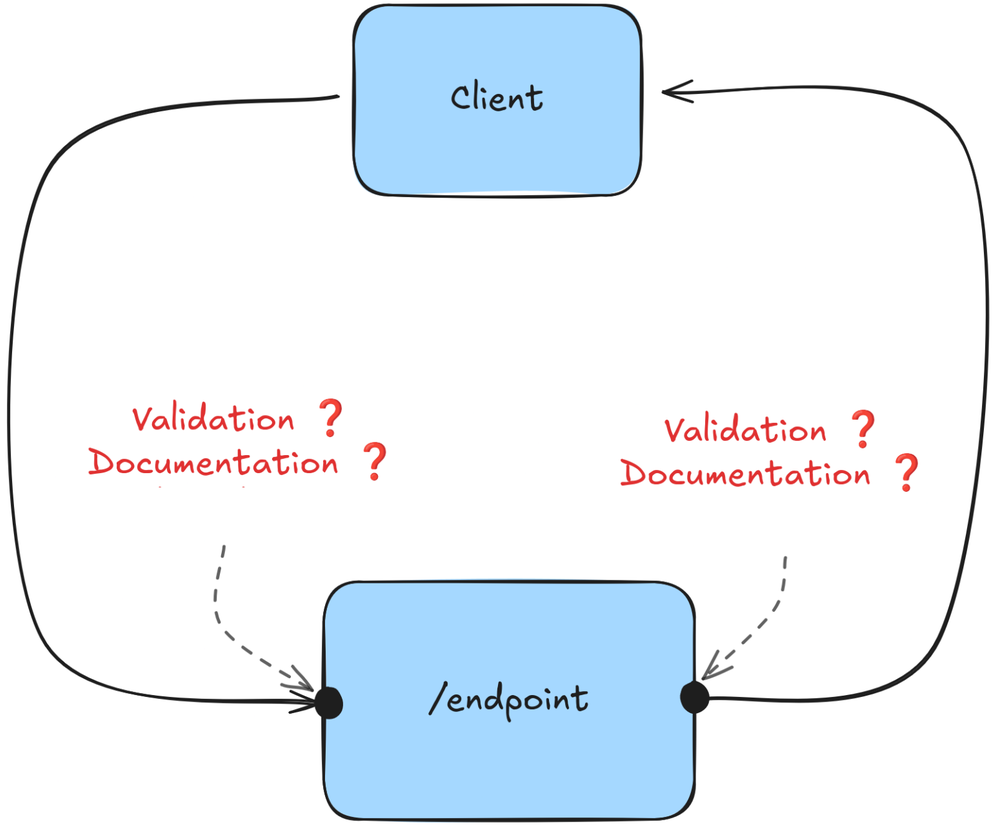
```python
class UserOut(BaseModel):
    name: str
    email: str

@app.get("/user/{id}",
     response_model=UserOut,
     status_code=status.HTTP_200_OK
 )
def get_user(
    id: int,
    db: Annotated[Dict, Depends(get_db)]
 ):
    # get user from database
    user = db.get(id)

    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

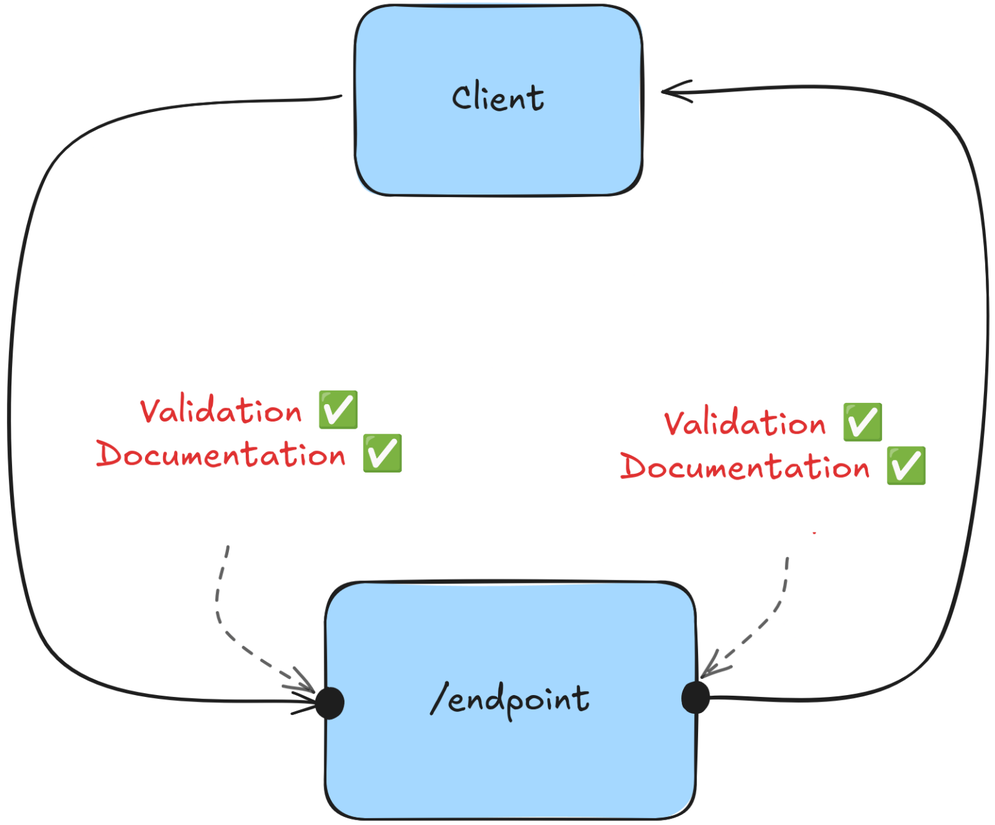
FASTAPI IN/OUT

- Works with Pydantic Models
- Automatic OpenAPI docs



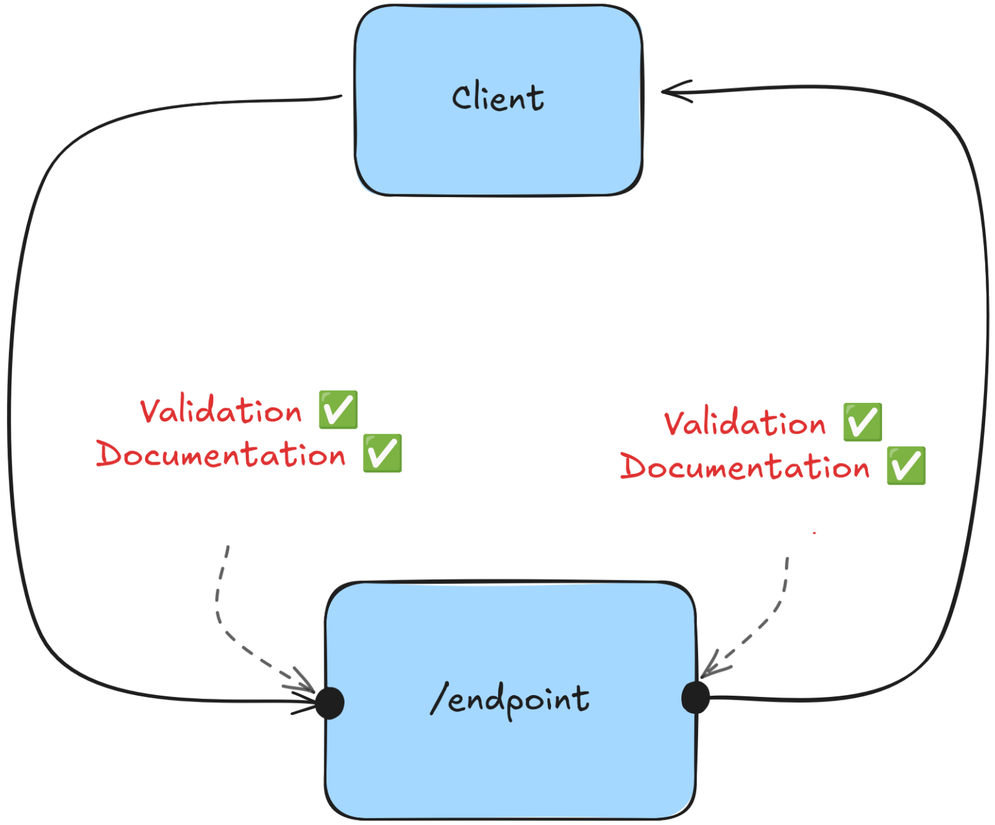
FASTAPI IN/OUT

- 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:

```python
from fastapi import FastAPI

app = FastAPI()

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

THE PROBLEM

Code in tutorials usually looks like this:

```python
from fastapi import FastAPI

app = FastAPI()

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

ASYNC

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

THE PROBLEM

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    time.sleep(5) # Simulate long-running blocking op
    return {"message": "Hello World"}
```

What will happen here?

THE PROBLEM

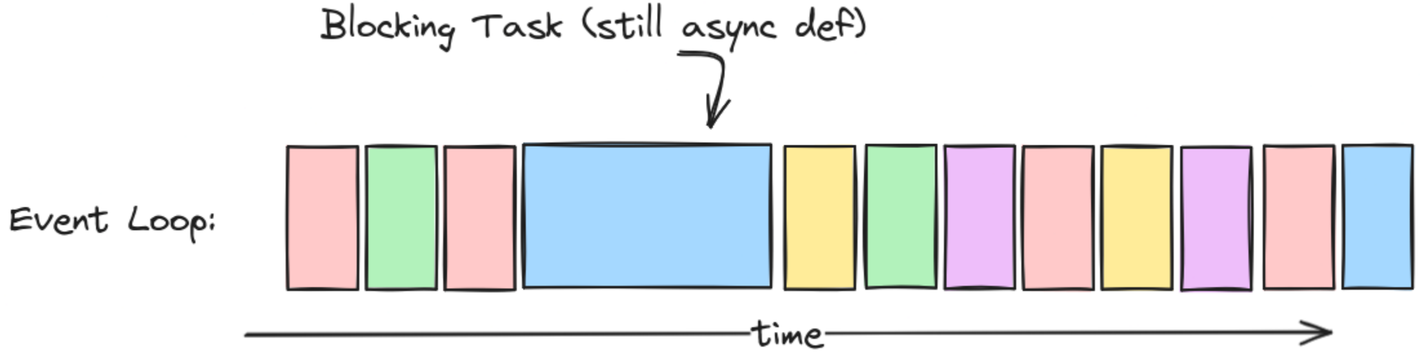
```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    time.sleep(5) # Simulate long-running blocking op
    return {"message": "Hello World"}
```

What will happen here?

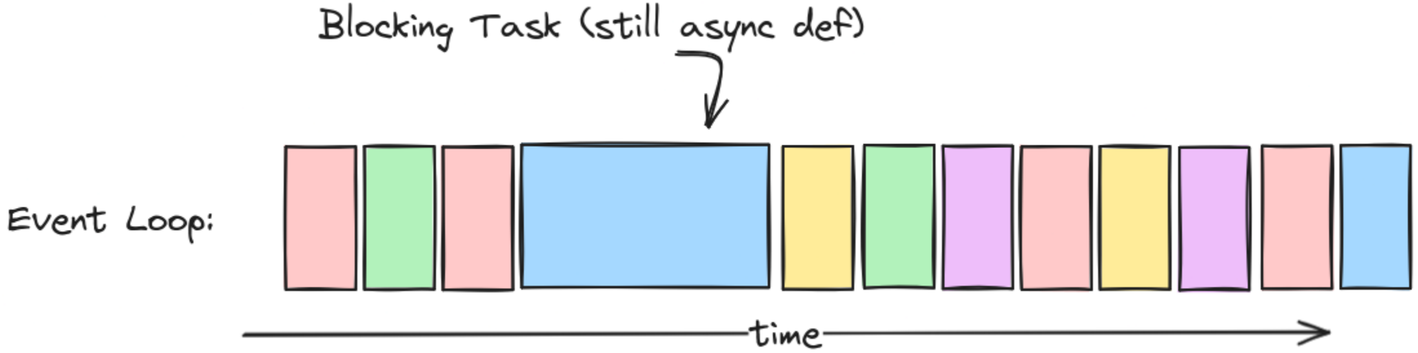
ASYNC

ASYNC


Async Task (async def)
Event Loop:
time


Blocking Task (still async def)
Event Loop:
time

ASYNC



The entire API is unresponsive

THE SOLUTION

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

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    await asyncio.sleep(5) # long-running non-blocking operation
    return {"message": "Hello World"}
```

THE SOLUTION

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

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    await asyncio.sleep(5) # long-running non-blocking operation
    return {"message": "Hello World"}
```

THE OTHER SOLUTION

- Do not use async and FastAPI knows what to do:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    time.sleep(5) # long-running blocking operation
    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:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    time.sleep(5) # long-running blocking operation
    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?

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:

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

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

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

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

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

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

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

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

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

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

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    return file
```

PROBLEMS?

1) FLASK - GLOBAL OBJECTS

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

```python
# core/processing/files.py
def enrich_file(file):

    # TODO: don't use request here
    tag = request.args.get('tag', 'unidentified')

    # do more things

    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 you 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`

```python
def get_db():
    with sessionmaker() as session:
        yield session

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
 ):
    ...
```

1) FASTAPI - DEPENDENCY INJECTION

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

```python
def get_db():
    with sessionmaker() as session:
        yield session

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
 ):
    ...
```

1) FASTAPI - DEPENDENCY INJECTION

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

```python
def get_db():
    with sessionmaker() as session:
        yield session

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
 ):
    ...
```

1) FASTAPI - DEPENDENCY INJECTION

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

```python
def get_db():
    with sessionmaker() as session:
        yield session

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user(
    user: UserIn,
    db: Annotated[Dict, Depends(get_db)] # DB session as dependency injection
 ):
    ...
```

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 explicitely 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 explicitely 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 explicitely 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