FastAPI automated testing

INTRODUCTION TO FASTAPI

Matt Eckerle
Software and Data Engineering Leader

What Are Automated Tests?

Unit Tests

- **Focus:** Isolated code
- **Purpose:** Validate code function
- **Scope:** Function or method
- **Environment:** Isolated Python env

```
def test_main():
    response = main()
    assert response == {"msg": "Hello"}
```

System Tests

- **Focus:** Isolated system operations
- **Purpose:** Validate system function
- **Scope:** Endpoint
- **Environment:** Python env with app running

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

Using TestClient

`TestClient` : HTTP client for `pytest`

```
# Import TestClient and app
from fastapi.testclient import TestClient
from .main import app

# Create test client with application context
client = TestClient(app)

def test_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"msg": "Hello"}
```

Testing Error or Failure Responses

App

```
app = FastAPI()

@app.delete("/items")
def delete_item(item: Item):
    if item.id not in item_ids:
        raise HTTPException(
            status_code=404,
            detail="Item not found.")
    else:
        delete_item_in_database(item)
        return {}
```

Test

```
def test_delete_nonexistent_item():
    response = client.delete(
        "/items",
        json={"id": -999})
    assert response.status_code == 404
    json = response.json()
    assert json == {"detail":
                    "Item not found."}
```

Let's practice!

INTRODUCTION TO FASTAPI

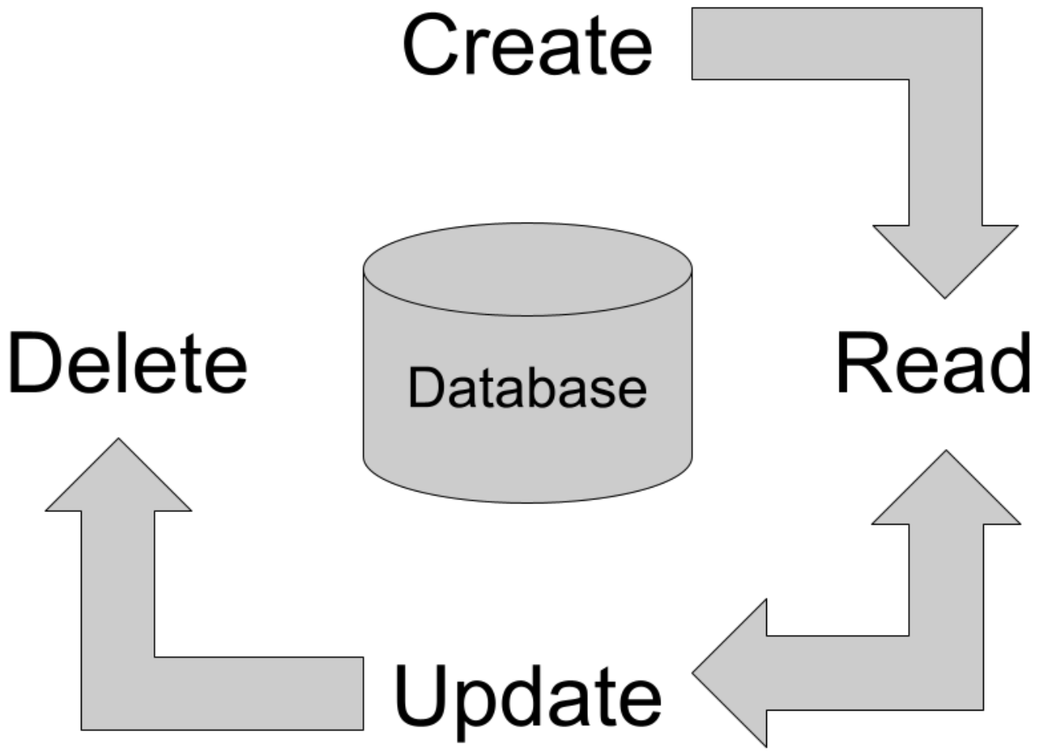
Building a JSON CRUD API

INTRODUCTION TO FASTAPI

Matt Eckerle
Software and Data Engineering Leader

datacamp

Four Steps in Object Management Lifecycle (CRUD)



API Operations

Create

- POST operation

Read

- GET operation

Update

- PUT operation

Delete

- DELETE operation

JSON CRUD API Motivation

Fundamentals

- Manage the entire object lifecycle
- Understand best practices for HTTP API operations
- Design our own data management APIs

Opportunities

- Business logic for more complex data operations
- High throughput data pipelines
- Machine Learning inference pipelines

Building a CRUD Module

```python
from pydantic import BaseModel

class Review(BaseModel):
    movie: str
    num_stars: int
    text: str

class DbReview(BaseModel):
    movie: str
    num_stars: int
    text: str
    # Reference database ID of Reviews
    review_id: int
```

```python
# crud.py
def create_review(review: Review):
    # Create review in database

def read_review(review_id: int):
    # Read review from database

def update_review(review: DbReview):
    # Update review in database

def delete_review(review_id: int):
    # Delete review from database
```

POST Endpoint to Create

- Endpoint: `/reviews`
- Input: `Review`
- Output: `DbReview`

```
@app.post("/reviews", response_model=DbReview)
def create_review(review: Review):
    # Create the movie review in the database
    db_review = crud.create_review(review)
    # Return the created review with database ID
    return db_review
```

GET Endpoint to Read

- Endpoint: `/reviews`
- Input: `?review_id=1234`
- Output: `DbReview`

```
@app.get("/reviews", response_model=DbReview)
def read_review(review_id: int):
    # Read the movie review from the database
    db_review = crud.read_review(review_id)
    # Return the review
    return db_review
```

PUT Endpoint to Update

- Endpoint: `/reviews`
- Input: `DbReview`
- Output: `DbReview`

```python
@app.put("/reviews", response_model=DbReview)
def update_review(review: DbReview):
    # Update the movie review in the database
    db_review = crud.update_review(review)
    # Return the updated review
    return db_review
```

DELETE Endpoint to Delete

- Endpoint: `/reviews`
- Input: `DbReview`
- Output: `{}`

```
@app.delete("/reviews")
def delete_review(review: DbReview):
    # Delete the movie review from the database
    crud.delete_review(review.review_id)
    # Return nothing since the data is gone
    return {}
```

Let's practice!

INTRODUCTION TO FASTAPI

Writing a manual functional test

INTRODUCTION TO FASTAPI

Matt Eckerle
Software and Data Engineering Leader

What Are Functional Tests?

System Tests

- **Focus:** Isolated system operations
- **Purpose:** Validate system function
- **Scope:** Endpoint
- **Environment:** Python env with app running

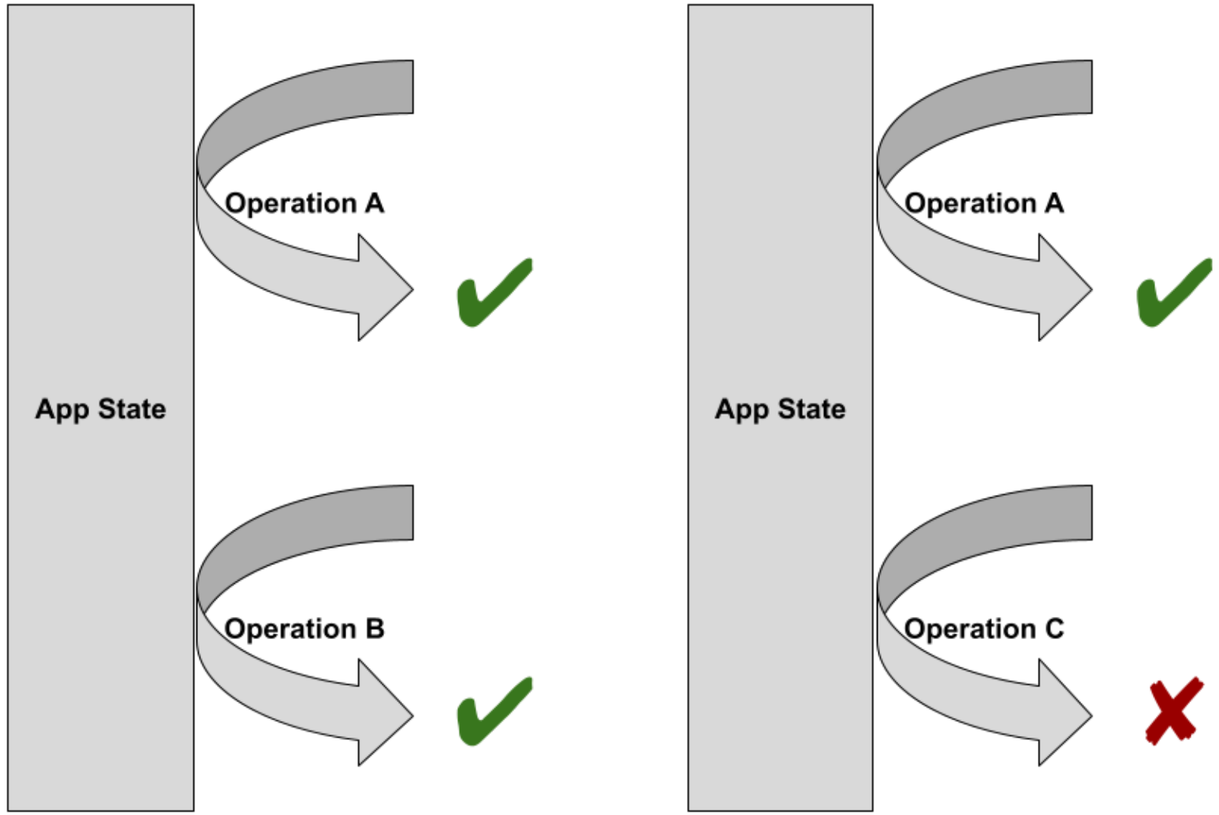
```
def test_read():
    response = client.get("/items/1")
    assert response.status_code == 200
```

Functional Tests

- **Focus:** Integrated system
- **Purpose:** Validate system overall
- **Scope:** Application
- **Environment:** Python env with app running

```
def test_delete_then_read():
    response = client.delete("/items/1")
    assert response.status_code == 200
    response = client.get("/items/1")
    assert response.status_code == 404
```

Test Workflows



Functional Test Workflow Examples

Successful workflows

- Create, then read
- Create, then update
- Create, then delete
- ...

Failing workflows

- Read without create
- Update after delete
- Delete without create
- ...

Functional Test Scripts

- Outside test framework - "Manual tests"
- Use `requests`

```
import requests
ENDPOINT = "http://localhost:8000/items"
# Create item "rock"
r = requests.post(ENDPOINT, json={"name": "rock"})
assert r.status_code == 200
# Get item rock
r = requests.get(ENDPOINT, json={"name": "rock"})
assert r.status_code == 200
```

- Workflows built against known application state

Let's practice!

INTRODUCTION TO FASTAPI

Wrap up

INTRODUCTION TO FASTAPI

Matt Eckerle
Software and Data Engineering Leader

FastAPI Review

- FastAPI key features and use cases
- Four types of HTTP operations
- Building a JSON CRUD API
- Using status codes to communicate success and failure
- Using `async`
- System tests
- Manual functional tests

Congratulations!

INTRODUCTION TO FASTAPI