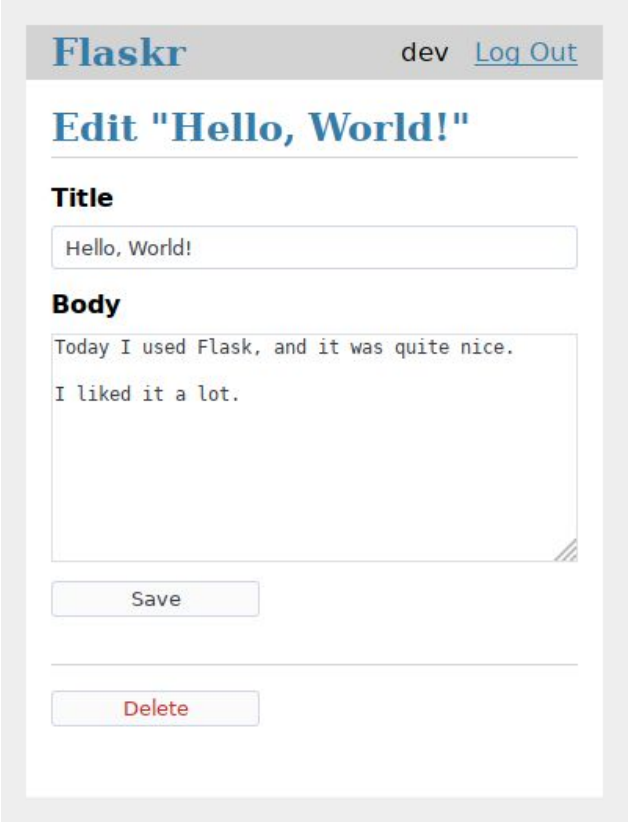
API - Applications

Agenda

- Flask Application Review
- Deploy Flask Application (on Heroku/AWS/Azure)
- Introduction to FastAPI
- Using 3rd Party API: Create and Use Azure Cognitive Service

Flaskr Application Review

- This tutorial will walk you through creating a basic blog application called Flaskr. Users will be able to register, log in, create posts, and edit or delete their own posts. You will be able to package and install the application on other computers.



Project File Structure

```
flaskr/
│
├── flaskr/
│   ├── __init__.py        ──── flask app object defined in __init__.py
│   ├── db.py              ──── db: initialize tables and define flask cli init-db
│   ├── schema.sql         ──── Table schema
│   ├── auth.py            ──── Authentication methods/endpoints
│   ├── blog.py            ──── Blog methods/endpoints
│   ├── templates/         ──── Templates
│   │   ├── base.html
│   │   ├── auth/
│   │   │   ├── login.html
│   │   │   └── register.html
│   │   │
│   │   └── blog/
│   │       ├── create.html
│   │       ├── index.html
│   │       └── update.html
│   │
│   └── static/
│       └── style.css      ──── static files
│
├── tests/                 ──── Test cases
│   ├── conftest.py
│   ├── data.sql
│   ├── test_factory.py
│   ├── test_db.py
│   ├── test_auth.py
│   └── test_blog.py
│
├── venv/                  ──── Virtual environment
│
├── .gitignore
├── setup.py               ──── setup.py for packaging
└── MANIFEST.in
```

FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

- One of the fastest Python frameworks available.
- Based on the open standards for APIs
- Get production-ready code. With automatic interactive documentation.
- pip install fastapi[all]

Flask Vs FastAPI

	FLASK	FastAPI
Sample App		

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

```python
from fastapi import FastAPI

app = FastAPI()


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

Start Server

```
$ export FLASK_APP=hello.py
$ flask run
 * Running on http://127.0.0.1:5000/
```

```
$ uvicorn main:app --reload

INFO: Uvicorn running on http://127.0.0.1:8000 (Press
CTRL+C to quit)
INFO: Started reloader process [28720]
INFO: Started server process [28722]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Demo Project

Installation:

- pip install fastapi[all]
- `pip install uvicorn[standard]`

Features

- Interactive API Docs (Swagger UI)
- Type Data Validation

Typings in FastAPI

```python
from fastapi import FastAPI

app = FastAPI()


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

```json
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

FastAPI takes advantage of typings to **Type checks** and **FastAPI** uses the same declarations to:

- **Define requirements**: from request path parameters, query parameters, headers, bodies, dependencies, etc.
- **Convert data**: from the request to the required type.
- **Validate data**: coming from each request: Generating **automatic errors** returned to the client when the data is invalid.
- **Document** the API using OpenAPI

Pydantic Data Model

```python
from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel


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


app = FastAPI()


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

You can think of models as similar to types in strictly typed languages, or as the requirements of a single endpoint in an API.

Untrusted data can be passed to a model, and after parsing and validation *pydantic* guarantees that the fields of the resultant model instance will conform to the field types defined on the model.

FastAPI: Working with SQL (RDBMS)

- **FastAPI** works with any database and any style of library to talk to the database.
- A common pattern is to use an "ORM": an "object-relational mapping" library. E.g. SQLAlchemy
- With an ORM, you normally create a class that represents a table in a SQL database, each attribute of the class represents a column, with a name and a type.
- For example a ***class Pet*** could represent a SQL table **pets**.
- And each instance object of that class represents a row in the database.

```
.
└── sql_app
    ├── __init__.py
    ├── crud.py
    ├── database.py
    ├── main.py
    ├── models.py
    └── schemas.py
```

ORM Utils (CRUD Operations)
Db config, engine, session
FastAPI main app
Sqlalchemy models
Pydantic CRUD model

Demo: Azure Cognitive Services

Example: Text Analytics Cognitive Service:

Docker Image

Public Cloud API

Ref: https://azure.microsoft.com/en-us/services/cognitive-services/