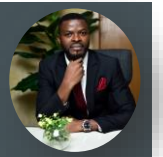
FULL-STACK NANODEGREE SESSION 3

AJIROGHENE SUNDAY



GENERAL DISCUSSION

10 MINS

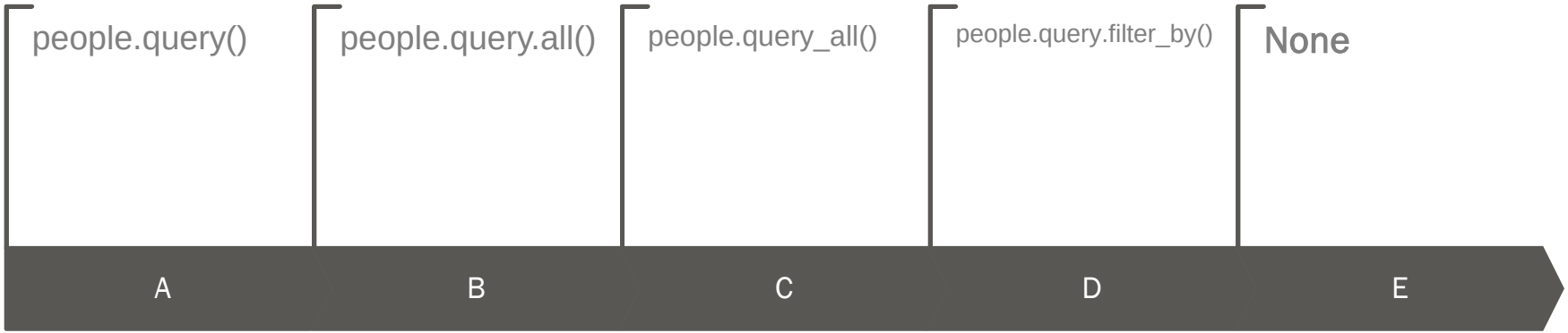
TOPIC REFRESHER

10 MINS

1. **CONSIDER A SIMPLE DATABASE TABLE OF <** People **> WITH THE FOLLOWING RECORDS:**

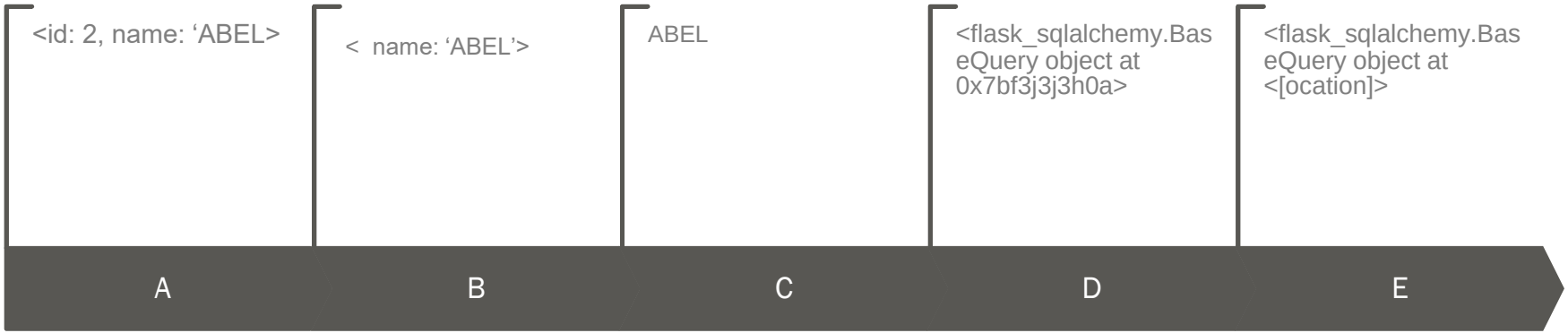
ID: 1, NAME: TOLU
ID: 2, NAME: ABEL
ID: 3, NAME: SHOLA
ID: 4, NAME: EJIRO
ID: 5, NAME: AKPOS

WHAT FOLLOWING COMMAND ENABLES GETTING ALL ITEMS FROM people **TABLE**



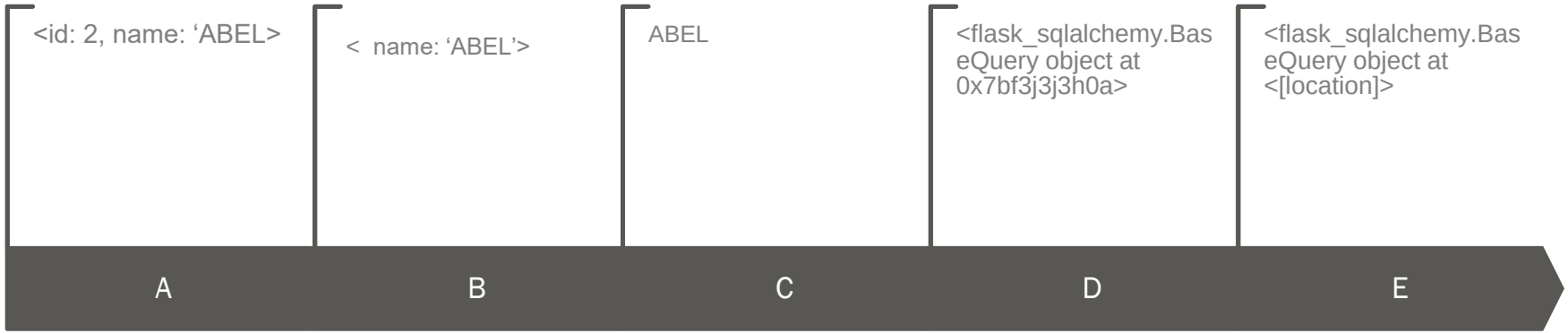
2 WHAT WILL THE FOLLOWING COMMAND RETURNS TO US:

Person.query.filter_by(name='abel').all()



3 WHAT WILL THE FOLLOWING COMMAND RETURNS TO US:

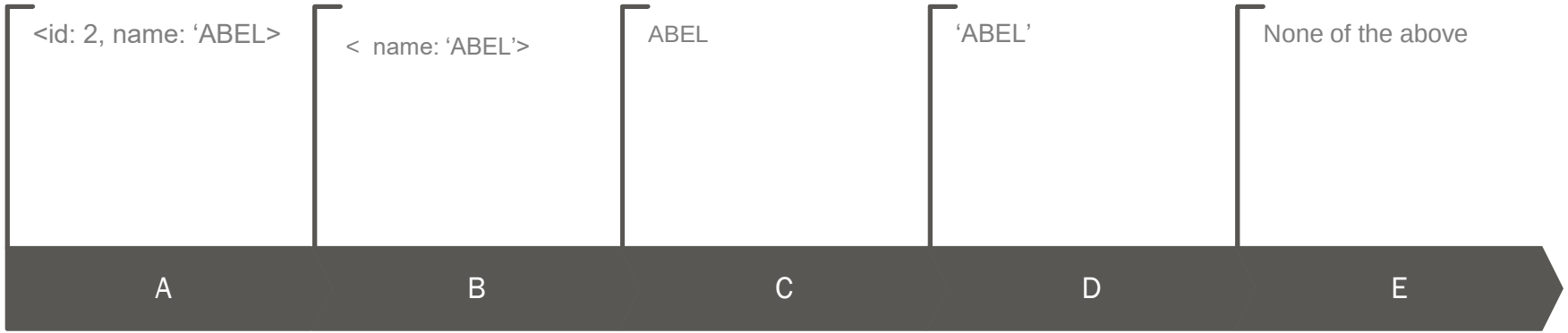
Person.query.filter_by(name='abel')



4 WHAT WILL THE FOLLOWING COMMAND RETURNS TO US:

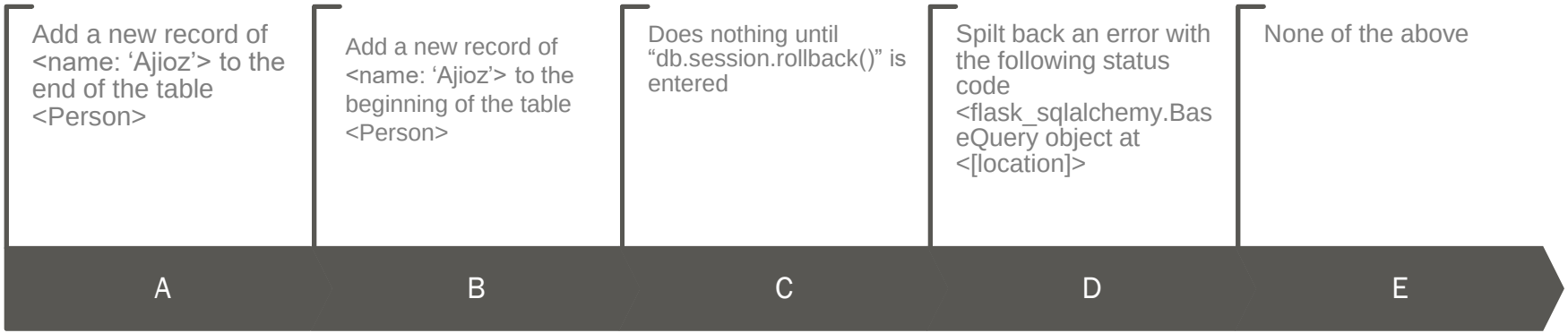
person = Person.query.filter_by(name='abel').first()

person.name

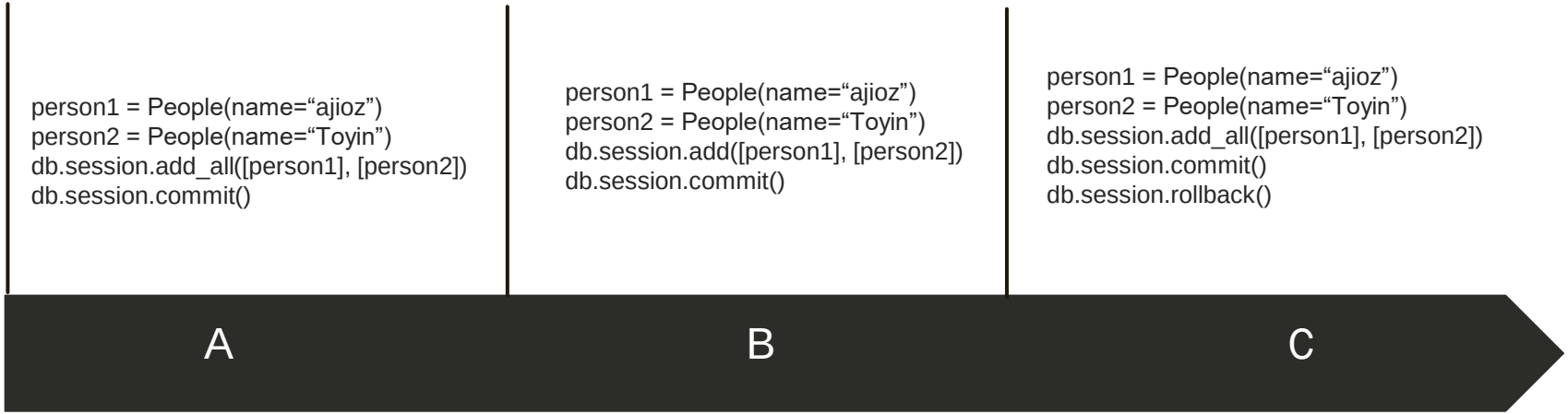


5 WHAT WILL THE FOLLOWING COMMAND DO TO THE EXISTING TABLE <people> **IN THE DATABSE:**

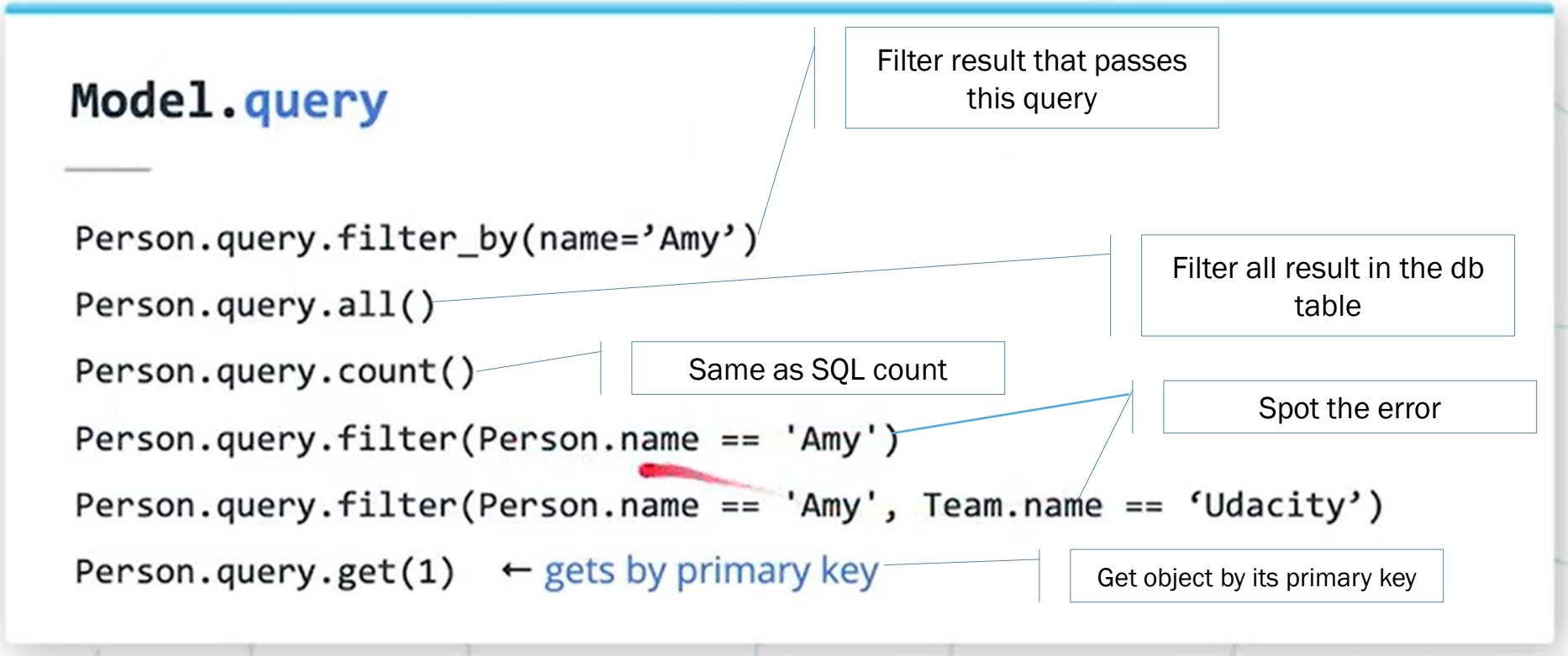
person = People(name="ajioz")
db.session.add(person)
db.session.commit()



6 A DEVELOPER DECIDED TO ADD MULTIPLE RECORDS INTO THE EXISTING TABLE <people> IN A DATABASE, CONSIDER WHICH OF THE FOLLOWING COMMAND IS MOST APPROPRIATE USING SQLALCHEMY



BONUS: MIND REWIND ON SQLALCHEMY ORM



BONUS: MIND REWIND ON SQLALCHEMY ORM

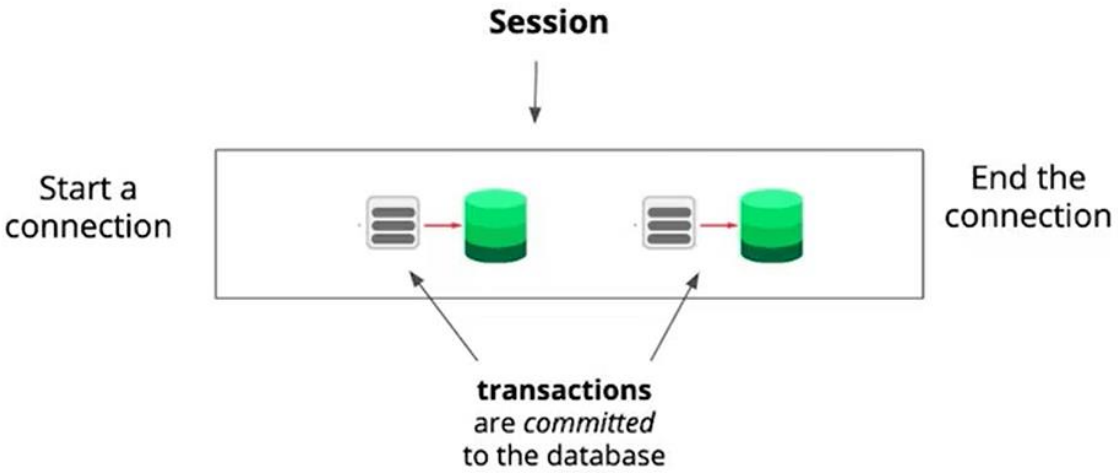
```
Model.query
```

```
Product.query.filter_by(category='Misc').delete()
```

BONUS: MIND REWIND ON SQLALCHEMY ORM OBJECT LIFE-CYCLE

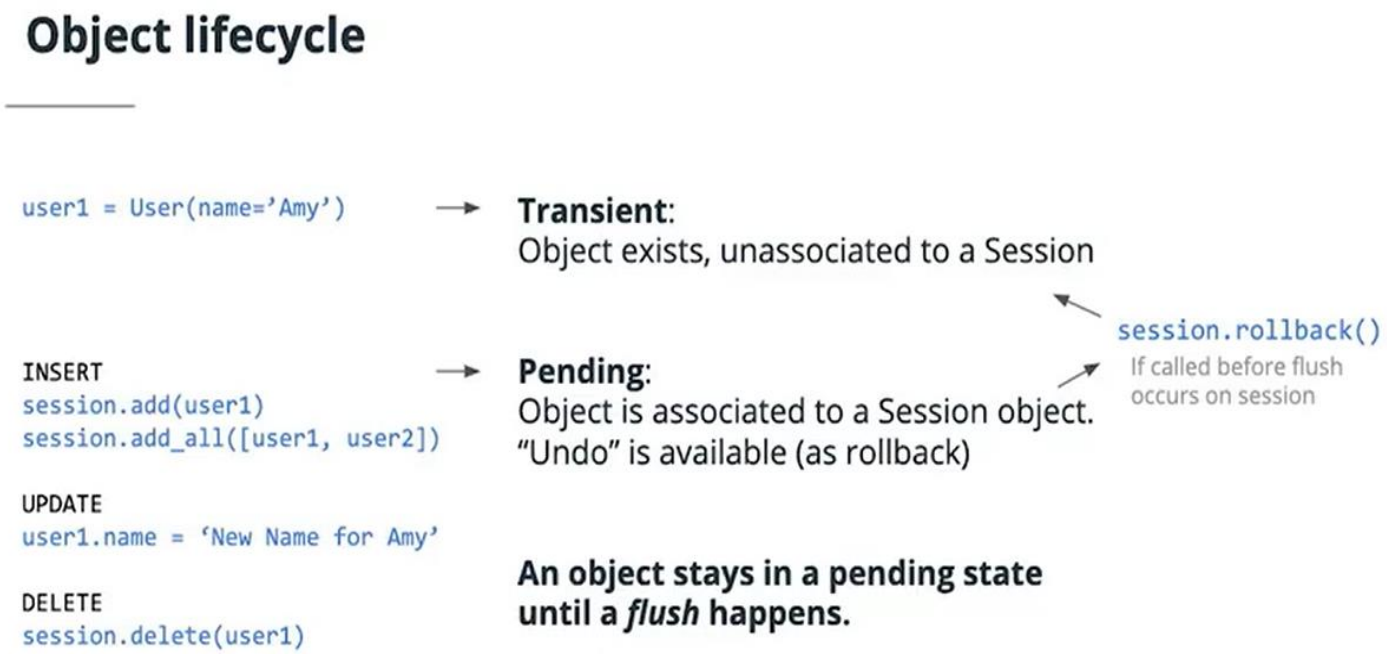
`db.session`

It isn't until we execute db.session.commit() that we commit anything to the database



- Not until we execute the db.session.commit(), we haven't committed or persisted any data to the db
- Every time we want to interact with the database, we start the connection and end the connection when all the interactions are done
- Within that interaction session, we create transaction that we commit into the db
- Proposed database change are not immediately committed to the db, once defined. Changes go through stages in order to provide the ability to "undo" a mistake before committing it to a db

BONUS: MIND REWIND ON SQLALCHEMY ORM OBJECT LIFE-CYCLE



- First is defining an object, in a floating state.
- It isn't we call session.add(user1) Or session.add_all, or session.delete() or we do something to model like the update method that we end up proposing an action to the db. At this point the object is in pending state
- At the pending state it is said to be associated to a session object but we haven't committed it yet, at this stage one can decide to rollback so long flush hasn't yet happen.
- Flush takes pending changes and translate them into SQL commands ready to be committed to db
- Flush occurs under the hood, once you call de.session.commit() to get the data persisted

BONUS: MIND REWIND ON SQLALCHEMY ORM OBJECT LIFE-CYCLE

Calling `Query` flushes pending changes added to `session`

```
# db.session.add(person)
# db.session.add_all([p1, p2])
# db.session.delete(p3)
```

← Adds pending change to a transaction (`db.session` always works within a transaction)

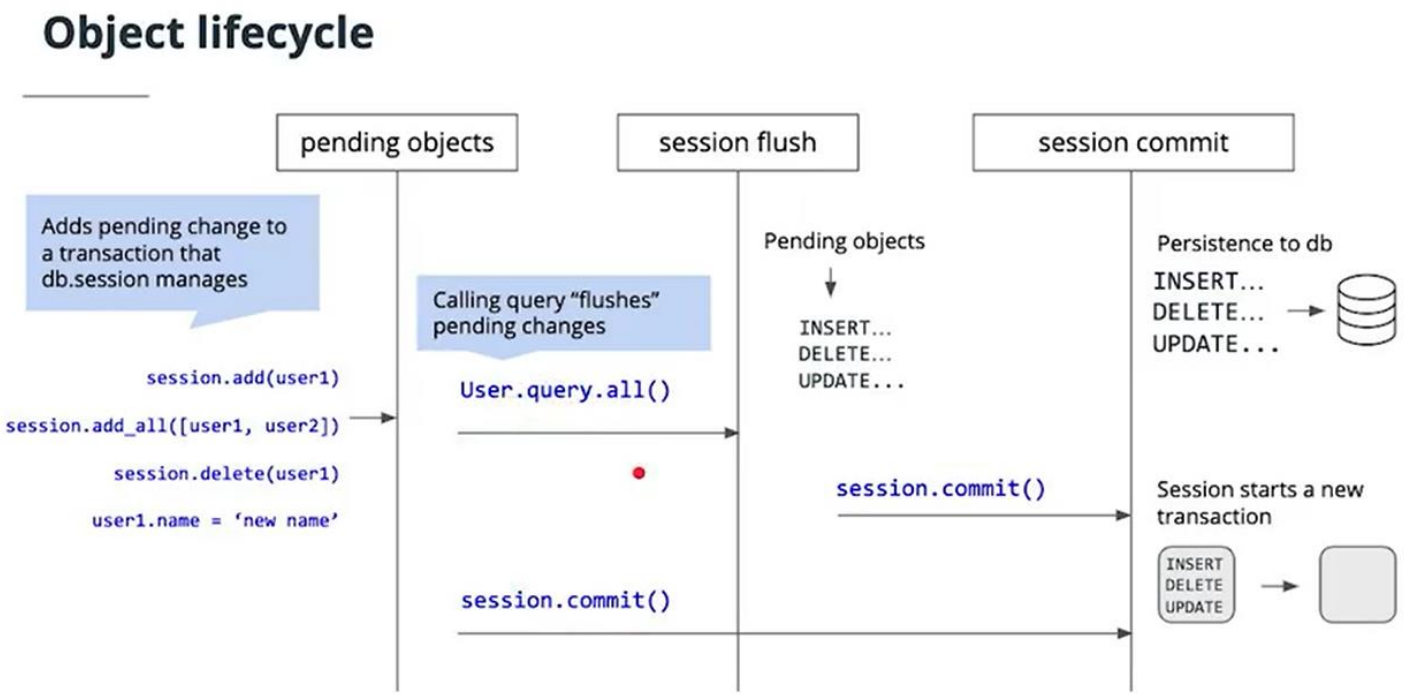
```
# Person.query.first()
```

Calling query "flushes" pending changes

`db.session.commit()` still needs to be called after a flush.

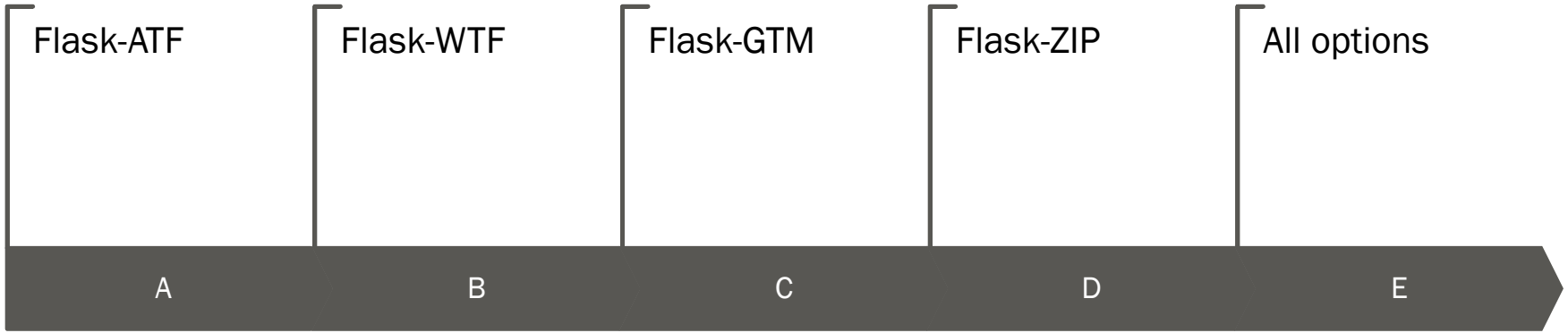
- The one way Flushes occurs

BONUS: MIND REWIND ON SQLALCHEMY ORM OBJECT LIFE-CYCLE

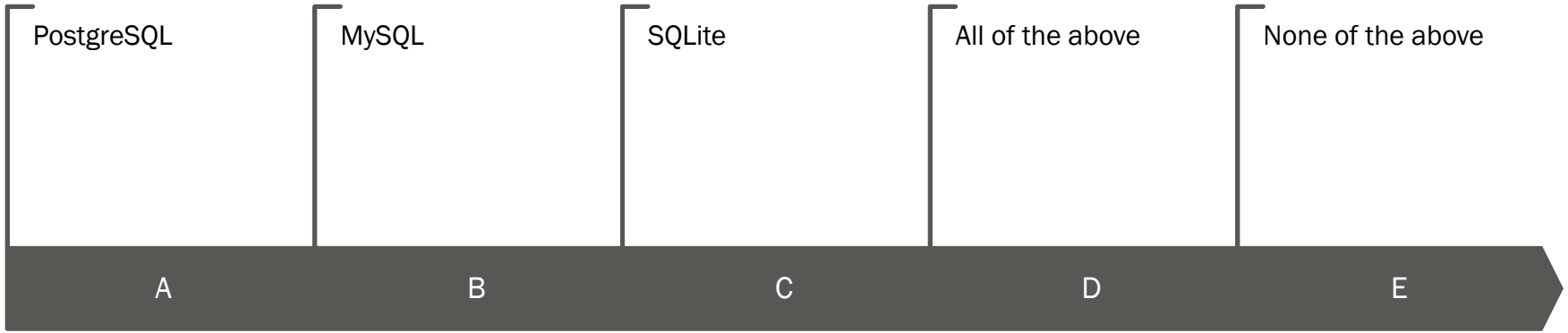


- Another Way Flush can occur again s when we invoke the query command before committing

7 **FORMS IN FLASK CAN BE IMPLEMENTED BY USING AN EXTENSION CALLED ?**



8. FLASK WORKS WITH MOST OF THE RDBMSS, SUCH AS?



9. THE _____ DECORATOR IN FLASK IS USED TO BIND URL TO A FUNCTION.

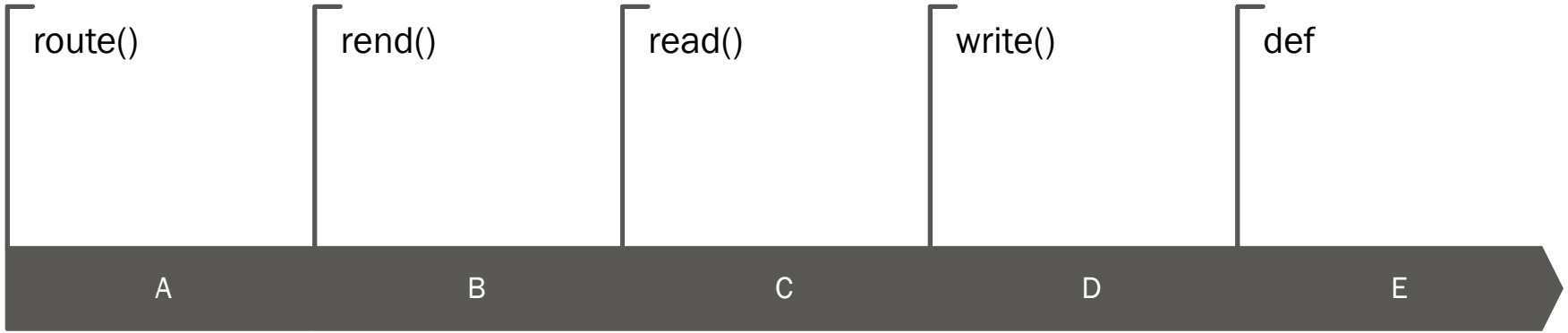


Table of contents

01

SQLAlchemy In Depth- by student

02

Migrations

Why Migrations

03

Building our hello app with migrations

04

Our next step forward

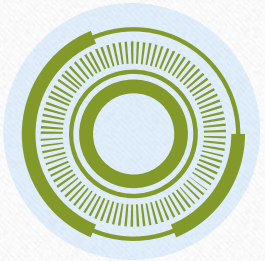
FLASK APP

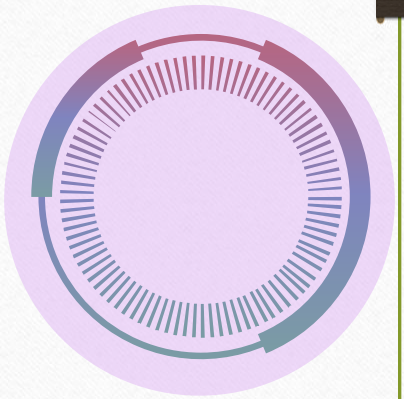
02

MIGRATIONS

What to expect

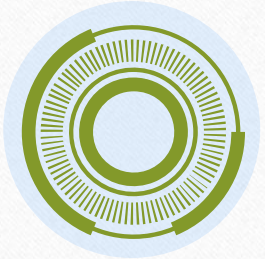
- What are migrations
- Why we use migrations
- How to install necessary libraries
- Steps to get migrations going
- Upgrades and Downgrades

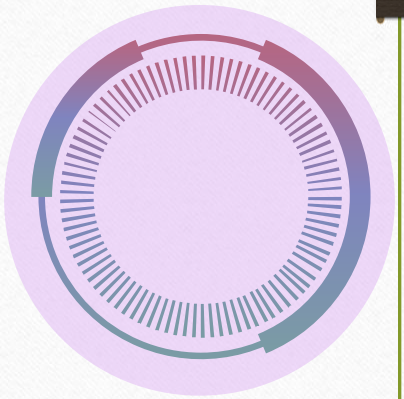




Takeaways

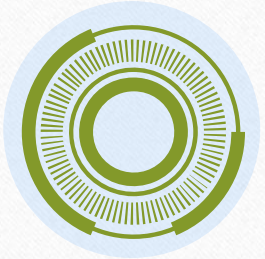
- Migrations deal with how we manage modifications to our data schema, over time.
- Mistakes to our database schema are very expensive to make. The entire app can go down, so we want to
  - quickly roll back changes, and
  - test changes before we make them
- A Migration is a file that keeps track of changes to our database schema (structure of our database).
  - Offers version control on our schema.

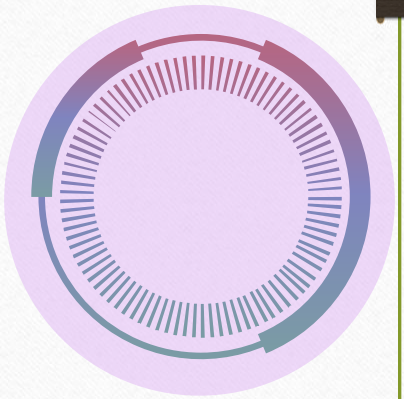




Takeaways (Upgrades and rollbacks)

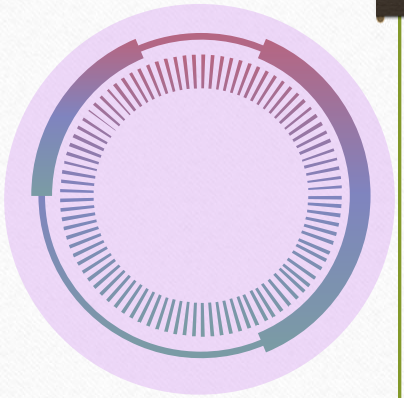
- Migrations stack together in order to form the latest version of our database schema
- We can upgrade our database schema by applying migrations
- We can roll back our database schema to a former version by reverting migrations that we applied



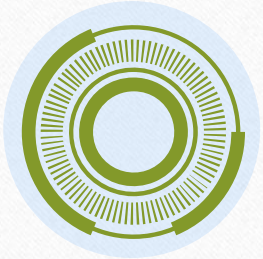


Why are migrations necessary

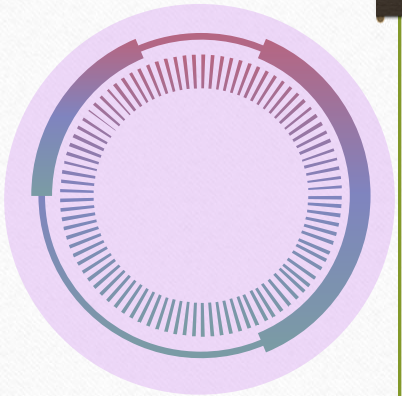
QA SESSION



Migrations (Command-line scripts)



- **migrate**: creating a migration script template to fill out; generating a migration file based on changes to be made
- **upgrade**: applying migrations that hadn't been applied yet ("upgrading" our database)
- **downgrade**: rolling back applied migrations that were problematic ("downgrading" our database)

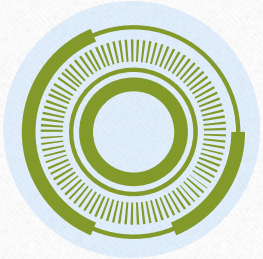


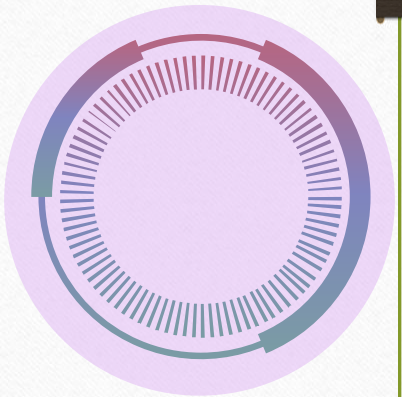
REQUIREMENTS

Flask-Migrate is our library for migrating changes using SQLAlchemy. It uses a library called Alembic underneath the hood

Flask-Migrate (flask_migrate) is our migration manager for migrating SQLALchemy-based database changes

The second library that we're using is called Flask-Script Flask-Script (flask_script) lets us run migration scripts we defined, from the terminal.

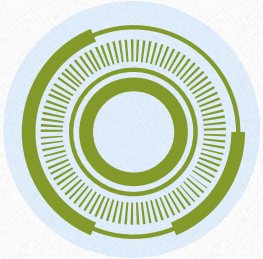


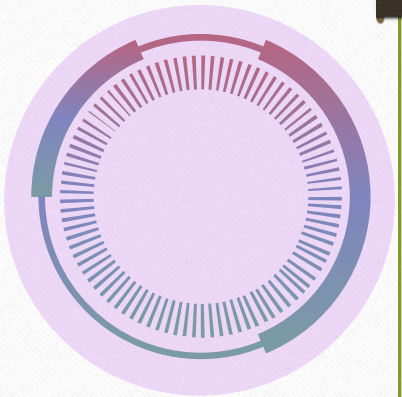


REQUIREMENTS

You can install Flask-Migrate (flask_migrate) and Flask-Script (flask_script) using pip3 or pip if you haven't

To install Flask-Migrate run pip3 install Flask-Migrate





STEPS

1. Initialize the migration repository structure for storing migrations

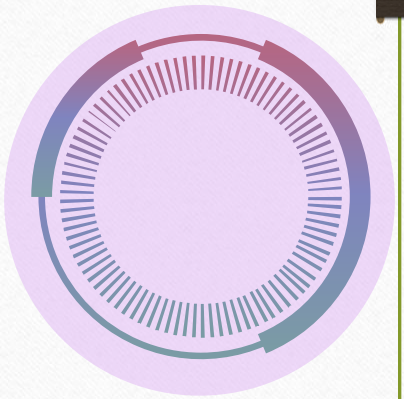
   Command: (flask db init)

2. Create a migration script (using Flask-Migrate

   Command: (flask db migrate)

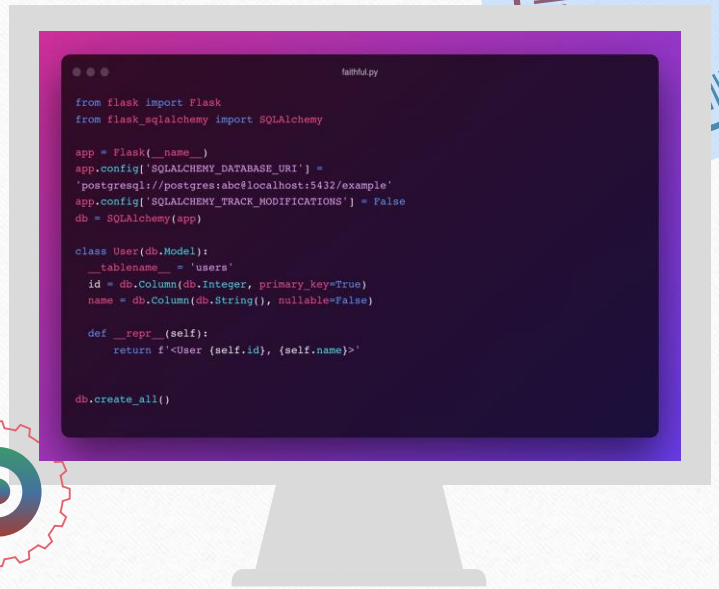
3. (Manually) Run the migration script (using Flask-Script)

   Command :(flask db upgrade) or :(flask db downgrade)



Live Coding Session 3

We will be having our third live coding session for Migrations



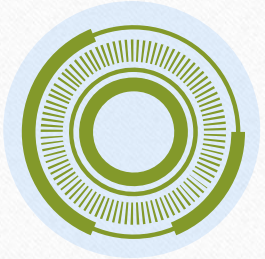
Migration Practice App

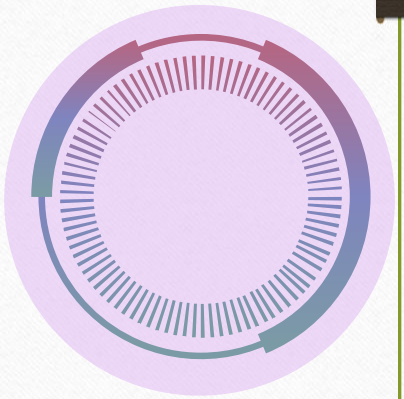
faithful_migration.py

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate


app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

migrate = Migrate(app, db)

class Person(db.Model):
  __tablename__ = 'user'
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(), nullable=False)

  def __repr__(self):
    return f'<Person ID: {self.id}, name: {self.name}>'

@app.route('/')
def hello():
  person = Person.query.first()
  return f'Hello my name is  {person.name}'
```

WHAT ARE MIGRATIONS

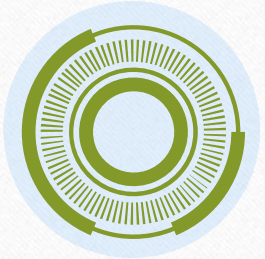
Migrations refer to the management of incremental, reversible changes and version control to relational database schemas.

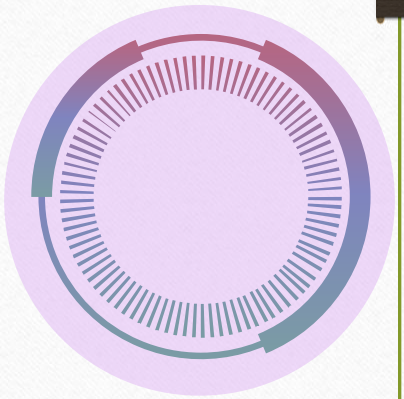




WHY USE MIGRATIONS

- Migrations allow us to keep track of schema changes like how git tracks code changes
- We Can do a granular application of the schema change





03 FLASK APP

Demo the usage of Flask-Migrate

REQUIREMENTS

- Install flask-migrate

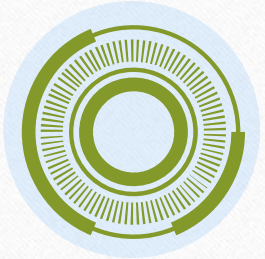
  pip install Flask-Migrate

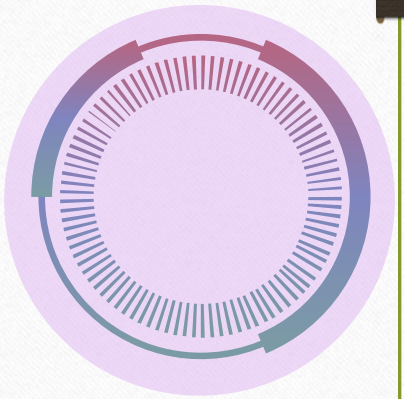
- Initialize flask-migrate

  Using: flask db init

- Update our app to use Flask-Migrate

  Import the migrate library and create a migrate instance





REQUIREMENTS

- Sync your models

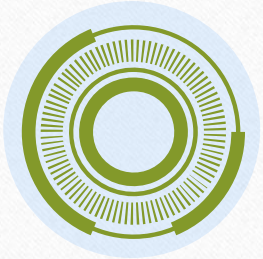
  Using: flask db migrate

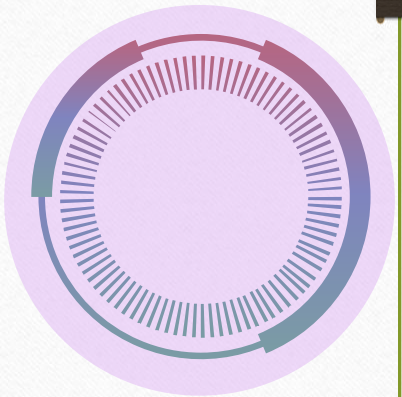
- Upgrade

  Using: flask db upgrade

- Downgrades

  Using: flask db downgrade

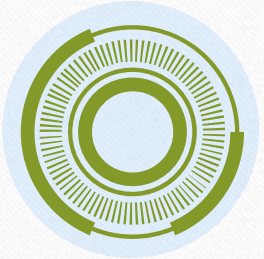


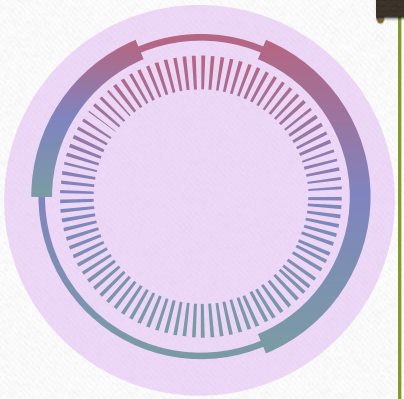


04 What Next

What Next

- Build a CRUD app with SQLAlchemy ORM - Part 1
- Build a CRUD app with SQLAlchemy ORM - Part 2
- Submit Project One: Fyyur





Questions

Feedback

Additional Resources

- Flask Documentation
- Flask Migrate Documentation
- Flask SQLAlchemy Documentation
- A Premier on Database Relationships (Understanding One-to-One, One-to-Many, and Many-to-Many)
- Database Relationships
- SQLAlchemy Cheat Sheet

Additional Information

Using Windows command prompt CMD

- Add PostgreSQL tools to Windows PATH

  Add the PostgreSQL bin directory to the PATH variable

- create database called demo: create -U <super(dbuser)> <dbname>

  createdb -U postgres demo

- Login into database called demo: psql -U <dbuser> -d <dbname>

  psql -U postgres -d demo

Additional Information

Using A Virtual Environment

- Create a virtual environment

  python -m venv env

- Activate a virtual Environment
  - For windows

    env\Scripts\activate

  - For linux and macOs:

    source env/bin/activate

Additional Information

To Run a Flask app in a file called hello.py

- Bash

  export FLASK_APP=hello

  flask run

- PowerShell

  $env:FLASK_APP = "hello"

  flask run

- CMD

  set FLASK_APP=hello

  flask run

Additional Information

To Run interactive python terminal

- Bash

  > type python

  > python -i