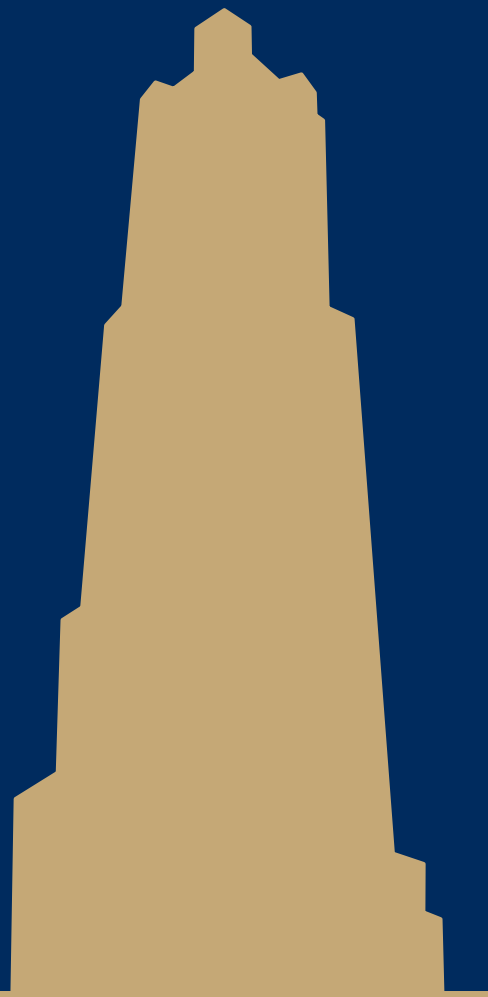
CS/COE 1520

pitt.edu/~ach54/cs1520

Developing Models in Flask



Database overview

- Our models are going to represent the state of a database that contains all of the data used by our web application
- We'll assume the use of transactional object-relational database management systems
  - MySQL, PostgreSQL, Oracle, SQLServer, etc.
- A transaction is a logical unit of work in DBMSs
  - Examples:
    - Transferring money between bank accounts
    - Inventory updates

ACID

- **A**tomicity
  - Either all the operations associated with a transaction happen or none of them happens
- **C**onsistency Preservation
  - A transaction is a correct program segment. It satisfies the database's integrity constraints at its boundaries
- **I**solation
  - Transactions are independent, the result of the execution of concurrent transactions is the same as if transactions were executed serially, one after the other
- **D**urability (a.k.a. Permanency)
  - The effects of completed transactions become permanent surviving any subsequent failure(s)

Data tables

- In relational DBMSs, data is stored in *tables*
- The table rows are the *records* stored in the database, while the columns are the attributes of each record
  - Based on the mathematical concept of a relation (a set of tuples)

Students			
Name	ID	Major	GPA
Alice	334322	CS	3.45
Bob	546346	Math	3.23
Charlie	045628	CS	2.75
Denise	964389	Art	4.0

Relationships between tables

- Each table should have a *primary key*
  - Attribute that uniquely identifies each row
- *Foreign keys* are attributes that refer to rows in other tables
- Cardinality ratios of relationships between tables must be carefully considered
  - 1:1
    - A person has a driver's license
  - 1:n
    - A movie has a director, but a director will make many movies
  - n:m
    - A student enrolls in many classes and each class will have many students

SQL

- Structured Query Language
- De facto query language for object-relational database management systems
- It is a *declarative* language
  - State what you want, not how to get it.
  - E.g.,

```
SELECT *
FROM Students
WHERE GPA > 3.5;
```

SQL crash course

- Not happening

Object-relational mapping (ORM)

- Will map relational data (records from database tables) to objects that we can directly use within Python
  - Glean all of the benefits provided by the data, all while writing only Pythonic code!

SQLAlchemy

- Database abstraction toolkit and ORM
- We will use an extension to Flask that allows us to use SQLAlchemy's ORM within our Flask applications
  - This is the "micro" part of Flask being a "microframework", no ORM by default

Using SQLAlchemy within Flask

- An extension, so must be imported on its own:
  - `from flask_sqlalchemy import SQLAlchemy`
- Must be tied to the flask app at initialization

Example model

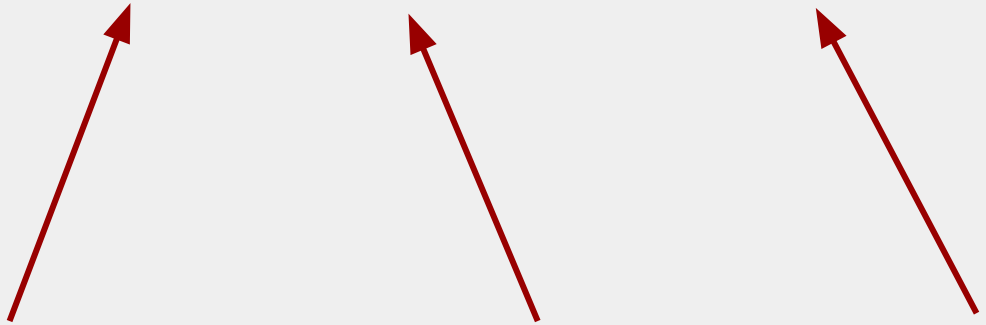
```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)
    def __repr__(self):
        return "<User {}>".format(repr(self.username))
```

- And performed the following:

```
a = User(username="admin", email="admin@example.com")
db.session.add(a)
p = User(username="peter", email="peter@example.org")
db.session.add(p)
g = User(username="guest", email="guest@example.com")
db.session.add(g)
db.session.commit()
```

Querying models

- Answering questions about data stored in the database
- Using SQLAlchemy, we will express such questions by chaining together calls to functions that produce SQLAlchemy Query objects
  - `entries = Entry.query.order_by(Entry.id).all()`



Consider the following queries

- `User.query.filter_by(username='peter').first()`
- `User.query.filter_by(username='missing').first()`
- `User.query.filter(User.email.endswith('@example.com')).all()`
- `User.query.order_by(User.username)`
- `User.query.order_by(User.username).all()`
- `User.query.all()`
- `User.query.limit(1).all()`
- `User.query.get(1)`

Relationships

- SQLAlchemy provides constructs for easily accessing related models
  - Through defining attributes using `db.relationship()`

One-to-Many relationship

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    addresses = db.relationship('Address',
                                backref='person',
                                lazy='dynamic')


class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(50))
    person_id = db.Column(db.Integer,
                          db.ForeignKey('person.id'))
```

lazy-ness

- `select`
  - The default
  - SQLAlchemy will load the data as necessary in one go using a standard select statement
- `joined`
  - SQLAlchemy will load the relationship in the same query as the parent using a JOIN statement.
- `subquery`
  - Works like `joined` but instead SQLAlchemy will use a subquery
- `dynamic`
  - Instead of loading the items SQLAlchemy will return another query object which you can further refine before loading the items

Many-to-Many relationship

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id')),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'))
)


class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags,
                    lazy='select',
                    backref=db.backref('pages', lazy='select'))


class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```