MAKE SCHOOL

More Models + Review

BEW 1.2

Agenda

MAKE SCHOOL

- Learning Outcomes
- Warm-Up: Code Review
- One-to-Many Relationships
- Many-to-Many Relationships
- Querying the Database
- **BREAK**
- SQLite DB Browser tool
- Lab Time

Warm-Up: Models Lab (8 minutes)

MAKE SCHOOL

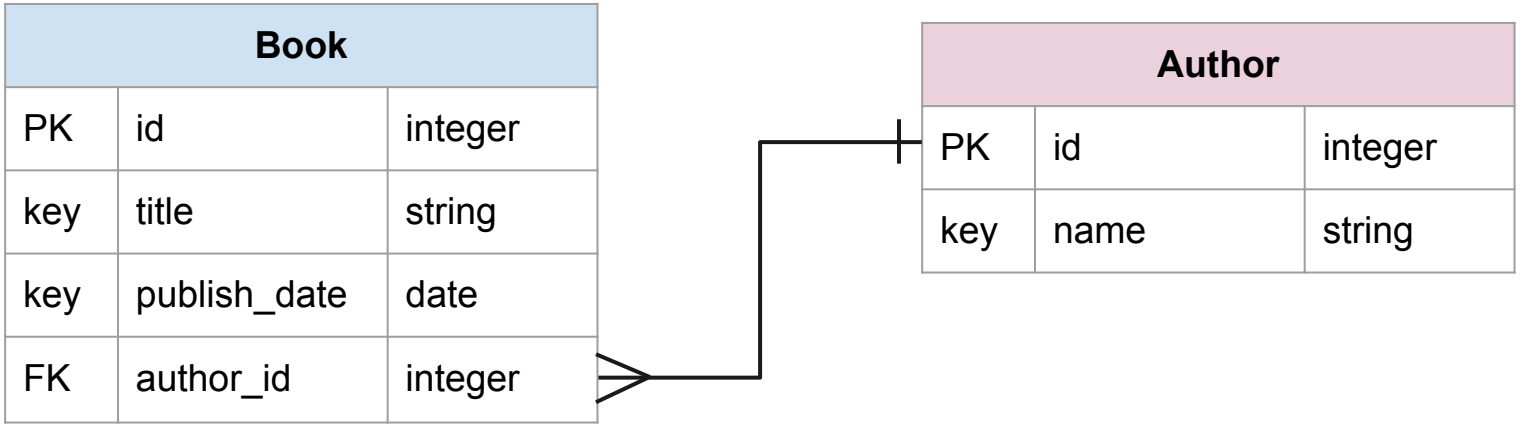
In a group of 3, answer the following questions about the Models (Books) lab:

1. How far did you get?
2. Which parts were confusing?
3. What questions do you still have?

One-to-Many Relationships

One-to-Many Relationships

Let's say our site has **Book** and **Author** entities with the following relationship:



Book		
PK	id	integer
key	title	string
key	publish_date	date
FK	author_id	integer

Author		
PK	id	integer
key	name	string

One-to-Many Relationships

MAKE SCHOOL

We can use a **db.ForeignKey** field to store the author id:

```
class Author(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)
    publish_date = db.Column(db.Date)
    author_id = db.Column(db.Integer, db.ForeignKey('author.id'))
```

One-to-Many Relationships

MAKE SCHOOL

However, it would be pretty annoying to search for a book, and then only be able to see its author's id.

Ideally, when we query for a book, **we want to get the author's info** without having to do an extra query.

Luckily, SQLAlchemy has us covered! :)

One-to-Many Relationships

MAKE SCHOOL

We can use **relationship** fields to tell SQLAlchemy to automatically populate some extra data when it does a query. These aren't real columns in the SQL table, unlike the **Column** fields.

```
class Author(db.Model):
    # ...
    books = db.relationship('Book', back_populates='author')

class Book(db.Model):
    # ...
    author_id = db.Column(db.Integer, db.ForeignKey('author.id'))
    author = db.relationship('Author', back_populates='books')
```

One-to-Many Relationships

MAKE SCHOOL

When two models both have relationships to each other, we can use **back_populates** to denote a two-way relationship.

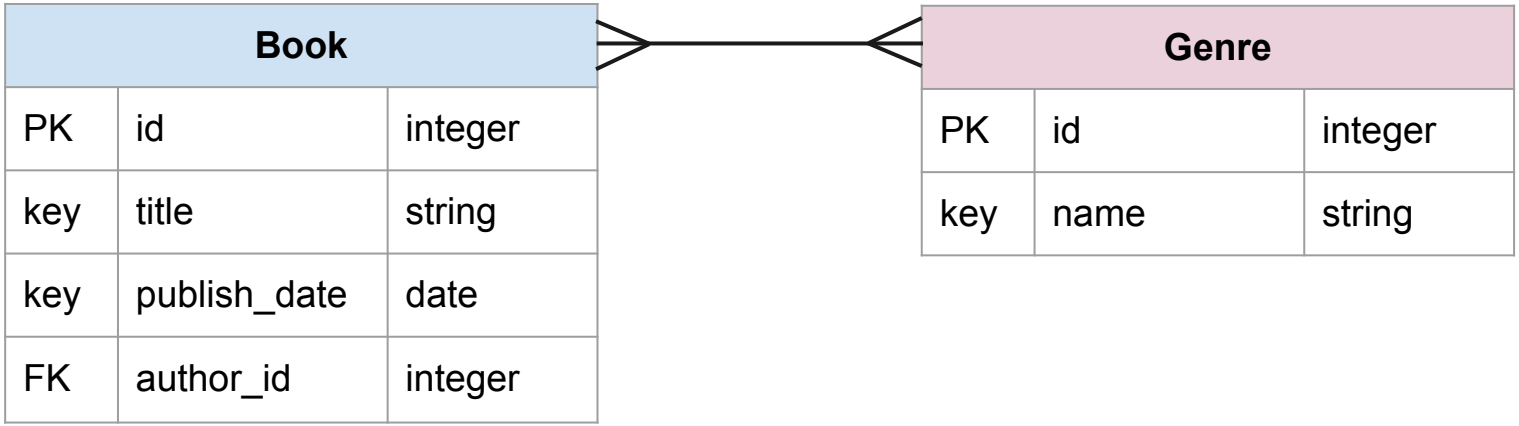
```
class Author(db.Model):
    # ...
    books = db.relationship('Book', back_populates='author')

class Book(db.Model):
    # ...
    author_id = db.Column(db.Integer, db.ForeignKey('author.id'))
    author = db.relationship('Author', back_populates='books')
```

Many-to-Many Relationships

Many-to-Many Relationships

Now let's say we have the following relationship:



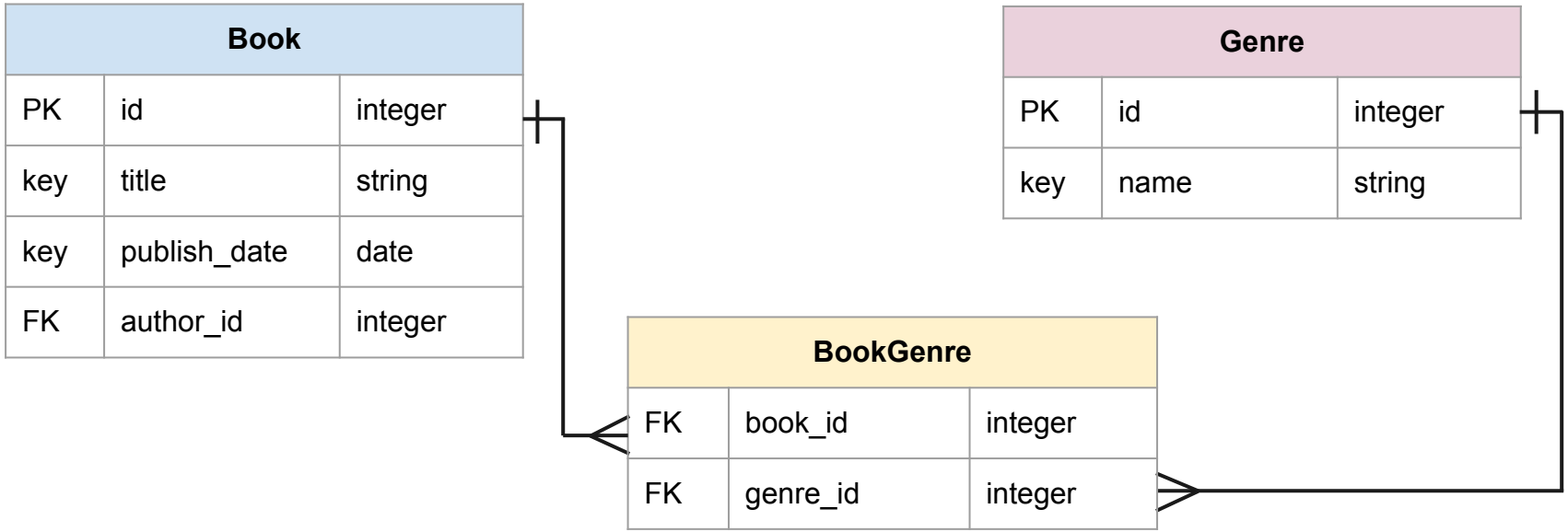
Book		
PK	id	integer
key	title	string
key	publish_date	date
FK	author_id	integer

Genre		
PK	id	integer
key	name	string

Notice that neither **Book** nor **Genre** have a foreign key field that explicitly refers to the other.

Many-to-Many Relationships

Remember, **we can re-write this relationship** by using a **bridge table**:



Notice that there isn't any important information in the bridge table. **It's just a tool** to get us that many-to-many relationship.

Many-to-Many Relationships

MAKE SCHOOL

It's recommended to make the bridge table a **db.Table**, not a **db.Model**, because we don't need to query it directly.

```
class Genre(db.Model):
    # ...
    books = db.relationship('Book', secondary='book_genre', back_populates='genres')

class Book(db.Model):
    # ...
    genres = db.relationship('Genre', secondary='book_genre', back_populates='books')

book_genre_table = db.Table('book_genre',
    db.Column('book_id', db.Integer, db.ForeignKey('book.id')),
    db.Column('genre_id', db.Integer, db.ForeignKey('genre.id'))
)
```

MAKE SCHOOL

Break - 10 min

MAKE SCHOOL

Querying the Database

Querying Objects

MAKE SCHOOL

We can query objects with .get() or .filter_by():

```
# Get the user with id=5
my_user = User.query.get(5)
```

```
# Get the user with username "me"
my_user = User.query.filter_by(username="me").one()
```

Querying Many Objects

MAKE SCHOOL

We can get a list of all objects matching a filter with .all() or .filter_by():

```
# Get all users
my_user = User.query.all()
```

```
# Get all users with type "student"
my_user = User.query.filter_by(type="student").all()
```

Creating Objects

We can create an object by creating an instance of the model class:

```
# Create a user with username="ilikecoding"
# and password "1234"
my_user = User(username="ilikecoding", password="1234")
```

```
# Save the user to the database
db.session.add(my_user)
db.session.commit()
```

Updating Objects

MAKE SCHOOL

We can update an object by updating its model instance:

```
# Update the user to have birth date of 1/15/2000
my_user.birth_date = datetime.Date(2000, 1, 15)
```

```
# Save the user to the database
db.session.add(my_user)
db.session.commit()
```

Querying Objects

MAKE SCHOOL

Let's say we have a model **BlogPost** with fields **id**, **title**, **description**, & **author_id**.

Write code to get the **BlogPost** with **id=7** and print its title.

Students, write your response!

Creating Objects

MAKE SCHOOL

Let's say we have a model **BlogPost** with fields **id**, **title**, **description**, & **author_id**. How do we create an instance of it & save to the database?

Write example code to create & save a **BlogPost** with **author_id=1**.

Students, write your response!

Updating Objects

MAKE SCHOOL

Let's say we have a model **BlogPost** with fields **id**, **title**, **description**, & **author_id**.

Write code to update the **BlogPost** object with **id=7** to have the **title** of **"Hello, World"**.

Students, write your response!

MAKE SCHOOL

SQLite Browser

SQLite Browser

MAKE SCHOOL

Download this DB Browser for SQLite.

Watch as your instructor demonstrates how it works to open & inspect your SQLite data.

Then, breakout into groups of 3. Make sure everyone in your group is able to use the tool to view, add, modify, & delete data.

Python Modules & Packages

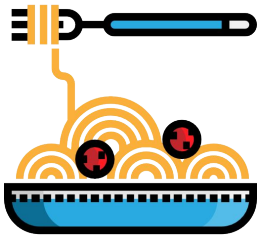
What is Modular Programming?

MAKE SCHOOL

Modular programming is a design technique to break your code into separate parts, with minimal dependencies on each other.

You will see modular programming used in lots of different contexts. It minimizes bugs and makes it easier to find the source of errors.

On the opposite extreme of modular programming is "spaghetti code", where many parts all reference each other and it becomes more difficult to locate errors.



Modules vs. Packages

MAKE SCHOOL

A module is a single file (or files) that is imported under a single import statement.

```
from my_module import my_function
```

A package is a collection of modules in directories that give a file hierarchy.

```
from my_package.foo.bar import my_function
```

We use packages in our projects to make it easier to organize the code.

`__init__.py`

MAKE SCHOOL

`__init__.py` is a special file that declares a Python package. It used to be required to declare a package, but as of Python 3.3 it is no longer required.

However, it is still useful! For example, the **`unittest`** library will not search inside your folders unless they include `__init__.py`.

We can also declare variables & run setup code inside of `__init__.py`.

__init__.py

For example, we can use `__init__.py` to declare the `app` & `db` variables (source):

```
# __init__.py
# (imports omitted for brevity)
app = Flask(__name__)
app.config.from_object(Config)

db = SQLAlchemy(app)

from books_app.routes import main

app.register_blueprint(main)

with app.app_context():
    db.create_all()
```

Circular Imports

MAKE SCHOOL

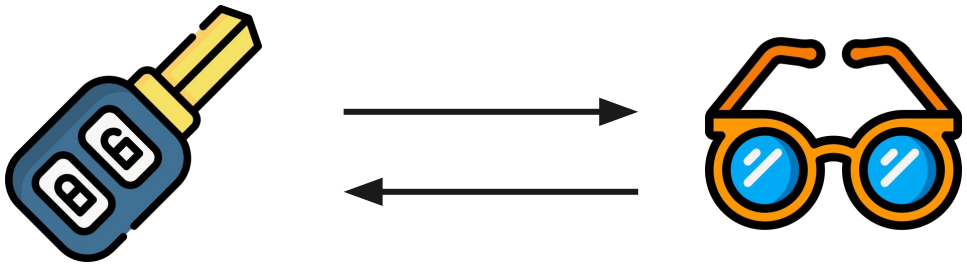
Ok, so... Why can't we put `from books_app.routes import main` at the top of the file?

That's because it would cause a **circular import**.

Circular Imports

A **circular import** occurs when two files are both importing each other, and both depend on each other to be initialized.

Think of it like a catch-22: *"I can't find my car keys without my glasses, but I left my glasses in the car so I can't get them without my car keys."*



Circular Imports

```python
# books_app/__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from books_app.config import Config
import os
from books_app.routes import main

app = Flask(__name__)
app.config.from_object(Config)

db = SQLAlchemy(app)


app.register_blueprint(main)

with app.app_context():
   db.create_all()
```

```python
# books_app/routes.py
from flask import Blueprint
from books_app.models import Book,
Author, Genre


from books_app import app, db


main = Blueprint("main", __name__)
# ... routes ...
```

app, db not yet initialized

Circular Imports

```python
# books_app/__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from books_app.config import Config
import os

app = Flask(__name__)
app.config.from_object(Config)

db = SQLAlchemy(app)

from books_app.routes import main

app.register_blueprint(main)

with app.app_context():
    db.create_all()
```

```python
# books_app/routes.py
from flask import Blueprint
from books_app.models import Book,
Author, Genre

from books_app import app, db

main = Blueprint("main", __name__)
# ... routes ...
```

app, db are initialized

MAKE SCHOOL

Blueprints

Flask Blueprints

MAKE SCHOOL

Flask Blueprints are another way to use modular programming to organize our code. They allow us to separate out Flask routes into multiple files.

So far, our applications have been pretty small. But for a larger website, you could have hundreds of routes. It makes sense to organize them into categories based on their purpose.

For example, we may have `main` & `auth` blueprints to separate out authentication routes. If your site has an API, you may want to have an `api` blueprint.

Flask Blueprints

MAKE SCHOOL

We can **declare** a blueprint in the routes file:

```
main = Blueprint("main", __name__)

@main.route('/')
def homepage():
    # ...
```

Then we need to register the blueprint in the `__init__.py` file:

```
app.register_blueprint(main)
```

Flask Blueprints

MAKE SCHOOL

Two blueprints can each have a route with the same name, and there won't be a naming conflict:

```
main = Blueprint("main", __name__)

@main.route('/')
def index():
    # ...
```

```
api = Blueprint("api", __name__)

@api.route('/api')
def index():
    # ...
```

```
url_for('main.index')
  -> '/'
```

```
url_for('api.index')
  -> '/api'
```

MAKE SCHOOL

Lab Time

MAKE SCHOOL

Wrap-Up

Wrap-Up

MAKE SCHOOL

Homework 2: Events Site

- Due by next Tuesday at 11:59pm

Quiz 1: Models