Data Traffic Management

au820421106031 | M.Mukilan | Phase-2

What is data traffic management?

Internet traffic management, also known as application traffic management, refers to tInnovative Uses for Big Data in Traffic Management.

In python programming data traffic management is underways controlled by these varies file texting techniques:

- Flat files for data storage
- SQL to improve access to persistent data
- SQLite for data storage
- SQLAlchemy to work with data as Python objects

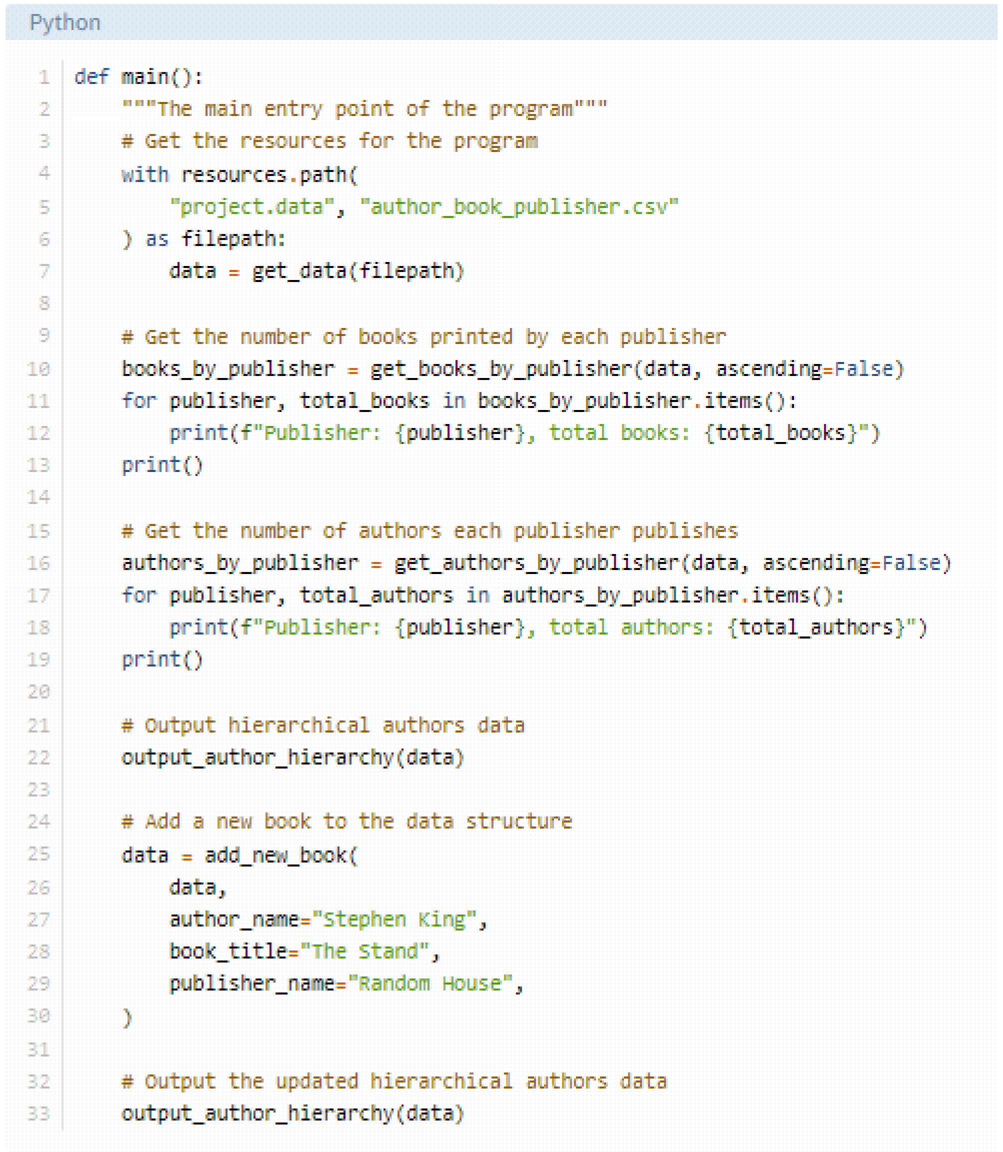
Using Flat Files for Data Storage

A flat file is a file containing data with no internal hierarchy and usually no references to external files. Flat files contain human-readable characters and are very useful for creating and reading data. Because they don't have to use fixed field widths, flat files often use other structures to make it possible for a program to parse text.

For example, comma-separated value (CSV) files are lines of plain text in which the comma character separates the data elements. Each line of text represents a row of data, and each comma-separated value is a field within that row. The comma character delimiter indicates the boundary between data values.

Python excels at reading from and saving to files. Being able to read data files with Python allows you to restore an application to a useful state when you rerun it at a later time. Being able to save data in a file allows you to share information from the program between users and sites where the application runs.

Before a program can read a data file, it has to be able to understand the data. Usually, this means the data file needs to have some structure that the application can use to read and parse the text in the file.



```python
def main():
    """The main entry point of the program"""
    # Get the resources for the program
    with resources.path(
        "project.data", "author_book_publisher.csv"
    ) as filepath:
        data = get_data(filepath)

    # Get the number of books printed by each publisher
    books_by_publisher = get_books_by_publisher(data, ascending=False)
    for publisher, total_books in books_by_publisher.items():
        print(f"Publisher: {publisher}, total books: {total_books}")
    print()

    # Get the number of authors each publisher publishes
    authors_by_publisher = get_authors_by_publisher(data, ascending=False)
    for publisher, total_authors in authors_by_publisher.items():
        print(f"Publisher: {publisher}, total authors: {total_authors}")
    print()

    # Output hierarchical authors data
    output_author_hierarchy(data)

    # Add a new book to the data structure
    data = add_new_book(
        data,
        author_name="Stephen King",
        book_title="The Stand",
        publisher_name="Random House",
    )

    # Output the updated hierarchical authors data
    output_author_hierarchy(data)
```

Advantages of Flat Files

Working with data in flat files is manageable and straightforward to implement. Having the data in a human-readable format is helpful not only for creating the data file with a text editor but also for examining the data and looking for any inconsistencies or problems.

Many applications can export flat-file versions of the data generated by the file. For example, Excel can import or export a CSV file to and from a spreadsheet. Flat files also have the advantage of being self-contained and transferable if you want to share the data.

Almost every programming language has tools and libraries that make working with CSV files easier. Python has the built-in csv module and the powerful pandas module available, making working with CSV files a potent solution.

Disadvantages of Flat Files

The advantages of working with flat files start to diminish as the data becomes larger. Large files are still human-readable, but editing them to create data or look for problems becomes a more difficult task. If your application will change the data in the file, then one solution would be to read the entire file into memory, make the changes, and write the data out to another file.

Another problem with using flat files is that you'll need to explicitly create and maintain any relationships between parts of your data and the application program within the file syntax. Additionally, you'll need to generate code in your application to use those relationships.

A final complication is that people you want to share your data file with will also need to know about and act on the structures and relationships you've created in the data. To access the information, those users will need to understand not only the structure of the data but also the programming tools necessary for accessing it.

Using SQLite to Persist Data

As you saw earlier, there's redundant data in the author_book_publisher.csv file. For example, all information about Pearl Buck's The Good Earth is listed twice because two different publishers have published the book.

Imagine if this data file contained more related data, like the author's address and phone number, publication dates and ISBNs for books, or addresses, phone numbers, and perhaps yearly revenue for publishers. This data would be duplicated for each root data item, like author, book, or publisher.

Shell

```
$ python main.py
Publisher: Simon & Schuster, total books: 4
Publisher: Random House, total books: 4
Publisher: Penguin Random House, total books: 2
Publisher: Berkley, total books: 2

Publisher: Simon & Schuster, total authors: 4
Publisher: Random House, total authors: 3
Publisher: Berkley, total authors: 2
Publisher: Penguin Random House, total authors: 1

Authors
├── Alex Michaelides
│   └── The Silent Patient
│       └── Simon & Schuster
├── Carol Shaben
│   └── Into The Abyss
│       └── Simon & Schuster
├── Isaac Asimov
│   └── Foundation
│       └── Random House
├── John Le Carre
│   └── Tinker, Tailor, Soldier, Spy: A George Smiley Novel
│       └── Berkley
├── Pearl Buck
│   └── The Good Earth
│       ├── Random House
│       └── Simon & Schuster
├── Stephen King
│   ├── Dead Zone
│   │   └── Random House
│   ├── It
│   │   ├── Penguin Random House
│   │   └── Random House
│   └── The Shining
│       └── Penguin Random House
└── Tom Clancy
    ├── Patriot Games
    │   └── Simon & Schuster
    └── The Hunt For Red October
        └── Berkley
```

It’s possible to create data this way, but it would be exceptionally unwieldy.

Think about the problems keeping this data file current. What if Stephen King wanted to change his name? You'd have to update multiple records containing his name and make sure there were no typos.

Worse than the data duplication would be the complexity of adding other relationships to the data. What if you decided to add phone numbers for the authors, and they had phone numbers for home, work, mobile, and perhaps more? Every new relationship that you'd want to add for any root item would multiply the number of records by the number of items in that new relationship.

This problem is one reason that relationships exist in database systems. An important topic in database engineering is database normalization, or the process of breaking apart data to reduce redundancy and increase integrity. When a database structure is extended with new types of data, having it normalized beforehand keeps changes to the existing structure to a minimum.

The SQLite database is available in Python, and according to the SQLite home page, it's used more than all other database systems combined. It offers a full-featured relational database management system (RDBMS) that works with a single file to maintain all the database functionality.

It also has the advantage of not requiring a separate database server to function. The database file format is cross-platform and accessible to any programming language that supports SQLite.

Python

```python
def get_data(filepath):
    """Get book data from the csv file"""
    return pd.read_csv(filepath)
```

This function takes in the file path to the CSV file and uses pandas to read it into a pandas DataFrame, which it then passes back to the caller. The return value of this function becomes the data structure passed to the other functions that make up the program.

`get_books_by_publisher()` calculates the number of books published by each publisher. The resulting pandas Series uses the pandas GroupBy functionality to group by publisher and then sort based on the `ascending` flag:

Python

```python
def get_books_by_publisher(data, ascending=True):
    """Return the number of books by each publisher as a pandas series"""
    return data.groupby("publisher").size().sort_values(ascending=ascending)
```

`get_authors_by_publisher()` does essentially the same thing as the previous function, but for authors:

Python

```python
def get_authors_by_publisher(data, ascending=True):
    """Returns the number of authors by each publisher as a pandas series"""
    return (
        data.assign(name=data.first_name.str.cat(data.last_name, sep=" "))
        .groupby("publisher")
        .nunique()
        .loc[:, "name"]
        .sort_values(ascending=ascending)
    )
```

Using Flask With Python, SQLite, and SQLAlchemy

The examples/example_3/chinook_server.py program creates a Flask application that you can interact with using a browser. The application makes use of the following technologies:

- Flask Blueprint is part of Flask and provides a good way to follow the separation of concerns design principle and create distinct modules to contain functionality.

- Flask SQLAlchemy is an extension for Flask that adds support for

SQLAlchemy in your web applications.

- Flask_Bootstrap4 packages the Bootstrap front-end tool kit, integrating it with your Flask web applications.

- Flask_WTF extends Flask with WTForms, giving your web applications a useful way to generate and validate web forms.

- python_dotenv is a Python module that an application uses to read environment variables from a file and keep sensitive information out of program code.

Though not necessary for this example, a .env file holds the environment variables for the application. The .env file exists to contain sensitive information like passwords, which you should keep out of your code files. However, the content of the project .env file is shown below since it doesn't contain any sensitive data:

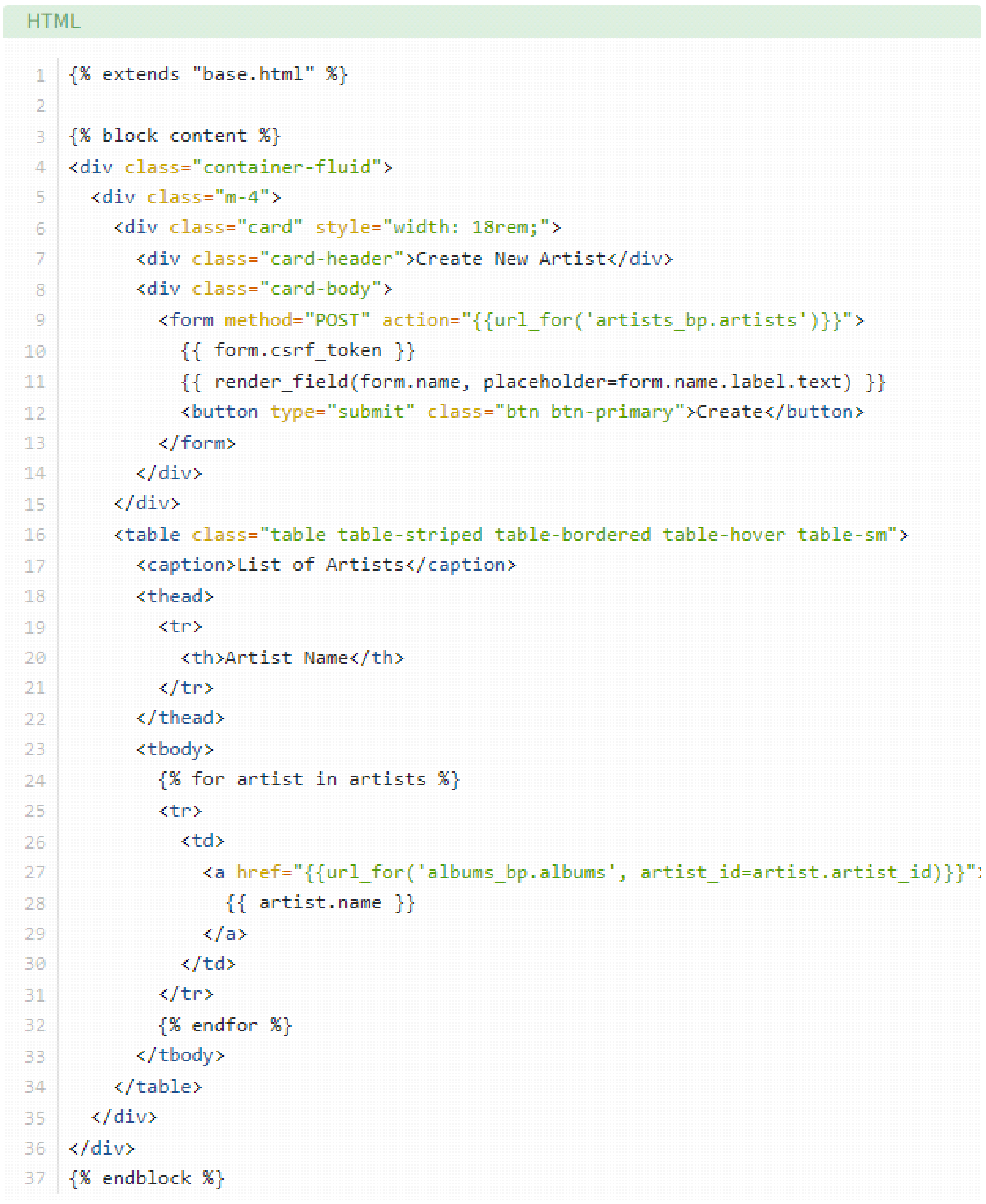
```
SECRET_KEY = "you-will-never-guess"
SQLALCHEMY_TRACK_MODIFICATIONS = False
SQLAlCHEMY_ECHO = False
DEBUG = True
```

The example application is fairly large, and only some of it is relevant to this tutorial. For this reason, examining and learning from the code is left as an exercise for the reader. That said, you can take a look at an animated screen capture of the application below, followed by the HTML that renders the home page and the Python Flask route that provides the dynamic data.

Here's the application in action, navigating through various menus and features:

The chinook database web application in action as an animated GIF

The animated screen capture starts on the application home page, styled using Bootstrap 4. The page displays the artists in the database, sorted in ascending order. The remainder of the screen capture presents the results of clicking on the displayed links or navigating around the application from the top-level menu.



HTML

```
 1  {% extends "base.html" %}
 2
 3  {% block content %}
 4  <div class="container-fluid">
 5    <div class="m-4">
 6      <div class="card" style="width: 18rem;">
 7        <div class="card-header">Create New Artist</div>
 8        <div class="card-body">
 9          <form method="POST" action="{{url_for('artists_bp.artists')}}">
10            {{ form.csrf_token }}
11            {{ render_field(form.name, placeholder=form.name.label.text) }}
12            <button type="submit" class="btn btn-primary">Create</button>
13          </form>
14        </div>
15      </div>
16      <table class="table table-striped table-bordered table-hover table-sm">
17        <caption>List of Artists</caption>
18        <thead>
19          <tr>
20            <th>Artist Name</th>
21          </tr>
22        </thead>
23        <tbody>
24          {% for artist in artists %}
25          <tr>
26            <td>
27              <a href="{{url_for('albums_bp.albums', artist_id=artist.artist_id)}}"
28                {{ artist.name }}
29              </a>
30            </td>
31          </tr>
32          {% endfor %}
33        </tbody>
34      </table>
35    </div>
36  </div>
37  {% endblock %}
```

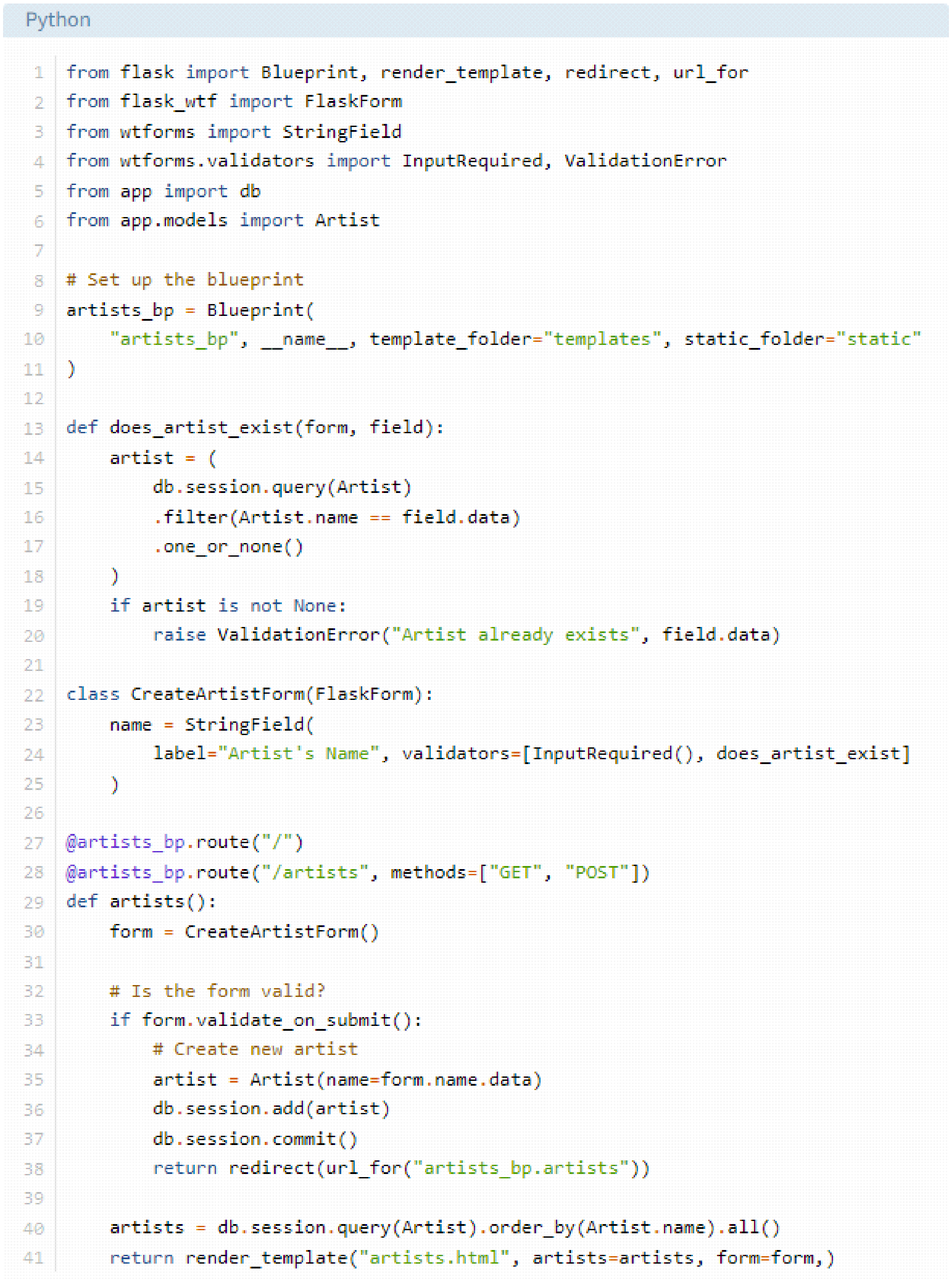
Here's what's going on in this Jinja2 template code:

- Line 1 uses Jinja2 template inheritance to build this template from the base.html template. The base.html template contains all the HTML5 boilerplate code as well as the Bootstrap navigation bar consistent across all pages of the site.

- Lines 3 to 37 contain the block content of the page, which is incorporated into the Jinja2 macro of the same name in the base.html base template.

- Lines 9 to 13 render the form to create a new artist. This uses the features of Flask-WTF to generate the form.
- Lines 24 to 32 create a for loop that renders the table of artist names.
- Lines 27 to 29 render the artist name as a link to the artist's album page showing the songs associated with a particular artist.

Here's the Python route that renders the page:



Python

```python
1   from flask import Blueprint, render_template, redirect, url_for
2   from flask_wtf import FlaskForm
3   from wtforms import StringField
4   from wtforms.validators import InputRequired, ValidationError
5   from app import db
6   from app.models import Artist
7
8   # Set up the blueprint
9   artists_bp = Blueprint(
10      "artists_bp", __name__, template_folder="templates", static_folder="static"
11  )
12
13  def does_artist_exist(form, field):
14      artist = (
15          db.session.query(Artist)
16          .filter(Artist.name == field.data)
17          .one_or_none()
18      )
19      if artist is not None:
20          raise ValidationError("Artist already exists", field.data)
21
22  class CreateArtistForm(FlaskForm):
23      name = StringField(
24          label="Artist's Name", validators=[InputRequired(), does_artist_exist]
25      )
26
27  @artists_bp.route("/")
28  @artists_bp.route("/artists", methods=["GET", "POST"])
29  def artists():
30      form = CreateArtistForm()
31
32      # Is the form valid?
33      if form.validate_on_submit():
34          # Create new artist
35          artist = Artist(name=form.name.data)
36          db.session.add(artist)
37          db.session.commit()
38          return redirect(url_for("artists_bp.artists"))
39
40      artists = db.session.query(Artist).order_by(Artist.name).all()
41      return render_template("artists.html", artists=artists, form=form,)
```

Lines 1 to 6 import all the modules necessary to render the page and initialize forms with data from the database.

- Lines 9 to 11 create the blueprint for the artists page.

- Lines 13 to 20 create a custom validator function for the Flask-WTF forms to make sure a request to create a new artist doesn't conflict with an already existing artist.

- Lines 22 to 25 create the form class to handle the artist form rendered in the browser and provide validation of the form field inputs.

- Lines 27 to 28 connect two routes to the artists() function they decorate.

- Line 30 creates an instance of the CreateArtistForm() class.

- Line 33 determines if the page was requested through the HTTP methods GET or POST (submit). If it was a POST, then it also validates the fields of the form and informs the user if the fields are invalid.

- Lines 35 to 37 create a new artist object, add it to the SQLAlchemy session, and commit the artist object to the database, persisting it.

- Line 38 redirects back to the artists page, which will be rerendered with the newly created artist.

- Line 40 runs an SQLAlchemy query to get all the artists in the database and sort them by Artist.name.

- Line 41 renders the artists page if the HTTP request method was a GET.

Conclusion:

SQLite, SQL, and SQLAlchemy are the tools used to move data contained in flat files to an SQLite database, access the data with SQL and SQLAlchemy, and provide that data through a web server.