CSCI S-33a

CS50's Web Programming
with Python and JavaScript

Harvard Summer School (https://www.summer.harvard.edu/)
Summer 2020

Lecture 4

Introduction

- So far, we've discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, and started using Django to create web applications.
- Today, we'll learn about using SQL and Django models to efficiently store and access data.

SQL

: SQL (https://www.w3schools.com/sql/), or Structured Query Language, is a programming language that allows us to update and query databases.

Databases

Before we get into how to use the SQL language, we should discuss how our data is stored. When using SQL, we'll work with a relational database (https://www.oracle.com/database/what-is-a-relational-database/#:~:text=A%20relational%20database%20is%20a,of%20representing%20data%20in%20tables.) where we can find all of our data stored in a number of tables (https://www.essentialsql.com/what-is-a-database-table/). Each of these tables is made up of a set number of columns and a flexible number of rows.

To illustrate how to work with SQL, we'll use the example of a website for an airline used to keep track of flights and passengers. In the following table, we see that we're keeping track of a number of flights, each of which has an `origin`, a `destination`, and a `duration`.

origin	destination	duration
New York	London	415
Shanghai	Paris	760
Istanbul	Tokyo	700
New York	Paris	435
Moscow	Paris	245
Lima	New York	455

There are several different relational database management systems that are commonly used to store information, and that can easily interact with SQL commands:

- MySQL (https://www.mysql.com/)
- PostgreSQL (https://www.postgresql.org/)
- SQLite (https://www.sqlite.org/index.html)
- …

The first two, MySQL and PostgreSQL, are heavier-duty database management systems that are typically run on servers separate from those running a website. SQLite, on the other hand, is a lighter-weight system that can store all of its data in a single file. We'll be using SQLite throughout this course, as it is the default system used by Django.

Column Types

Just as we worked with several different variable types in Python, SQLite has types (https://www.sqlite.org/datatype3.html) that represent different forms of information. Other management systems may have different data types, but all are fairly similar to those of SQLite:

- `TEXT` : For strings of text (Ex. a person's name)
- `NUMERIC` : A more general form of numeric data (Ex. A date or boolean value)
- `INTEGER` : Any non-decimal number (Ex. a person's age)
- `REAL` : Any real number (Ex. a person's weight)
- `BLOB` (Binary Large Object): Any other binary data that we may want to store in our database (Ex. an image)

Tables

Now, to actually get started with using SQL to interact with a database, let's begin by creating a new table. The command to create a new table (https://www.w3schools.com/sql/sql_create_table.asp) looks something like this:

```
CREATE TABLE flights(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    origin TEXT NOT NULL,
```

```
    destination TEXT NOT NULL,
    duration INTEGER NOT NULL
);
```

In the above command, we're creating a new table that we've decided to call `flights` , and we've added four columns to this table:

1. `id` : It is often helpful to have an number that allows us to uniquely identify each row in a table. Here we have specified that `id` is an integer, and also that it is our [primary key (https://www.w3schools.com/sql/sql_primarykey.ASP)](https://www.w3schools.com/sql/sql_primarykey.ASP), meaning it is our unique identifier. We have additionally specified that it will `AUTOINCREMENT` , which means we will not have to provide an id every time we add to the table because it will be done automatically.
2. `origin` : Here we've specified that this will be a text field, and by writing `NOT NULL` we have required that it have a value.
3. `destination` : Again we've specified that this will be a text field and prevented it from being null.
4. `duration` : Again this value cannot be null, but this time it is represented by an integer rather than as text.

We just saw the `NOT NULL` and `PRIMARY KEY` constraint when making a column, but there are several other [constraints (https://www.tutorialspoint.com/sqlite/sqlite_constraints.htm)](https://www.tutorialspoint.com/sqlite/sqlite_constraints.htm) available to us:

- `CHECK` : Makes sure certain constraints are met before allowing a row to be added/modified
- `DEFAULT` : Provides a default value if no value is given
- `NOT NULL` : Makes sure a value is provided
- `PRIMARY KEY` : Indicates this is the primary way of searching for a row in the database
- `UNIQUE` : Ensures that no two rows have the same value in that column.
- …

Now that we've seen how to create a table, let's look at how we can add rows to it. In SQL, we do this using the `INSERT` command:

```
INSERT INTO flights
    (origin, destination, duration)
    VALUES ("New York", "London", 415);
```

In the above command, we've specified the table name we wish to insert into, then provided a list of the column names we will be providing information on, and then specified the `VALUES` we would like to fill that row in the table, making sure the `VALUES` come in the same order as our corresponding list of columns. Note that we don't need to provide a value for `id` because it is automatically incrementing.

SELECT

Once a table has been populated with some rows, we'll probably want a way to access data within that table. We do this using SQL's [SELECT (https://www.w3schools.com/sql/sql_select.asp)](https://www.w3schools.com/sql/sql_select.asp) query. The simplest `SELECT` query into our flights table might look something like this:

```
SELECT * FROM flights;
```

The above command (*) retrieves all of the data from our flights table

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

It may be the case though that we don't really need all of the columns from the database, just origin and destination. To access just these

It may be the case though that we don't really need all of the columns from the database, just origin and destination. To access just these columns, we can replace the * with the column names we would like access to. The following query returns all of the origins and destinations.

```
SELECT origin, destination FROM flights;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

As our tables get larger and larger, we will also want to narrow down which rows our query returns. We do this by adding a WHERE (https://www.w3schools.com/sql/sql_where.asp) followed by some condition. For example, the following command selects only row with an `id` of `3`:

```
SELECT * FROM flights WHERE id = 3;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

e can filter by any column, not just `id` !

```
SELECT * FROM flights WHERE origin = "New York";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

Working with SQL in the Terminal

Now that we know some basic SQL commands, let's test them out in the terminal! In order to work with SQLite on your computer, you must first download SQLite (https://www.sqlite.org/download.html). (We won't use it in lecture, but you can also download DB Browser (https://sqlitebrowser.org/dl/) for a more user-friendly way to run SQL queries.)

We can start by creating a file for our database either by manually creating a new file, or running `touch flights.sql` in the terminal. Now, if we run `sqlite3 flights.sql` in the terminal, we'll be brought to a SQLite prompt where we can run SQL commands:

```
# Entering into the SQLite Prompt
(base) % sqlite3 flights.sql
SQLite version 3.26.0 2018-12-01 12:34:55
Enter ".help" for usage hints.

# Creating a new Table
sqlite> CREATE TABLE flights(
   ...>      id INTEGER PRIMARY KEY AUTOINCREMENT,
   ...>      origin TEXT NOT NULL,
   ...>      destination TEXT NOT NULL,
   ...>      duration INTEGER NOT NULL
   ...> );

# Listing all current tables (Just flights for now)
sqlite> .tables
flights

# Querying for everything within flights (Which is now empty)
sqlite> SELECT * FROM flights;

# Adding one flight
sqlite> INSERT INTO flights
   ...>      (origin, destination, duration)
   ...>      VALUES ("New York", "London", 415);

# Checking for new information, which we can now see
sqlite> SELECT * FROM flights;
1|New York|London|415

# Adding some more flights
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Shanghai", "Paris", 760);
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Istanbul", "Tokyo", 700);
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("New York", "Paris", 435);
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Moscow", "Paris", 245);
sqlite> INSERT INTO flights (origin, destination, duration) VALUES ("Lima", "New York", 455);

# Querying this new information
sqlite> SELECT * FROM flights;
1|New York|London|415
2|Shanghai|Paris|760
3|Istanbul|Tokyo|700
4|New York|Paris|435
5|Moscow|Paris|245
6|Lima|New York|455

# Changing the settings to make output more readable
sqlite> .mode columns
sqlite> .headers yes

# Querying all information again
sqlite> SELECT * FROM flights;
id          origin      destination  duration
----------  ----------  -----------  ----------
1           New York    London       415
2           Shanghai    Paris        760
3           Istanbul    Tokyo        700
4           New York    Paris        435
5           Moscow      Paris        245
6           Lima        New York     455

# Searching for just those flights originating in New York
sqlite> SELECT * FROM flights WHERE origin = "New York";
id          origin      destination  duration
----------  ----------  -----------  ----------
1           New York    London       415
4           New York    Paris        435
```

We can also use more than just equality to filter out our flights. For integer and real values, we can use greater than or less than:

```
SELECT * FROM flights WHERE duration > 500;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245

nd we can also use other logic ([AND, OR (https://www.w3schools.com/sql/sql_and_or.asp)](https://www.w3schools.com/sql/sql_and_or.asp)) like in Python:

```
SELECT * FROM flights WHERE duration > 500 AND destination = "Paris";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245

```
SELECT * FROM flights WHERE duration > 500 OR destination = "Paris";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245

We can also use the keyword IN (https://www.w3schools.com/sql/sql_in.asp) to see if a bit of data is one of several options:

```
SELECT * FROM flights WHERE origin IN ("New York", "Lima");
```

id	origin	destination	duration

We can even use regular expressions to search words more broadly using the LIKE (https://www.w3schools.com/sql/sql_like.asp) keyword. The below query finds all results with an `a` in the origin, by using `%` as a wildcard character.

```
SELECT * FROM flights WHERE origin LIKE "%a%";
```

Functions

There are also a number of SQL functions we can apply to the results of a query. These can be useful if we don't need all of the data returned by a query, but just some summary statistics of the data.

- AVERAGE (https://www.w3schools.com/sql/sql_count_avg_sum.asp)
- COUNT (https://www.w3schools.com/sql/sql_count_avg_sum.asp)
- MAX (https://www.w3schools.com/sql/sql_min_max.asp)
- MIN (https://www.w3schools.com/sql/sql_min_max.asp)
- SUM (https://www.w3schools.com/sql/sql_count_avg_sum.asp)
- …

UPDATE

We've now seen how to add to and search tables, but we may also want to be able update rows of a table that already exist. We do this using the UPDATE (https://www.w3schools.com/sql/sql_update.asp) command as shown below. As you may have guessed by reading this out loud, the command finds any flights that go from New York to London, and then sets their durations to 430.

```
UPDATE flights
    SET duration = 430
    WHERE origin = "New York"
    AND destination = "London";
```

DELETE

We also might want the ability to delete rows from our database, and we can do this using the DELETE (https://www.w3schools.com/sql/sql_delete.asp) command. The following code will remove all flights that land in Tokyo:

```
DELETE FROM flights WHERE destination = "Tokyo";
```

Other Clauses

There are a number of additional clauses we can use to control queries coming back to us

- LIMIT (https://www.w3schools.com/sql/sql_top.asp): Limits the number of results returned by a query
- ORDER BY (https://www.w3schools.com/sql/sql_orderby.asp): Orders the results based on a specified column
- GROUP BY (https://www.w3schools.com/sql/sql_groupby.asp): Groups results by a specified column

- HAVING (https://www.w3schools.com/sql/sql_having.asp): Allows for additional constraints based on the number of results

Joining Tables

So far, we've only been working with one table at a time, but many databases in practice are populated by a number of tables that all relate to each other in some way. In our flights example, let's imagine we also want to add an airport code to go with the city. The way our table is currently set up, we would have to add two more columns to go with each row. We would also be repeating information, as we would have to write in multiple places that city X is associated with code Y.

One way we can solve this problem is by deciding to have one table that keeps track of flights, and then another table keeping track of airports. The second table might look something like this

Now we have a table relating codes and cities, rather than storing an entire city name in our flights table, it will save storage space if we're able to just save the `id` of that airport. Therefore, we should rewrite the flights table accordingly. Since we're using the `id` column of the airports table to populate `origin_id` and `destination_id`, we call those values Foreign Keys (https://www.w3schools.com/sql/sql_foreignkey.asp)

In addition to flights and airports, an airline might also want to store data about its passengers, like which flight each passenger will be on. Using the power of relational databases, we can add another table that stores first and last names, and a foreign key representing the flight they are on

We can do even better than this though, as the same person may be on more than one flight. To account for this, we can create a `people` table that stores first and last names, and a `passengers` table that pairs people with flights

![People](images/people.png

ecause in this case a single person can be on many flights and a single flight can have many people, we call the relationship between `flights` and `people` a **Many to Many** relationship. The `passengers` table that connects the two is known as an **association table**.

JOIN Query

Although our data is now more efficiently stored, it seems like it may be harder to query our data. Thankfully, SQL has a JOIN (https://www.w3schools.com/sql/sql_join.asp) query where we can combine two tables for the purposes of another query.

For example, let's say we want to find the origin, destination, and first name of every trip a passenger is taking. Also for simplicity in this table, we're going to be using the unoptimized `passengers` table that includes the flight id, first name, and last name. The first part of this query looks fairly familiar:

```
SELECT first, origin, destination
FROM ...
```

But we run into a problem here because `first` is stored in the `passengers` table, while `origin` and `destination` are stored in the `flights` table. We solve this by joining the two tables using the fact that `flight_id` in the `passengers` table corresponds to `id` in the `flights` table:

```
SELECT first, origin, destination
FROM flights JOIN passengers
ON passengers.flight_id = flights.id;
```

We've just used something called an INNER JOIN (https://www.w3schools.com/sql/sql_join_inner.asp), which means we are ignoring rows that have no matches between the tables, but there are other types of joins, including **LEFT JOIN**s (https://www.w3schools.com/sql/sql_join_left.asp), **RIGHT JOIN**s (https://www.w3schools.com/sql/sql_join_right.asp), and **FULL OUTER JOIN**s (https://www.w3schools.com/sql/sql_join_full.asp), which we won't discuss here in detail.

Indexing:

One way we can make our queries more efficient when dealing with large tables is to create an index similar to the index you might see in the back of a textbook. For example, if we know that we'll often look up passengers by their last name, we could create an index from last name to id using the command:

```
CREATE INDEX name_index ON passengers (last);
```

SQL Vulnerabilities

Now that we know the basics of using SQL to work with data, it's important to point out the main vulnerabilities associated with using SQL. We'll start with SQL Injection (https://www.w3schools.com/sql/sql_injection.asp).

A SQL injection attack is when a malicious user enters SQL code as input on a site in order to bypass the sites security measures. For example, let's say we have a table storing usernames and passwords, and then a login form on the home site of a page. We may search for the user using a query such as:

```
SELECT * FROM users
WHERE username = username AND password = password;
```

A user named Harry might go to this site and type `harry` as a username and `12345` as a password, in which case the query would look like this:

```
SELECT * FROM users
WHERE username = "harry" AND password = "12345";
```

A hacker, on the other hand, might type `harry" --` as a username and nothing as a password. It turns out that `--` stands for a comment in SQL, meaning the query would look like:

```
SELECT * FROM users
WHERE username = "harry"--" AND password = "12345";
```

Because in this query the password checking has been commented out, the hacker can log into Harry's account without knowing their password. To solve this problem, we can use:

- Escape characters to make sure SQL treats the input as plain text and not as SQL code.
- An abstraction layer on top of SQL which includes its own escape sequence, so we don't have to write SQL queries ourselves.

The other main vulnerability when it comes to SQL is known as a Race Condition (https://searchstorage.techtarget.com/definition/race-condition#:~:text=A%20race%20condition%20is%20an,sequence%20to%20be%20done%20correctly.).

A race condition is a situation that occurs when multiple queries to a database occur simultaneously. When these are not adequately handled, problems can arise in the precise times that databases are updated. For example, let's say I have $150 in my bank account. A race condition could occur if I log into my bank account on both my phone and my laptop, and attempt to withdraw $100 on each device. If the bank's software developers did not deal with race conditions correctly, then I may be able to withdraw $200 from an account with only $150 in it. One potential solution for this problem would be locking the database. We could not allow any other interaction with the database until one transaction has been completed. In the bank example, after clicking navigating to the "Make a Withdrawl" page on my computer, the bank might not allow me to navigate to that page on my phone.

computer, the bank might not allow me to navigate to that page on my phone.

Django Models

: Django Models (https://docs.djangoproject.com/en/3.0/topics/db/models/) are a level of abstraction (https://techterms.com/definition/abstraction) on top of SQL that allow us to work with databases using Python classes and objects rather than direct SQL queries.

Let's get started on using models by creating a django project for our airline, and creating an app within that project.

```
django-admin startproject airline
cd airline
python manage.py startapp flights
```

Now we'll have to go through the process of adding an app as usual:

1. Add `flights` to the `INSTALLED_APPS` list in `settings.py`
2. Add a route for `flights` in `urls.py` :

```
path("flights/", include("flights.urls")),
```

1. Create a `urls.py` file within the `flights` application. And fill it with standard `urls.py` imports and lists.

Now, rather than creating actual paths and getting started on `views.py` , we'll create some models in the `models.py` file. In this file, we'll outline what data we want to store in our application. Then, Django will determine the SQL syntax necessary to store information on each of our models. Let's take a look at what a model for a single flight might look like:

```
class Flight(models.Model):
    origin = models.CharField(max_length=64)
    destination = models.CharField(max_length=64)
    duration = models.IntegerField()
```

Let's take a look at what's going on in this model definition:

- In the first line, we create a new model that **extends** Django's model class.
- Below, we add fields for origin, destination, and duration. The first two are Char Fields (https://docs.djangoproject.com/en/3.0/ref/forms/fields/#charfield), meaning they store strings, and the third is an Integer Field (https://docs.djangoproject.com/en/3.0/ref/forms/fields/#integerfield). These are just two of many built-in Django Field classes (https://docs.djangoproject.com/en/3.0/ref/forms/fields/#built-in-field-classes)
- We specify maximum lengths of 64 for the two Character Fields. you can check the specifications available for a given field by checking the documentation (https://docs.djangoproject.com/en/3.0/ref/forms/fields/#built-in-field-classes).

Migrations

Now, even though we've created a model, we do not yet have a database to store this information. to create a database from our models, we navigate to the main directory of our project and run the command.

```
python manage.py makemigrations
```

This command creates some Python files that will create or edit our database to be able to store what we have in our models. You should get an output that looks something like the one below, and if you navigate to your `migrations` directory, you'll notice a new file was created for us

ext, to apply these migrations to our database, we run the command

```
python manage.py migrate
```

Now, you'll see some default migrations have been applied along with our own, and you'll also notice that we now have a file called `db.sqlite3` in our project's directory

Shell

Now, to begin working adding information to and manipulating this database, we can enter Django's shell where we can run Python commands within our project.

```
python manage.py shell
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

```python
# Import our flight model
In [1]: from flights.models import Flight

# Create a new flight
In [2]: f = Flight(origin="New York", destination="London", duration=415)

# Instert that flight into our database
In [3]: f.save()

# Query for all flights stored in the database
In [4]: Flight.objects.all()
Out[4]: <QuerySet [<Flight: Flight object (1)>]>
```

When we query our database, we see that we get just one flight called `Flight object (1)`. This isn't a very informative name, but we can fix that. Inside `models.py`, we'll define a `__str__` function that provides instructions for how to turn a Flight object into a string:

```python
class Flight(models.Model):
    origin = models.CharField(max_length=64)
    destination = models.CharField(max_length=64)
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origin} to {self.destination}"
```

Now, when we go back to the shell, our output is a bit more readable:

```python
# Create a variable called flights to store the results of a query
In [7]: flights = Flight.objects.all()

# Displaying all flights
In [8]: flights
Out[8]: <QuerySet [<Flight: 1: New York to London>]>

# Isolating just the first flight
In [9]: flight = flights.first()

# Printing flight information
In [10]: flight
Out[10]: <Flight: 1: New York to London>

# Display flight id
In [11]: flight.id
Out[11]: 1

# Display flight origin
In [12]: flight.origin
Out[12]: 'New York'

# Display flight distination
In [13]: flight.destination
Out[13]: 'London'

# Display flight duration
In [14]: flight.duration
Out[14]: 415
```

This is a good start, but thinking back to earlier, we don't want to have to store the city name as an origin and destination for every flight, so we probably want another model for an airport that is somehow related to the flight model:

We probably want another model for an airport that is somehow related to the flight model.

```python
class Airport(models.Model):
    code = models.CharField(max_length=3)
    city = models.CharField(max_length=64)

    def __str__(self):
        return f"{self.city} ({self.code})"

class Flight(models.Model):
    origin = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="departures")
    destination = models.ForeignKey(Airport, on_delete=models.CASCADE, related_name="arrivals")
    duration = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origin} to {self.destination}"
```

We've seen everything in our new `Airport` class before, but the changes to the `origin` and `destination` fields within the `Flight` class are new to us:

- We specify that the `origin` and `destination` fields are each Foreign Keys (https://docs.djangoproject.com/en/3.0/topics/db/examples/many_to_one/), which means they refer to another object.
- By entering `Airport` as our first argument, we are specifying the type of object this field refers to.
- The next argument, `on_delete=models.CASCADE` gives instructions for what should happen if an airport is deleted. In this case, we specify that when an airport is deleted, all flights associated with it should also be deleted. There are several other options (https://docs.djangoproject.com/en/3.0/ref/models/fields/#django.db.models.ForeignKey.on_delete) in addition to `CASCADE` .
- We provide a related name (https://docs.djangoproject.com/en/3.0/ref/models/fields/#django.db.models.ForeignKey.related_name), which gives us a way to search for all flights with a given airport as their origin or destination.

Every time we make changes in `models.py` , we have to make migrations and then migrate. Note that you may have to delete your existing flight from New York to London, as it doesn't fit in with the new database structure.

```
# Create New Migrations
python manage.py makemigration

# Migrate
python manage.py migrate
```

Now, let's try these new models out in the Django shell:

```
# Import all models
In [1]: from flights.models import *

# Create some new airports
In [2]: jfk = Airport(code="JFK", city="New York")
In [4]: lhr = Airport(code="LHR", city="London")
In [6]: cdg = Airport(code="CDG", city="Paris")
In [9]: nrt = Airport(code="NRT", city="Tokyo")

# Save the airports to the database
In [3]: jfk.save()
In [5]: lhr.save()
In [8]: cdg.save()
In [10]: nrt.save()

# Add a flight and save it to the database
f = Flight(origin=jfk, destination=lhr, duration=414)
f.save()

# Display some info about the flight
In [14]: f
Out[14]: <Flight: 1: New York (JFK) to London (LHR)>
In [15]: f.origin
Out[15]: <Airport: New York (JFK)>

# Using the related name to query by airport of arrival:
In [17]: lhr.arrivals.all()
Out[17]: <QuerySet [<Flight: 1: New York (JFK) to London (LHR)>]>
```

Starting our application

We can now begin to build an application around this process of using models to interact with a database. Let's begin by creating an index route for our airline. Inside `urls.py` :

```
urlpatterns = [
    path('', views.index, name="index"),
]
```

Inside `views.py` :

```python
from django.shortcuts import render
from .models import Flight, Airport

# Create your views here.

def index(request):
    return render(request, "flights/index.html", {
        "flights": Flight.objects.all()
    })
```

Inside our new `layout.html` file:

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Flights</title>
    </head>
    <body>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

Inside a new `index.html` file:

```html
{% extends "flights/layout.html" %}

{% block body %}
    <h1>Flights:</h1>
    <ul>
        {% for flight in flights %}
            <li>Flight {{ flight.id }}: {{ flight.origin }} to {{ flight.destination }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

What we've done here is created a default page where we have a list of all flights we've created so far. When we open up the page now, it looks like this

Now, let's add some more flights to our application by returning to the Django shell:

```
# Using the filter command to find all airports based in New York
In [3]: Airport.objects.filter(city="New York")
Out[3]: <QuerySet [<Airport: New York (JFK)>]>

# Using the get command to get only one airport in New York
In [5]: Airport.objects.get(city="New York")
Out[5]: <Airport: New York (JFK)>

# Assigning some airports to variable names:
In [6]: jfk = Airport.objects.get(city="New York")
In [7]: cdg = Airport.objects.get(city="Paris")

# Creating and saving a new flight:
In [8]: f = Flight(origin=jfk, destination=cdg, duration=435)
In [9]: f.save()
```

Now, when we visit our site again

Django Admin

Since it is so common for developers to have to create new objects like we've been doing in the shell, Django comes with a default admin interface (https://docs.djangoproject.com/en/3.0/ref/contrib/admin/) that allows us to do this more easily. To begin using this tool, we must first create an administrative user:

```
(base) cleggett@Connors-MacBook-Pro airline % python manage.py createsuperuser
Username: user_a
Email address: a@a.com
Password:
Password (again):
Superuser created successfully.
```

Now, we must add our models to the admin application by entering the `admin.py` file within our app, and importing and registering our models. This tells Django which models we would like to have access to in the admin app.

```
from django.contrib import admin
from .models import Flight, Airport

# Register your models here.
admin.site.register(Flight)
admin.site.register(Airport)
```

Now, when we visit our site and add `/admin` to the url, we can log into a page that looks like this

After loggin in, you'll be brought to a page like the one below where you can create, edit, and delete objects stored in the database

Now, let's add a few more pages to our site. We'll begin by adding the ability to click on a flight to get more information about it. To do this, let's create a URL path that includes the `id` of a flight:

```
path("<int:flight_id>", views.flight, name="flight")
```

Then, in `views.py` we will create a `flight` function that takes in a flight id and renders a new html page:

```
def flight(request, flight_id):
    flight = Flight.objects.get(id=flight_id)
    return render(request, "flights/flight.html", {
        "flight": flight
```

```
    })
```

Now we'll create a template to display this flight information with a link back to the home page

```
{% extends "flights/layout.html" %}

{% block body %}
    <h1>Flight {{ flight.id }}</h1>
    <ul>
        <li>Origin: {{ flight.origin }}</li>
        <li>Destination: {{ flight.destination }}</li>
        <li>Duration: {{ flight.duration }} minutes</li>
    </ul>
    <a href="{% url 'index' %}">All Flights</a>
{% endblock %}
```

Finally, we need to add the ability to link from one page to another, so we'll modify our index page to include links:

```
{% extends "flights/layout.html" %}

{% block body %}
    <h1>Flights:</h1>
    <ul>
        {% for flight in flights %}
            <li><a href="{% url 'flight' flight.id %}">Flight {{ flight.id }}</a>: {{ flight.origin }} to {{ flight.destination }
        {% endfor %}
    </ul>
{% endblock %}
```

Now our homepage looks like this

And when we click on flight 5, for example, we're brought to this page

Many-to-Many Relationships

Now, let's work on integrating passengers into our models. We'll create a passenger model to start:

```
class Passenger(models.Model):
    first = models.CharField(max_length=64)
    last = models.CharField(max_length=64)
    flights = models.ManyToManyField(Flight, blank=True, related_name="passengers")

    def __str__(self):
        return f"{self.first} {self.last}"
```

- As we discussed, passengers have a **Many to Many** relationship with flights, which we describe in Django using the ManyToManyField.
- The first argument in this field is the class of objects that this one is related to.
- We have provided the argument `blank=True` which means a passenger can have no flights
- We have added a `related_name` that serves the same purpose as it did earlier: it will allow us to find all passengers on a given flight.

To actually make these changes, we must make migrations and migrate. We can then register the Passenger model in `admin.py` and visit the admin page to create some passengers!

Now that we've added some passengers, let's update our flight page so that it displays all passengers on a flight. We'll first visit `views.py` and update our flight view to provide a list of passengers as context. We access the list using the related name we defined earlier.

```
def flight(request, flight_id):
    flight = Flight.objects.get(id=flight_id)
```

```
        passengers = flight.passengers.all()
        return render(request, "flights/flight.html", {
            "flight": flight,
            "passengers": passengers
        })
```

Now, add a list of passengers to `flight.html` :

```html
<h2>Passengers:</h2>
<ul>
    {% for passenger in passengers %}
        <li>{{ passenger }}</li>
    {% empty %}
        <li>No Passengers.</li>
    {% endfor %}
</ul>
```

At this point, when we click on flight 5, we see

Now, let's work on giving visitors to our site the ability to book a flight. We'll do this by adding a booking route in `urls.py` :

```python
path("<int:flight_id>/book", views.book, name="book")
```

Now, we'll add a book function to `views.py` that adds a passenger to a flight:

```python
def book(request, flight_id):

    # For a post request, add a new flight
    if request.method == "POST":

        # Accessing the flight
        flight = Flight.objects.get(pk=flight_id)

        # Finding the passenger id from the submitted form data
        passenger_id = int(request.POST["passenger"])

        # Finding the passenger based on the id
        passenger = Passenger.objects.get(pk=passenger_id)

        # Add passenger to the flight
        passenger.flights.add(flight)

        # Redirect user to flight page
        return HttpResponseRedirect(reverse("flight", args=(flight.id,)))
```

Next, we'll add some context to our flight template so that the page has access to everyone who is not currently a passenger on the flight using Django's ability to [exclude (https://docs.djangoproject.com/en/3.0/topics/db/queries/#retrieving-specific-objects-with-filters)](https://docs.djangoproject.com/en/3.0/topics/db/queries/#retrieving-specific-objects-with-filters) certain objects from a query:

```python
def flight(request, flight_id):
    flight = Flight.objects.get(id=flight_id)
    passengers = flight.passengers.all()
    non_passengers = Passenger.objects.exclude(flights=flight).all()
    return render(request, "flights/flight.html", {
        "flight": flight,
        "passengers": passengers,
        "non_passengers": non_passengers
    })
```

Now, we'll add a form to our flight page's HTML using a select input field:

```html
<form action="{% url 'book' flight.id %}" method="post">
    {% csrf_token %}
    <select name="passenger" id="">
        {% for passenger in non_passengers %}
            <option value="{{ passenger.id }}">{{ passenger }}</option>
        {% endfor %}
    </select>
    <input type="submit">
</form>
```

Now, let's see what the site looks like when I go to a flight page and then add a passenger

![form](images/form.png)

Another advantage of using the Django admin app is that it is customizable. For example, if we wish to see all aspects of a flight in the admin interface, we can create a new class within `admin.py` and add it as an argument when registering the `Flight` model:

```python
class FlightAdmin(admin.ModelAdmin):
    list_display = ("id", "origin", "destination", "duration")

# Register your models here.
admin.site.register(Flight, FlightAdmin)
```

Now, when we visit the admin page for flights, we can see the `id` as well

heck out Django's admin documentation (https://docs.djangoproject.com/en/3.0/ref/contrib/admin/) to find more ways to customize the admin app.

Users

The last thing we'll discuss in lecture today is the idea of authentication, or allowing users to log in and out of a website. Fortunately, Django makes this very easy for us, so let's go through an example of how we would do this. We'll start by creating a new app called `users`. Here we'll go through all the normal steps of creating a new app, but in our new `urls.py` file, we'll add a few more routes:

```python
urlpatterns = [
    path('', views.index, name="index"),
    path("login", views.login_view, name="login"),
    path("logout", views.logout_view, name="logout")
]
```

Let's begin by creating a form where a user can log in. We'll create a `layout.html` file as always, and then create a `login.html` file which contains a form, and that displays a message if one exists.

```html
{% extends "users/layout.html" %}

{% block body %}
    {% if message %}
        <div>{{ message }}</div>
    {% endif %}

    <form action="{% url 'login' %}" method="post">
        {% csrf_token %}
        <input type="text", name="username", placeholder="Username">
        <input type="password", name="password", placeholder="Password">
        <input type="submit", value="Login">
    </form>
{% endblock %}
```

Now, in `views.py` , we'll add three functions:

```python
def index(request):
    # If no user is signed in, return to login page:
    if not request.user.is_authenticated:
        return HttpResponseRedirect(reverse("login"))
    return render(request, "users/user.html")

def login_view(request):
    return render(request, "users/login.html")

def logout_view(request):
    # Pass is a simple way to tell python to do nothing.
    pass
```

Next, we can head to the admin site and add some users. After doing that, we'll go back to `views.py` and update our `login_view` function to handle a `POST` request with a username and password:

```python
# Additional imports we'll need:
from django.contrib.auth import authenticate, login, logout

def login_view(request):
    if request.method == "POST":
        # Accessing username and password from form data
        username = request.POST["username"]
        password = request.POST["password"]

        # Check if username and password are correct, returning User object if so
        user = authenticate(request, username=username, password=password)

        # If user object is returned, log in and route to index page:
        if user:
            login(request, user)
            return HttpResponseRedirect(reverse("index"))
        # Otherwise, return login page again with new context
        else:
            return render(request, "users/login.html", {
                "message": "Invalid Credentials"
            })
    return render(request, "users/login.html")
```

Now, we'll create the `user.html` file that the `index` function renders when a user is authenticated:

```html
{% extends "users/layout.html" %}

{% block body %}
    <h1>Welcome, {{ request.user.first_name }}</h1>
    <ul>
        <li>Username: {{ request.user.username }}</li>
        <li>Email: {{ request.user.email }}</li>
    </ul>

    <a href="{% url 'logout' %}">Log Out</a>
{% endblock %}
```

Finally, to allow the user to log out, we'll update the `logout_view` function so that it uses Django's built-in `logout` function:

```python
def logout_view(request):
    logout(request)
    return render(request, "users/login.html", {
                "message": "Logged Out"
            })
```

Now that we're finished, here's a demonstration of the site

That's all for this lecture! Next time, we'll learn our second programming language of the course: JavaScript.