[**Django**](https://cs50.harvard.edu/web/notes/7/#django)

* Django is a much heavier weight web framework than Flask with a lot more out-of-the-box features that would’ve had to be built up manually and repetitively with a micro-framework like Flask.

[**Using Django**](https://cs50.harvard.edu/web/notes/7/#using-django)

* Pip install virtualenv, pip install virtualenvwrapper
* Virtualenv CS50-Lec7, cd CS50-lec7, pip install django
* Django-admin startproject lec7
* Django divides all of its web applications into ‘projects’, composed of different parts. To start a new project, run django-admin startproject projectname.

[**Project Components**](https://cs50.harvard.edu/web/notes/7/#project-components)

* Django creates a number of files with a new project:
  + \_\_init\_\_.py : defines the directory projectname as a Python ‘package’, a collection of multiple Python files
    - Django is built on the idea of packages. A web application can be made up of multiple packages, each serving a slightly different purpose, and Django will help manage these.
  + manage.py : a Python script that can be used to perform useful operations on a web application
  + settings.py : basic settings, like time zone, other applications installed in the project, what sort of database is used, etc.
  + urls.py : determines what URLs/routes can be accessed when using the web application
  + wsgi.py : a file that helps to deploy an application to a web server
  + project\_name/ : the directory for the project that contains all of the above files by default
* A Django project consists of one or more Django applications, or apps, which serves a particular purpose.

[**A Basic Application**](https://cs50.harvard.edu/web/notes/7/#a-basic-application)

* To create an app, inside the project directory (lec7, not CS50-lec7), run python manage.py startapp appname. This will create a directory appname inside of the project directory. appname will contain a number of files automatically.
* Inside of appname, views.py is analogous to application.py for a Flask application. It contains the code that determines what the user sees at a particular route.At first, it will look like this:
* from django.shortcuts import render
* # Create your views here.
* All view functions should take the request object as an argument. Like in Flask, this object will contain information about what sort of arguments were passed in to the request, etc. A baisc view might just return an simple HTTP response.
* from django.http import HttpResponse
* from django.shortcuts import render
* # Create your views here.
* def index(request):
* return HttpResponse("Hello, world!")
* That basic view, however, does not specify what route it is at. To do so, a new urls.py must be created inside the appname directory (this is a different urls.py than the project-level file of the same name). Each application will often have its own routes, and these separate urls.py help to signal that difference in functionality, keep things organized, and make apps easily reusable in other projects. appname/urls.py could look like this:
* from django.urls import path
* from . import views
* urlpatterns = [
* path("", views.index),
* ]
* from . import views imports views.py from the appname directory, so that URLs can be linked to views.
* urlpatterns is a list of all the URLs supported by this application.
* "" indicates the empty route.
* When the Django project starts up, it will only check the urls.py at the project level. So, the final step before this basic application is actually usable, appname/urls.py must be linked to the project’s urls.py, which starts off with some code in it already.
* from django.contrib import admin
* from django.urls import include, path
* urlpatterns = [
* path("admin/", admin.site.urls)
* ]
* To link the new path, simply add the path to urlpatterns:
* urlpatterns = [
* path("", include("appname.urls")),
* path("admin/", admin.site.urls)
* ]
  + The reason for this apparent complexity is to allow for routing amongst multiple different applications. This urls.py serves as the dispatch point for all those lower-level urls.py files.
* To run the application, run python manage.py runserver from lec7.

[**Flights Revisited**](https://cs50.harvard.edu/web/notes/7/#flights-revisited)

* To demonstrate Django more completely, the next example will reconstruct and expand upon the flight manager application that was originally built with Flask. The project name will be djangoair, and it will contain an application called flights.
* To start off, flights/urls.py:
* from django.urls import path
* from . import views
* urlpatterns = [
* path("", views.index),
* ]
  + These urls should be linked to djangoair/urls.py in the same way as the previous example.
* flights/views.py:
* from django.http import HttpResponse
* from django.shortcuts import render
* # Create your views here
* def index(request):
* return HttpResponse("Flights")
* This application is now at the same point as the previous example. The next step is to add the database. Django was designed for interacting with data, so it makes it very easy to do so. flights/models.py looks like this right now:
* from django.db import models
* # Create your models here
  + This the file to define the classes which will define the types of data being stored in the database. The information to be contained here is very analagous to the information created with Flask-SQLAlchemy.
* A model for a flight might look like this:
* class Flight(models.Model):
* origin = models.CharField(max\_length=64)
* destination = models.CharField(max\_length=64)
* duration = models.IntegerField()
  + Inherting models.Model just establishes this class as a Django model.
  + Django has a number of built-in types of fields that map to different types of data in a SQL database, for instance.
* Now, as with new URLs, the models must be linked to the Django project. In djangoair/settings.py, there is a list called INSTALLED\_APPS, pre-populated with Django’s installed apps. To add the flights app, flights.app.FlightsConfig should be added to that list.
* INSTALLED\_APPS = [
* 'flights.apps.FlightsConfig',
* 'django.contrib.admin',
* 'django.contrib.auth',
* 'django.contrib.contenttypes',
* 'django.contrib.sessions',
* 'django.contrib.messages',
* 'django.contrib.staticfiles',
* ]
* FlightsConfig is the class the defines the settings for the flights application.

[**Migrations**](https://cs50.harvard.edu/web/notes/7/#migrations)

* When building a web application, very rarely will all the tables be defined will all the correct columns from the beginning. Usually, tables are built up as the application grows, and the database will be modified. It would be tedious to change both the Django model code and run the SQL commands to modify the database.
* Django’s solution to this problem ‘migrations’. Django automatically detects and changes to models.py and automatically generates the necessary SQL code to make the necessary changes.
* To create the table for managing flights inside the database, run python manage.py makemigrations. This will look through model files for any changes and generate a ‘migration’, which represents the necessary changes for the database. Running this command will create a file migrations/0001\_initial.py:
* # Generated by Django 2.0 on 2018-07-19 22:14
* from django.db import migrations, models
* class Migration(migrations.Migration):
* initial = True
* dependencies = [
* ]
* operations = [
* migrations.CreateModel(
* name='Flight',
* fields=[
* ('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),
* ('origin', models.CharField(max\_length=64)),
* ('destination', models.CharField(max\_length=64)),
* ('duration', models.IntegerField()),
* ],
* ),
* ]
  + Inside the class Migration is a list operations, which contains everything that should happen to the database. In this case, the model Flight should be created, with fields id, origin, destination, and duration.
  + Note that id was never specified in models.py. Django adds this column by default.
* The command python manage.py sqlmigrate flights(appname) 0001(migration id) will produce the SQL code that actually corresponds to this migration. This command doesn’t need to be run, but it is helpful in demonstrating what’s actually going on. The SQL command is very similar to what has been shown before, but it didn’t need to be written. It is all generated by Django’s migration system.
* To actually apply this migration to the database, run python manage.py migrate, which will apply the new migration as well as some default Django ones.
* The database that is actually being used here is defined in djangoair/settings.py in the DATABASES dictionary. By default, it uses a SQLite 3 (another version of SQL that uses a local file for a database) and the database file db.sqlite3.

[**Django Shell**](https://cs50.harvard.edu/web/notes/7/#django-shell)

* Django provides a shell, similar to Python’s interpreter, that allows for direct modification of the database. Start the shell by running python manage.py shell. Inside the shell, Python commands can be run.
* To create a new flight, the following commands can be run inside the shell.
* from flights.models import Flight
* f = Flight(origin="New York", destination="London", duration=415)
* f.save()
* Flight.objects.all()
* # Returns <QuerySet [<Flight: Flight object(1)>]>
  + f.save() is analogous to SQL’s COMMIT.
  + A QuerySet is like a list, with added functionality.
* The representation of the QuerySet that the shell returns isn’t really readable or helpful. To produce a more useful, string representation of a flight, a \_\_str\_\_ function can be added to Flight class in flights/models.py.
* def \_\_str\_\_(self):
* return f"{self.id} - {self.origin} to {self.destination}"
  + For any class, not just in Django, a \_\_str\_\_ function defines what an object should like look when printed, whether to a terminal, an HTML page, etc.
* Back to the shell:
* Flight.objects.all()
* # Returns <QuerySet [<Flight: 1 - New York to London>]>
* f = Flight.objects.first()
* f
* # Returns <Flight: 1 - New York to London>
* f.origin()
* # Returns 'New York'
* f.id
* # Returns 1
* f.delete()
* # Deletes the flight as expected

[**Better Models**](https://cs50.harvard.edu/web/notes/7/#better-models)

* A more robust design for the Flight model would have an id field that links to a table of airports instead of just text for origins and destinations. To do so, a new Airport model must first be created.
* 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})"
* Then, the Flight model can be modified appropriately, with origin and destination being ForeignKeys.
* 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")
  + Django models allow for specific behavior when an airport, for instance, is deleteda. on\_delete=models.CASCADE means that if an airport is deleted, all flights with that airport as an origin or destination will be deleted as well.
  + related\_name allows for the accessing of all flights departing from or arriving at a particular airport, using the keys deparatures or arrivals.
  + Note that there is no literal definitions of origin and destination as IDs, nor any actual commands for how to associate the two tables. The only things specified is that origin and destination should be Airports. All of the work to make that happen is left to Django.
* To apply these changes, they must be migrated in the same fashion as before. Now, in the shell, it’s a lot easier and more intuitive to create flights.
* from flights.models import Airport, Flight
* jfk = Airport(code="JFK", city="New York City")
* lhr = Airport(code="LHR", city="London")
* jfk.save()
* lhr.save()
* f = Flight(origin=jfk, destination=lhr, duration=415)
* f.save()
* f.origin
* # Returns <Airport: New York City (JFK)>
* f.origin.code
* # Returns 'JFK'
* jfk.departures.all()
* # Returns <QuerySet [<Flight: 1 - New York City (JFK) to London (LHR)>]>

[**Rendering Templates**](https://cs50.harvard.edu/web/notes/7/#rendering-templates)

* Similar to Flask, in order to render an HTML template, the rendered template should be returned by the function which handles a route. For Django, that’s in flights/views.py.
* def index(request)
* return render(request, "flights/index.html")
  + The second argument to render is simply the path to the template to be rendered.
  + These should be stored in a path like so: flights/templates/flights/index.html. Note that render takes a path starting from the template folder. The apparent redudancy of this path, although not strictly necessary in this example, is good practice to avoid issues where multiples applications might have their own index.html.
* index.html can be simple for now.
* <html>
* <head>
* <title>Flights</title>
* </head>
* <body>
* <h1>Flights</h1>
* </body>
* </html>
* To display information about flights, Django’s templating system, which is very similar to Jinja, can be used. Django passes information into a template via the context dictionary.
* from .models import Flight
* def index(request)
* context = {
* "flights": Flights.objects.all()
* }
* return render(request, "flights/index.html", context)
* <body>
* <h1>Flights</h1>
* <ul>
* {% for flight in flights %}
* <li>
* {{ flight }}
* </li>
* {% endfor %}
* </ul>
* </body>

[**Admin**](https://cs50.harvard.edu/web/notes/7/#admin)

* Admin is a built in Django app that makes it very easy to add or modify existing data on a web page. Note that this is a task that would require a good bit of code in Flask. This is perhaps one of the most powerful features of Django, especially when it comes to dealing with and manipulating data.
* flights/admin.py starts out like this.
* from django.contrib import admin
* # Register your models here.
* Adding the Airport and Flight models is simple.
* from django.contrib import admin
* from .models import Airpot, Flight
* # Register your models here.
* admin.site.register(Airport)
* admin.site.register(Flight)
  + This allows the admin app to manipulate airports and flights.
* To access the admin site online, a user must log in. This alone is a task that would be quite tedious in Flask, but again, Django comes with this functionality built-in. The first step is to create a ‘superuser’ account with access to everything: python manage.py createsuperuser. Django will then prompt for a username, email address, and password. This data will then be entered into a users table, entirely taken care of by Django.
* The admin site is already linked by default in the project’s urls.py at the admin/ route. On the admin site, a user can log in and manipulate the data. The admin interface is straightforward and easily navigated. Note that this admin interface isn’t meant to be used by all users of the website, but rather just content managers to dothings like populate models and add information, whearas users will view that information in a separately rendered page.

[**Adding More Routes**](https://cs50.harvard.edu/web/notes/7/#adding-more-routes)

* To add more routes, for specific flight info, for example, the URLs just need to be added to flights/urls.py along with the corresponding view in flights/views.py and template in templates/flights.
* urlpatterns = [
* path("", views.index),
* path("<int:flight\_id>", views.flight),
* ]
  + The syntax for creating routes that accept arguments is very similar to Flask’s.
  + def flight(request, flight\_id):
  + try:
  + flight = Flight.objects.get(pk=flight\_id)
  + except Flight.DoesNotExist:
  + raise Http404("Flight does not exist")
  + context = {
  + "flight": flight,
  + }
  + return render(request, "flights/flight.html", context)
  + Because flight\_id was parameter in the URL, flight\_id gets passed to the flight view.
  + pk stands for ‘primary key’.
  + DoesNotExist is a special exception built into Django models.
  + Http404 is another built-in Django feature (imported from django.http) that simply raises a 404 error. ``` html

**Flight {{ flight.id }}**

* + Origin: {{ flight.origin }}
  + Destination: {{ flight.destination }}

```

* + head contents can be the same as index.html for now. Note the current redundancy in HTML templates.

[**Template Inheritance**](https://cs50.harvard.edu/web/notes/7/#template-inheritance)

* Template inheritance for HTML pages works much the same way in Django as in Flask. Here’s what a generic template base.html could look like:
* <html>
* <head>
* <title>{% block title %}{% endblock %}</title>
* </head>
* <body>
* {% block body %}
* {% endblock %}
* </body>
* </html>
* Now, index.html and flight.html can be simplified.
* {% extends "flight/base.html" %}
* {% block title %}
* Flights
* {% endblock %}
* {% body block %}
* <h1>Flights</h1>
* <uL>
* {% for flight in flights $}
* <li>
* <a href="{% for flight in flights $}
* <li>
* <a href="{% url 'flight' flight.id %}">{{ flight }}</a>
* </li>
* {% endfor %}
* </ul>
* {% endblock %}

and

{% extends "flight/base.html" %}

{% block title %}

Flight {{ flight.id }}

{% endblock %}

<h1>Flight {{ flight.id }}</h1>

<ul>

<li>Origin: {{ flight.origin }}</li>

<li>Destination: {{ flight.destination }}</li>

</ul>

<a href="{% url 'index' %}">Back to full listing</a>

{% block body %}

{% endblock %}

[**Model Relationships**](https://cs50.harvard.edu/web/notes/7/#model-relationships)

* Before, with Flask and SQL, in order to link passengers to flights, there was an flight ID column in the passenger table so that each passenger can be associated with a flight. The problem with this approach is that each passenger can only be on a single flight. What is more desirable is a ‘many-to-many’ relationship, in which a passenger can be on multiple flights and a flight can have multiple passengers. A common paradigm for this is to implement an ‘in-between table’, which simply has two columns, one for a passenger ID and one for a flight ID, with as many rows as are necessary. Django allows for this, but does the work of implementing the in-between table automatically.
* The first step is to implement a passenger model in flights/models.py.
* 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}"
  + Before, when associating two tables, models.ForeignKey was used. models.ManyToManyField allows for the desired behavior of a many-to-many relationship.
  + blank=True allows for a passenger to be be associated with no flights.
  + Like before, related\_name allows for the querying of all passengers on a given flight.
* Updating the database as before, with python manage.py makemigrations and inspecting the SQL with python manage.py sqlmigrate flights 0003 reveals code for creating a table flights\_passengers, as specified, but also a table flights\_passengers\_flights, which was not specified, but is the in-betweent table that was automatically generated.
* After finishing the migration with python manage.py migrate, the shell can be used to try out these new models.
* from flights.models import Flight, Passenger
* f = Flight.objects.get(pk=1)
* f
* # Returns <Flight: 1 - New York City (JFK) to London (LHR)>
* p = Passenger(first="Alice", last="Adams")
* p.save()
* p.flights.add(f)
* p.flights.all()
* # Returns <QuerySet [<Flight: 1 - New York City (JFK) to London (LHR)>]>
* f.passengers.all()
* # Returns <QuerySet [<Passenger: Alice Adams>]>
* The flight view and its corresponding HTML can be updated to now display passenger info.
* def flight(request, flight\_id):
* try:
* flight = Flight.objects.get(pk=flight\_id)
* except Flight.DoesNotExist:
* raise Http404("Flight does not exist")
* context = {
* "flight": flight,
* "passengers": flight.passengers.all(),
* }
* return render(request, "flights/flight.html", context)

and

<h2>Passengers</h2>

<ul>

{% for passenger in passengers %}

<li>{{ passenger }}</li>

{% empty %}

<li>No passengers</li>

{% endfor %}

</ul>

* + {% empty %} executes if passengers is empty.
* The Passenger model can also be added to admin and modified on the admin application in the same was before.

[**User Registration**](https://cs50.harvard.edu/web/notes/7/#user-registration)

* The first step to creating a web UI for use flight registration might creating a new route, along with a corresponding view and HTML template.
* urlpatterns = [
* path("", views.index, name="index"),
* path("<int:flight\_id>", views.flight, name="flight"),
* path("<int:flight\_id>/book", views.book, name="book")
* ]
* def book(request, flight\_id):
* try:
* passenger\_id = int(request.POST["passenger"])
* flight = Flight.objects.get(pk=flight\_id)
* passenger = Passenger.objects.get(pk=passenger\_id)
* except KeyError:
* return render(request, "flights/error.html", {"message": "No selection."})
* except Flight.DoesNotExist:
* return render(request, "flights/error.html", {"message": "No flight."})
* except Passenger.DoesNotExist:
* return render(request, "flights/error.html", {"message": "No passenger."})
* passenger.flights.add(flight)
* return HttpResponseRedirect(reverse("flight", args=(flight\_id,)))
  + This code is written on the assumption that the user will submit a web form via a POST request with one argument being named passenger.
  + A KeyError will be raised if either a POST request wasn’t submitted or the passenger argument wasn’t provided, leaving no data to be extracted.
  + flights/error.html will be a new generic template to render any number of error messages.
  + HttpResponseRedirect (imported from django.http) is used to send the user to their flight page after being registered for it.
  + reverse() (imported from django.urls) returns the URL given the route name. Arguments can also be passed as a tuple.
* Assuming that creating a passenger is a separate process from registering a passenger for a flight, when the user goes to register for a flight, they should only be able to select from created passengers. To do so, all the ‘non-passengers’ on a flight should be passed into the flight.html template.
* def flight(request, flight\_id):
* try:
* flight = Flight.objects.get(pk=flight\_id)
* except Flight.DoesNotExist:
* raise Http404("Flight does not exist")
* context = {
* "flight": flight,
* "passengers": flight.passengers.all(),
* "non\_passengers": Passenger.objects.exclude(flights=flight).all()
* }
* return render(request, "flights/flight.html", context)
  + Passenger.objects returns all passenger objects, which can then be filtered in a variety of ways. exclude removes objects with a particular property; in this case, all passengers on the current flight are excluded.

{% if non\_passengers %}

<h2>Add a Passenger</h2>

<form action="{% url 'book' flight.id %}" method="post">

<select name="passenger">

{% for passenger in non\_passengers %}

<option value="{{ passenger.id }}">{{ passenger }}</option>

{% endfor %}

</select>

<input type="submit" value="Book Flight" />

</form>

{% else %}

<div>No passengers to add.</div>

{% endif %}

* The enclosing if block only allows for registration if there is someone to register.
* Here, the passenger select element is the corresponding data that’s being sent back to the book view, and inside passenger is passenger.id, which is what is expected.

[**Cross-Site Request Forgery**](https://cs50.harvard.edu/web/notes/7/#cross-site-request-forgery)

* Although the booking functionality looks nearly complete, when the registration form is submitted, Django will not allow the user to be redirected to their flight page, and will instead produce a 403 Forbidden error: CSRF verification failed. Request aborted. CSRF (Cross-Site Request Forgery) is a potential security vulnerability in forms whereby someone could forge where the form is coming from. Django is built in to protect these type of attacks. To allow for this nonetheless, a little bit of extra syntax must be added whenever dealing with a form in Django.

<form action="{% url 'book' flight.id %}" method="post">

{% csrf\_token %}

<select name="passenger">

{% for passenger in non\_passengers %}

<option value="{{ passenger.id }}">{{ passenger }}</option>

{% endfor %}

</select>

<input type="submit" value="Book Flight" />

</form>

* When the form is submitted, a CSRF token is submitted with it to allow Django to verify that is indeed the same web application is submitting the request.

[**Modifying Admin**](https://cs50.harvard.edu/web/notes/7/#modifying-admin)

* Django’s admin interface can be extended to allow for custom behavior. Returning to the flights example, here’s how flights/admin.py could be modified.
* from django.contrib import admin
* from .models import Airport, Flight, Passenger
* # Register your models here.
* class PassengerInline(admin.StackedInline):
* model = Passenger.flights.through
* extra = 1
* class FlightAdmin(admin.ModelAdmin):
* inlines = [PassengerInline]
* class PassengerAdmin(admin.ModelAdmin):
* filter\_horizontal = ("flights",)
* admin.site.register(Airport)
* admin.site.register(Flight, FlightAdmin)
* admin.site.register(Passenger, PassengerAdmin)

B

* + Because the Flights model does not have a reference to Passengers, managing the flights on the admin app does not allow for the addition or removal of passengers in the same way that flights can be added or removed to a passenger. This can be solved by creating the PassengerInline class, which inherits from the built-in class StackedInline that allows for the addition of new relationships between objects. PassengerInline represents the place in the UI where a flight’s passengers can be modified.
  + Passenger.flights.through refers to the in-between table linking flights and passengers. By setting model to this in-between table, that table is associated with PassengerInline.
  + extra = 1 sets the number of passengers which can be edited at a time to 1.
  + FlightAdmin is a new class which inherits from ModelAdmin, and contains a special set of configurations only to be used when editing passengers. These settings are applied by passing FlightAdmin to admin.site.regiser.
  + inlines contains all additional inline modification sections for the admin page, which in this case only contains PassengerInline.
  + filter\_horizontal helps to manipulate what flights a passenger is on. It simply allows for an additional UI element on the admin app to make it easy to add or remove flights that a passenger is on.

[**Static Files**](https://cs50.harvard.edu/web/notes/7/#static-files)

* To use external static files, like .css or .js files, some special Django syntax has to be used. A base template with static files might look like this:

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>{% block title %}{% endblock %}</title>

<link rel="stylesheet" href="{% static 'flights/styles.css' %}"/>

</head>

<body>

{% block body %}

{% endblock %}

</body>

</html>

\* `{% load static %}` allows for the use of static files.

\* Any static file must have its `href` formattaed as `"{% static 'path/static.css' %}"`. \* Inside of the application directory (e.g. `flights`), there should be a `static` directory containing a directory for the application which in turn contains any static files. So, the entire hierarchy would look like `flights/static/flights/styles.css`. This is exactly analogous to how templates are stored.

[**Login and Authentication**](https://cs50.harvard.edu/web/notes/7/#login-and-authentication)

* Authentication and Authorization is a built-in app designed to handle users accounts and log-in functionality. This last example features this account system in an app called users. users/urls.py looks like this:
* from django.urls import path
* from . import views
* urlpatterns = [
* path("", views.index, name="index"),
* path("login", views.login\_view, name="login"),
* path("logout", views.logout\_view, name="logout")
* users/views.py:
* from django.contrib.auth import authenticate, login, logout
* from django.http import HttpResponse, HttpResponseRedirect
* from django.shortcuts import render
* from django.urls import reverse
* # Create your views here.
* def index(request):
* if not request.user.is\_authenticated:
* return render(request, "users/login.html", {"message": None})
* context = {
* "user": request.user
* }
* return render(request, "users/user.html", context)
* def login\_view(request):
* username = request.POST["username"]
* password = request.POST["password"]
* user = authenticate(request, username=username, password=password)
* if user is not None:
* login(request, user)
* return HttpResponseRedirect(reverse("index"))
* else:
* return render(request, "users/login.html", {"message": "Invalid credentials."})
* def logout\_view(request):
* logout(request)
* return render(request, "users/login.html", {"message": "Logged out."})
  + django.contrib.auth is Django’s authentication package, which contains the User model, along with the functions authenticate ,login, and logout.
  + request.user.is\_authenticated is true if the user has logged in. If they aren’t logged in, they are redirectd to the login page. If they are, the user is directed to their user page.
  + login\_view first checks that a user exists with authenticate, which takes the user’s username and password and returns that user object.
  + login takes a user and logs them into the authentication system.
  + logout simply logs the user out.
* login.html:

{% block body %}

<h1>Login</h1>

{% if message %}

<div>

{{ message }}

</div>

{% endif %}

<form action="{% url 'login' %}" method="post">

{% csrf\_token %}

<input name="username" type="text"/>

<input name="password" type="password"/>

<input type="submit" value="Login"/>

</form>

{% endblock %}

* user.html:

{% block body %}

<h1>Hello, {{ user.first\_name }}</h1>

<ul>

<li>Currently logged in as: {{ user.username }}</li>

<li><a href="{% url 'logout' %}">Logout</a></li>

</ul>

{% endblock %}

* The User contains fields such as first\_name, last\_name, username, etc., but can also be extended.
* Registering a new user entails adding a new user to the database. This can be done through the admin interface with a superuser account or using the shell:
* from django.contrib.auth.models import User
* user = User.objects.create\_user("alice", "alice@something.com", "alice12345")
* user.first\_name = "Alice"
* user.save()
* create\_user takes the arguments username, e-mail, password.