**Cookies and Sessions**

**Cookies**

Cookie Definition: HTTP cookies (also called web cookies, Internet cookies, browser cookies, or simply cookies) are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer/device by the user’s web browser. Cookies are placed on the device used to access a website, and more than one cookie may be placed on a user’s device during a session.

* Created by the server: The server creates some cookie data, and then sends it to the client browser
* Stored on the client: The client browser stores the cookie data in the client device’s memory, and the client sends the cookie when it communicates with the server
* Cookie expiration: temporary or persistent
  + A temporary or session cookie (AKA in-memory cookie, transient cookie or non-persistent cookie) exists only in temporary memory while the user navigates a website. Session cookies expire or are deleted when the user closes the web browser. Session cookies are identified by the browser by the absence of an expiration date assigned to them. Session cookies should NOT be confused with sessions.
    - Effectively a session cookie is a way of marking each browser, that the web server meets, with a big random number. Using Firefox, in developer tools, you can find that cookie labeled as the “sessionid”. The sessionid is used to lookup the session database which is stored on the server.
  + A persistent cookie, AKA tracking cookie, expires at a specific date or after a specific length of time. For the persistent cookie's lifespan set by its creator, its information will be transmitted to the server every time the user visits the website that it belongs to, or every time the user views a resource belonging to that website from another website (such as an advertisement).
* To have the web server create a cookie on the web browser, use the [HttpResponse.set\_cookie() method](https://docs.djangoproject.com/en/3.0/ref/request-response/#django.http.HttpResponse):
  + resp.set\_cookie('zap', 42) # No expired date = until browser close
  + resp.set\_cookie('sakaicar', 42, max\_age=1000) # seconds until expire
  + Where “resp” is an HttpResponse object
    - e.g.,   
      resp = HttpResponse('In a view – some random text to be sent to the browser client')
* request.COOKIE
  + Every time a client browser makes a request to a server, it also attaches all its cookies associated with that web server, to the request. In the request object, the COOKIE attribute contains the key-value pairs of the cookie.
  + HttpRequest.COOKIES is an attribute – it is a dictionary variable containing all the client browser’s cookies (associated with the web server to where the client HttpRequest is being sent). The keys and values in this dictionary variable are strings.
  + Here is an example of a views.py function that accesses the COOKIE attribute to specifically manipulate a cookie named ‘zap” and another cookie named ‘sakaicar”:

from django.http import HttpResponse

def cookie(request):

print(request.COOKIES)

oldval = request.COOKIES.get('zap', None)

resp = HttpResponse('In a view - the zap cookie value is '+str(oldval))

if oldval :

resp.set\_cookie('zap', int(oldval)+1) # No expired date

else :

resp.set\_cookie('zap', 42) # No expired date = until browser close

resp.set\_cookie('sakaicar', 42, max\_age=1000) # seconds until expire

return resp

* + In this example, the session identifier cookie, ‘sessionid’, cookie would also be stored in the HttpResponse.COOKIE attribute.

**Django Sessions**

Session Definition: an HTTP Session is a temporary and interactive information interchange between a browser client and a web server. An established communication session can involve multiple messages in each direction. A session is stateful – in this case, the web server holds current state information in a database about the session history to be able to communicate with the browser client (as opposed to stateless communication, where the communication consists of independent requests with responses).

* In most server applications, as soon as we start a session for a new (unmarked) browser we create a session.
* The web server sets and sends a session cookie to be stored in the browser, which indicates the session id in use – this action gives this browser a unique “mark”.
  + The creation and destruction of sessions is handled by a Django middleware that we use in our applications
* The session identifier, sessionid, is a large, random number that the web server places in a browser cookie the first time we encounter a browser
  + This number is used to pick from the many sessions that the server has active at any one time.
* In Django, you have to enable the automatic setting of a sessionid cookie in the Dango project settings.py file, and specifically in the MIDDLEWARE list variable:

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

'social\_django.middleware.SocialAuthExceptionMiddleware', # Add

]

* Server software stores data in the session that it wants to have from one request to another from the same browser.
  + HttpRequest.session is an attribute of the HttpRequest class. It is readable and writable, dictionary-like object that represents the current session.
  + In Django, the HttpRequest session data is placed in a database stored on the web server.
    - If you are using SQLite3, it is stored in a SQLite3 table that is part of your project model
  + Here is an example of a views.py function that accesses the HttpRequest.session attribute to access and manipulate a session key-value pair with a key named ‘num\_visits’:

def myview(request):

num\_visits = request.session.get('num\_visits', 0) + 1

# the above line creates the variable ‘num\_visits’

# with an initial value of 0, and then increments by 1

# every time the page is loaded in a request

request.session['num\_visits'] = num\_visits

resp = HttpResponse('view count='+str(num\_visits))

resp.set\_cookie('dj4e\_cookie', 'b6d7a951', max\_age=1000)

return resp

* + It’s important to note that ‘num\_visits’ is not a cookie nor is it the entire session – rather, it is *one piece* of the data that gets set in the client browser-web server session
  + The Django web server uses the sessionid to locate and access the correct database instance which contains the relevant information for that specific browser-server session.
  + In Django, we can find the session data in a table named ‘django\_session’ that is part of our project model
    - This table will have the following row structure:
      * the sessionid
      * the data encoded in Base64, in our example num\_visits is the only key-value pair in the data
        + The data is organized in JSON format, which is helpful if there is a more complex set of data in the session
      * the session expiration date-time

**Assignment/Instructions Week #1: Building a Django Application Webpage**

Open a Django shell in the home directory of your Django project, move to your projects folder, and run “startapp”

* on pythonanywhere account, my home directory would be: /home/paAccountName
* More specifically, run the following commands:
  + workon django3 # as needed
  + cd ~/django\_projects/mysite
  + python3 manage.py startapp home

Create any template files and place them in the appropriate directory:

* /home/paAccountName/projectName/appName/templates/appName/templateName.html
* e.g.:
  + ~/django\_projects/mysite/home/templates/home/main.html

Edit the project urls.py file to add the appropriate path route command

* e.g.,
  + go to: ~/django\_projects/mysite/mysite/urls.py
  + add the following path route:  
    path('', TemplateView.as\_view(template\_name='home/main.html')),

Then edit the file ~/ projectsName/ settings.py and add a line to load the 'appName' application with the format: 'appName.apps.AppNameConfig

* eg., for a new app named ‘home’:
  + Edit the file ~/django\_projects/mysite/mysite/settings.py
  + For an app named “home”, you would add a line like this to settings.py (note the case settings):

INSTALLED\_APPS = [

'polls.apps.PollsConfig',

'home.apps.HomeConfig',

'hello.apps.HelloConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Move to your project folder and run manage.py. More specifically:

* python3 manage.py check
* Check your changes and correct any syntax errors, then Reload your web application (on the pythonanywhere “Web” selection at top of dashboard page or from “hamburger” icon pulldown).

**Users and Authentication**

**Creating and Managing Users in Django**

Django login/logout system

* Django comes with its own user administration system and interface
* This allows the developer to add users, superusers, groups, permissions

Superusers

* Superusers have all permissions
* To create a superuser, you must give it a name and an email address - the email address is for password recovery
* By creating a superuser, you are adding row to a table in your SQLite (or whatever database you are using with Django)
* You can wipe out your database and start fresh

*Refresher on rebuilding the Django database and creating a superuser:*

* *To remove the database, go to your project folder, and type:*
  + *rm db.sqlite*
* *To rebuild your database:*
  + *python3 manage.py migrate*
* *To create a superuser:*
  + *python3 manage.py createsuperuser*

**Login and Logout URLs in Django**

Logging into an app puts data into a session about the current user and their name, their email. Logging out removes the data. Logging in is NOT a session.

* Sessions are constructed on top of cookies, and logins are constructed on top of sessions.
* In a single session, you could login and logout as the same or multiple users

Enabling user authentication in Django

* User authentication and administration is automatically enabled by default in the project settings.py file.
* In /home/paAccountName/django\_projects/mysite/mysite/settings.py:

INSTALLED\_APPS = [

'polls.apps.PollsConfig',

'home.apps.HomeConfig',

'hello.apps.HelloConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

* The path to the authentication and administration tools must be added by the app developer into the project urls.py file
* In /home/paAccountName/django\_projects/mysite/mysite/urls.py:

urlpatterns = [

This line must be manually added

path('admin/', admin.site.urls),

path(‘accounts’, include(‘django.contrib.auth.urls)),

path('polls/', include('polls.urls')),

path('hello/', include('hello.urls')),

path('', TemplateView.as\_view(template\_name='home/main.html')),

url(r'^site/(?P<path>.\*)$', serve,

{'document\_root': SITE\_ROOT, 'show\_indexes': True},

name='site\_path'

),

]

* In django.contrib.auth.urls, there is a named view (using the “name =viewName” format). This is a view in that Django automatically writes into a file.
  + In the file there is a
    - name = 'login'
    - name = 'logout'
  + This allows the developer to write a reverse lookup for 'login' and 'logout'
* In an application webpage, the app developer may require that users are logged in to access the page – i.e., the page is protected by login. In this case, we want the user to access the page, go to a login screen and return to the very same page.
  + this required behavior can be enforced with the “next” parameter in a Django template html file.
    - see an example at /home/paAccountName/dj4e-samples/authz/templates/authz/main.html
    - more specifically from the example file:

{% if user.is\_authenticated %}

<p>Authenticated as

<pre>

Name: {{ user.get\_full\_name }}

Email: {{ user.email }}

Id: {{ user.id }}

</pre>

</p>

<p>You can <a href="{% url 'logout' %}?next={% url 'authz:open' %}">Logout</a></p>

{% else %}

<p>You are not logged in</p>

<p>You can <a href="{% url 'login' %}?next={{ request.path }}">Login</a> if you like.</p>

{% endif %}

* + - The “user” object is automatically injected into the template context by Django
    - “user” has the attributes: is\_authenticated (Boolean variable that is either true or false), get\_full\_name, email, and id (and other attributes)
      * In addition to accessing the “user” object in a template, it can also be accessed in the python (e.g., in views.py) – *see next section for how to access user object in python code*
    - {% url 'logout' %}?next={% url 'authz:open' %} is code that sends the user to the logout page, and, when they are done, returns the user to the …open' view in the 'authz…' application
    - The request.path is an important **pattern** to be familiar with:
      * <a href="{% url 'login' %}?next={{ request.path }}">Login</a>
        + {% url 'login' %} is a reverse Django view to the login page
        + ?next={{ request.path }} is code that returns the user to the present page after they are done at the 'login' page

**Using Django Login in Views**

To create a login page in Django that uses the built-in authentication, administration, error handling, etc., the developer must create a template – ideally, a template that matches the application style and possibly provides other helpful links and information.

* The developer must create a template that can be accessed as registration/login.html
  + No matter which application in Django project you're working in, the registration/login name is global.
* The login.html must:
  + handle errors and have some type of output:

{% if form.errors %}

<p>Your username and password didn't match. Please try again.</p>

{% endif %}

* The form within the login.html must:
  + 1. post to the login url:
    - form.as\_p is a context variable (?????)
  + 2. Place a submit button in the form
  + 3. Pass the ‘next’ variable back as a hidden field. A hidden form field is used in the pattern so that after the user submits this form, the user is directed to the next view.

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

{% csrf\_token %}

{{ form.as\_p }}

<input type="submit" class="btn btn-primary" value="Login" />

<input type="hidden" name="next" value="{{ next }}" />

</form>

{% endblock %}

* As stated earlier, the “user” object is automatically injected into the template context by Django, and in addition to accessing the “user” object in a template, it can also be accessed in the python (e.g., in views.py)
* The “user” object is part of the request object, and its attributes can be accessed by referring to the HttpRequest.user object in the python views.py code: is\_authenticated (Boolean variable that is either true or false), get\_full\_name, email, and id (and other attributes)

class DumpPython(View) :

def get(self, req):

resp = "<pre>\nUser Data in Python:\n\n"

resp += "Login url: " + reverse('login') + "\n"

resp += "Logout url: " + reverse('logout') + "\n\n"

if req.user.is\_authenticated:

resp += "User: " + req.user.username + "\n"

resp += "Email: " + req.user.email + "\n"

else:

resp += "User is not logged in\n"

resp += "\n"

resp += "</pre>\n"

resp += """<a href="/authz">Go back</a>"""

return HttpResponse(resp)

Add LoginRequiredMixin to views that can only be accessed by a logged in user.

* “Mixins are a form of multiple inheritance where behaviors and attributes of multiple parent classes can be combined…. Mixins are an excellent way of reusing code across multiple classes, but they come with some cost. The more your code is scattered among mixins, the harder it will be to read a child class and know what exactly it is doing, and the harder it will be to know which methods from which mixins to override if you are subclassing something that has a deep inheritance tree…. Note also that you can only inherit from one generic view - that is, only one parent class may inherit from View and the rest (if any) should be mixins.” (From djangoproject “Using mixins” [documentation](https://docs.djangoproject.com/en/3.2/topics/class-based-views/intro/)).
* From /dj43-samples/authz/views.py
  + class ManualProtect(View) is the long (manual) way of using request.user.is\_authenticated to determine whether to route the user to an authentication page or directly to the requested page
  + class ProtectView(LoginRequiredMixin, View) is the simpler **pattern** using the Django mixin, LoginRequiredMixin, to do the same thing
    - note that the app developer-defined class, ProtectView, inherits from both the Django base class, View, and mixin, LoginRequiredMixin

from django.shortcuts import render, redirect

from django.views import View

from django.urls import reverse

from django.utils.http import urlencode

class ManualProtect(View) :

def get(self, request):

if not request.user.is\_authenticated :

loginurl = reverse('login')+'?'+urlencode({'next': request.path})

return redirect(loginurl)

return render(request, 'authz/main.html')

from django.contrib.auth.mixins import LoginRequiredMixin

class ProtectView(LoginRequiredMixin, View) :

def get(self, request):

return render(request, 'authz/main.html')

* + - In ProtectView, the get method only runs if the LoginRequiredMixin determines that the user is, in fact, logged in.

**Django Forms**

**Using Django Forms Capabilities**

* Django forms are part of the view function in the model-view-controller structure – they interact with views.py and with templates, and they interact with the data model.
* By using Django forms, we are re-using hundreds of lines of code that produce user interfaces; validate user data entry into fields; receive the user provided data and interface with it to the data model in order to create, update, or delete table records; and respond to the user with success or error messages.
* Django forms live in a file called forms.py.
  + They resemble the models file in that they can specify fields with field types and can use validators to enforce rules about fields/attributes (e.g., field length).
  + However, the form validators can be even more complex than the model rules and types. In addition, you can write code with functions that parse an inbound, completed form, and have multiple validation criteria (e.g., a particular field must have two uppercase characters followed by three lowercase characters.
* In the next code snippets, we are going to create a view, in views.py, that uses a form, which is created in forms.py. This form is then inserted into a template through the use of the render method. The form is passed to the template as the context variable:

dj43-samples/form/forms.py

from django import forms

from django.core.exceptions import ValidationError

from django.core import validators

class BasicForm(forms.Form):

title = forms.CharField(validators=[

validators.MinLengthValidator(2, "...")])

mileage = forms.IntegerField()

purchase\_date = forms.DateField()

dj43-samples/form/views.py

from form.forms import BasicForm

class DumpPostView(View): # Reusable bit...

def post(self, request) :

dump = dumpdata('POST', request.POST)

ctx = {'title': 'request.POST', 'dump': dump}

return render(request, 'form/dump.html', ctx)

class SimpleCreate(DumpPostView):

def get(self, request) :

form = BasicForm()

ctx = {'form' : form}

return render(request, 'form/form.html', ctx)

dj43-samples/form/templates/form/form.html

<p>

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</table>

<input type="submit" value="Submit">

<input type="submit"

onclick="window.location='{% url 'form:main' %}' ; return false;"

value="Cancel">

</form>

</p>

* + In the above example,
    - the context, “ctx”, in the view, SimpleCreate, is the variable, “form”, which is an object of the BasicForm class that was defined in the forms.py file (1).
    - the context, “ctx”, along with the template, dj43-samples/form/templates/form/form.html, is rendered and returned to the client (2).
  + In the next code snippets, we are going to create a view combined with a form and populated data:

dj43-samples/form/views.py

from form.forms import BasicForm

class DumpPostView(View): # Reusable bit...

def post(self, request) :

dump = dumpdata('POST', request.POST)

ctx = {'title': 'request.POST', 'dump': dump}

return render(request, 'form/dump.html', ctx)

class SimpleUpdate(DumpPostView):

def get(self, request) :

old\_data = {

'title': 'SakaiCar',

'mileage' : 42,

'purchase\_date': '2018-08-14'

}

form = BasicForm(old\_data)

ctx = {'form' : form}

return render(request, 'form/form.html', ctx)

dj43-samples/form/templates/form/form.html

<p>

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</table>

<input type="submit" value="Submit">

<input type="submit"

onclick="window.location='{% url 'form:main' %}' ; return false;"

value="Cancel">

</form>

</p>

dj43-samples/form/forms.py

from django import forms

from django.core.exceptions import ValidationError

from django.core import validators

class BasicForm(forms.Form):

title = forms.CharField(validators=[

validators.MinLengthValidator(2, "...")])

mileage = forms.IntegerField()

purchase\_date = forms.DateField()

* + The form data, as “old\_data” (a dictionary variable), is passed to the BasicForm constructor (1), and the form is pre-populated.
    - A more common usage is to have the model pass the requested data to the form.

**Data Validation with Django Forms**

* Data validation occurs after the user has entered their data and hit the submit button. At that point, the data validation is the actions by the web server to examine the data and determine whether the user has submitted erroneous data. If the data is erroneous, then the server returns to the user a form with the appropriate message. If the data is valid, then the server stores the data and sends a success message.
* Here is the flow for a successful data validation using Django forms

Successful Data Validation Flow

dj43-samples/form/forms.py

class BasicForm(forms.Form):

title = forms.CharField(validators=[

validators.MinLengthValidator(2, "Please enter 2 or more characters")])

mileage = forms.IntegerField()

purchase\_date = forms.DateField()

dj43-samples/form/views.py

from django.urls import reverse

class Validate(DumpPostView):

def get(self, request) :

old\_data = {

'title': 'SakaiCar',

'mileage' : 42,

'purchase\_date': '2018-08-14'

}

form = BasicForm(initial=old\_data)

ctx = {'form' : form}

return render(request, 'form/form.html', ctx)

def post(self, request) :

form = BasicForm(request.POST)

if not form.is\_valid() :

ctx = {'form' : form}

return render(request, 'form/form.html', ctx)

# If there are no errors, we would save the data

x = reverse('form:success')

return redirect(x)

def success(request) :

return HttpResponse('Thank you!')

form/validate

Title:

Mileage:

Purchase Date:

Sakai Car

42

2018-08-14

Submit

Cancel

1. Initial GET to form/validate

2. User clicks “Submit” and POSTs form with valid data

form/success

Thank you!

3. Form is valid so …

4. the web server sends a 300 Redirect to the client for the form/success view, and …

5. and when the user requests (GETs) the form/success view,

6. The server sends a 200 HTTP Response with the simple message “Thank you!”

* Here is the flow if the user submitted data has errors. In this case the form is pre-populated with data where the “Title” is one character. The validation fails because the “Title” has to be at least two characters

**Please enter 2 or more characters**

Cancel

Submit

2018-08-14

42

S

form/validate

Title:

Mileage:

Purchase Date:

dj43-samples/form/forms.py

class BasicForm(forms.Form):

title = forms.CharField(validators=[

validators.MinLengthValidator(2, "Please enter 2 or more characters")])

mileage = forms.IntegerField()

purchase\_date = forms.DateField()

dj43-samples/form/views.py

from django.urls import reverse

class Validate(DumpPostView):

def get(self, request) :

old\_data = {

'title': 'S',

'mileage' : 42,

'purchase\_date': '2018-08-14'

}

form = BasicForm(initial=old\_data)

ctx = {'form' : form}

return render(request, 'form/form.html', ctx)

def post(self, request) :

form = BasicForm(request.POST)

if not form.is\_valid() :

ctx = {'form' : form}

return render(request, 'form/form.html', ctx)

# If there are no errors, we would save the data

x = reverse('form:success')

return redirect(x)

def success(request) :

return HttpResponse('Thank you!')

form/validate

Title:

Mileage:

Purchase Date:

S

42

2018-08-14

Submit

Cancel

1. Initial GET to form/validate

2. User clicks “Submit” and POSTs form with valid data

3. Form data is not valid (Title is only 1 character) so …

4. the web server renders and sends an HTTP response with a 200 status code. The template and the context, which is the form (that also includes any errors), are rendered for the HTTP response.

**Generic CreateView, UpdateView, DeleteView Django classes**

From Week #3 Walkthrough: Samples – Autos

Documentation can be found at:   
https://docs.djangoproject.com/en/3.2/ref/class-based-views/generic-editing/

from django.views.generic.edit import CreateView, UpdateView, DeleteView

* CreateView is a Django generic class that creates a view that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.
  + The following code snippet from a views.py file (for an app named “autos”) shows how to code a webpage that enables a user to create an automobile make (e.g. Subaru, GM, Toyota) record. The MakeCreate class inherits the Django CreateView generic class. It also inherits the LoginRequiredMixin class because we only allow authenticated, i.e., logged-in, users to create a new automobile make record.

class MakeCreate(LoginRequiredMixin, CreateView): # edited in week 3 assignment

model = Make

fields = '\_\_all\_\_'

success\_url = reverse\_lazy('autos:all')

* + **NOTE – important: The CreateView page displayed to a GET request uses a template\_name\_suffix of '\_form'.** For example, changing this attribute to '\_create\_form' for a view creating objects for the example Author model would cause the default template\_name to be 'myapp/author\_create\_form.html'.
  + In coordination with the MakeCreate class, we need to write a /autos/templates/autos/make\_form.html that looks like the following:

{% extends "base\_bootstrap.html" %}

{% block content %}

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</table>

<input type="submit" value="Submit">

<input type="submit" onclick="window.location='{% url 'autos:all' %}' ; return false;" value="Cancel">

</form>

{% endblock %}

* UpdateView is a Django generic class that creates a view that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes to the object. This uses a form automatically generated from the object’s model class (unless a form class is manually specified).
  + The following code snippet from a views.py file (for an app named “autos”) shows how to code a webpage that enables a user to update an automobile make (e.g. Subaru, GM, Toyota) record. The MakeUpdate class inherits the Django UpdateView generic class. It also inherits the LoginRequiredMixin class because we only allow authenticated, i.e., logged-in, users to update an existing automobile make record.

class MakeUpdate(LoginRequiredMixin, UpdateView):

model = Make

fields = '\_\_all\_\_'

success\_url = reverse\_lazy('autos:all')

* + **NOTE – important: The UpdateView page displayed to a GET request uses a template\_name\_suffix of '\_form'.** For example, changing this attribute to '\_update\_form' for a view updating objects for the example Author model would cause the default template\_name to be 'myapp/author\_update\_form.html'.
    - ***Where do we update the template\_name\_suffix? In the following example, it looks like template\_name\_suffix = \_list. How and where was this coded?***
  + In coordination with the MakeCreate class, we need to write a /autos/templates/autos/make\_list.html that looks like the following:

{% extends "base\_bootstrap.html" %}

{% block content %}

<h1>Make List</h1>

{% if make\_list %}

<ul>

{% for make in make\_list %}

<li>

{{ make.name }}

(<a href="{% url 'autos:make\_update' make.id %}">Update</a> |

<a href="{% url 'autos:make\_delete' make.id %}">Delete</a>)

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no makes in the library.</p>

{% endif %}

<p><a href="{% url 'autos:make\_create' %}">Add a make</a></p>

<p>

<a href="{% url 'autos:all' %}">Back to autos</a>

</p>

{% endblock %}

* DeleteView is a Django generic class that creates a view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is POST. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.
  + The following code snippet from a views.py file (for an app named “autos”) shows how to code a webpage that enables a user to delete an automobile make (e.g. Subaru, GM, Toyota) record. The MakeDelete class inherits the Django DeleteView generic class. It also inherits the LoginRequiredMixin class because we only allow authenticated, i.e., logged-in, users to delete an existing automobile make record.

class MakeDelete(LoginRequiredMixin, DeleteView):

model = Make

fields = '\_\_all\_\_'

success\_url = reverse\_lazy('autos:all')

* + **NOTE – important: The DeleteView page displayed to a GET request uses a template\_name\_suffix of '\_confirm\_delete'.** For example, changing this attribute to '\_check\_delete' for a view deleting objects for the example Author model would cause the default template\_name to be 'myapp/author\_check\_delete.html'.
  + In coordination with the MakeDelete class, we need to write a /autos/templates/autos/make\_confirm\_delete.html that looks like the following:

{% extends "base\_bootstrap.html" %}

{% block content %}

<h1>Delete Make</h1>

<p>Are you sure you want to delete the make: {{ make }}?</p>

<form action="" method="POST">

{% csrf\_token %}

<input type="submit" value="Yes, delete.">

<input type="submit" onclick="window.location='{% url 'autos:all' %}' ; return false;" value="Cancel">

</form>

{% endblock %}

**Assignment/Instructions Week #3: Autos App Create, Read, Update, and Delete (CRUD)**

**Making a New Application**

* Activate any virtual environment you need (if any) and go into your django\_projects folder and start a new application in your mysite project (this project already should have the 'hello' application from a previous assignment).
  + **workon django3 # as needed**
  + **cd ~/django\_projects/mysite**
  + **python manage.py startapp autos**

**Extending the home (i.e. main) page**

* Since we will build a number of applications in this project, we will use the home application to provide convenient urls to switch between applications.
* Validate that you have, or otherwise create, a file mysite/home/templates/home/main.html that has the text for the top-level page.
* Add a link to the "/autos" url in mysite/home/templates/home/main.html (and anything else the autograder needs):
  + **<li><a href="/autos">Autos CRUD</a>**
  + We’re adding a list because we will be adding more applications in the future.

**Building the Autos Application**

This task is adapts the code from: https://github.com/csev/dj4e-samples/tree/main/autos

* Create mysite/home/templates and mysite/home/templates/registration folders using mkdir and copy the (login.html) template from dj4e-samples into mysite/home/templates/registration/login.html.
* Copy the file from dj4e-samples/home/templates/base\_bootstrap.html into your mysite/home/templates - this will be used in your autos/templates and make our HTML look better by applying the Bootstrap and other styling libraries.
* Edit mysite/mysite/settings.py add the autos application to the list of INSTALLED\_APPS. You can follow the pattern of the HomeConfig line in that file.
* Edit mysite/mysite/urls.py and add the accounts/ path so you can use the Django built in login features. (Authentication Views). Also edit mysite/mysite/urls.py to route autos/ urls to autos/urls.py file.
  + **path('accounts/', include('django.contrib.auth.urls')), # Add**

**path('autos/', include('autos.urls')), # Add**

* Edit the autos/urls.py file to add routes for the list, edit, and delete pages for both autos and makes
* Edit the autos/views.py file to add views for the list, edit, and delete pages for both autos and makes. It will make things a lot simpler in the long run if you convert the Make views to the shorter form like the Auto views. (Example)
  + In your views.py file, you should not simply use the code for the Make views. You should rewrite the Make views using the same patterns as the Auto views. If you don't use the generic edit views on your Make views you will need to define the appropriate MakeForm in your forms.py just like in the sample code. The best approach is to build your views.py without using a forms.py - but you can do it either way.
* Run the **python manage.py check** until you see no errors
* Edit the autos/models.py file to add Auto and Makes models with a foreign key from Autos to Makes.
* Run the **python manage.py makemigrations** until it has no errors.
  + Sometimes when you make changes to models.py, the makemigrations will pick up on the changes and ask you for example if you want to rename a field.
  + Sometimes you make a change to your models.py and makemigrations gets stuck or lost. If migrations gets stuck, you might need to start with a fresh database.
* Run the **python manage.py migrate** to create the database.
* Edit autos/admin.py to add the Auto and Make models to the Django administration interface.
* Run the python manage.py check until you see no errors

**One to Many Data Models**

**One to Many Models Overview**

* model.py is where we define the structure of our database.
* We access the database through data models and, more specifically, use data model objects to reference the database and create, read, update, and delete table records.
* We register our data models in admin.py.
* It is important to have a well-defined database model so that your web application scales well and has good performance.
* The database model is represented in a picture with tables and links between tables.
* Database normalization is used to assess the design
  + The key design principle can be summed up by: “Don’t replicate string data”
  + Instead, define a string once in a table with an integer key for an index
  + Integers and integer keys are more efficient in storage usage, and easier to access and manipulate

**Removing Replication in One-to-Many Models**

* If you were start with a single flat table of all your application data, then you look for where you have vertical replication of data – i.e. repeating of data in the columns
  + This is where you will want to represent the replicated data once in a single table and provide links between tables
  + Removal of the replication of data results in a performance advantage, a storage advantage and it modifiability advantage (there's only one place where you need to go to update the data).
* In a picture of our database model, tables are represented by boxes (with the columns/attributes listed) and links are represented by lines
  + The lines are arrows and use the following legend at the start and end of each arrow to represent their cardinality
    - 1 One
    - 1..\* Many with a minimum of 1
    - 0..\* Many with a minimum of 0 (null value is OK)
  + Legends can vary with different modeling systems.

**Storing Primary and Foreign Keys in a Database**

* The primary key is a column that we add, and is the handle for the rows
  + The primary key is where we end a link arrow
  + We can auto-increment the primary key index to increase performance
* The foreign key is the column that we add to a table so that we can point to a row in another table.
  + The foreign key is where we will start a link arrow
  + By convention, the name of the foreign key column is the name of the table to which it point followed by “\_id”.

**Representing One-to-Many Models in Django**

* Here is an example of how you represent a One-to-Many model in Django
  + /dj4e-samples/bookone/models.py

from django.db import models

class Lang(models.Model):

name = models.CharField(max\_length=200)

class Book(models.Model):

title = models.CharField(max\_length=200)

isbn = models.CharField(max\_length=13)

lang = models.ForeignKey('Lang', on\_delete=models.SET\_NULL, null=True)

class Instance(models.Model):

book = models.ForeignKey('Book', on\_delete=models.CASCADE)

due\_back = models.DateField(null=True, blank=True)

* + Remember, these are the instructions for how to create a SQL database for your application – this is not the actual SQL syntax
  + Django will automatically include the instructions add the “id” field to a table
  + Django will automatically include the instructions for a foreign key so that the field uses the naming convention of the table to which you are linking followed by “\_id”
  + null=True allows the field to be empty
  + on\_delete=models.SET\_NULL indicates that if the foreign key value is deleted in the linked table, then set the foreign key in this table to NULL
  + on\_delete=models.CASCADE indicates that if the foreign key value is deleted in the linked table, then delete the rows in this table that had that value

**Using the Django Shell to Explore One-to-Many Models**

* To run the Django shell type

python manage.py shell

* + In creating the Django shell, this command causes the settings.py file to be read for all the listed applications, preloads the applications, and then gives you a shell.

**Loading One-to-Many Data using a Django Batch Script**

* Assume you have some source of data, perhaps in .CSV format, and you want to rad and put it into the database – but not manually by hand. here is how you write a script to load data into your Django models after you define those Django models.
* Running scripts is part of a feature of Django called the Django extensions.
  + Go into your virtual environment and type: pip3 install django-extensions
  + After you install it, you have to put a link into your settings.py, specifically
    - ProjectName/ProjectName/settings.py

…

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

…

# Extensions - installed with pip3 / requirements.txt

'django\_extensions',

'crispy\_forms',

…

# Sample Applications - don't copy

'hello.apps.HelloConfig',

'users.apps.UsersConfig',

'bookone.apps.BookoneConfig',

'bookmany.apps.BookmanyConfig',

…

'cats.apps.CatsConfig',

…

]

….

* + Create a “scripts” folder in your project directory
    - mkdir scripts
    - touch scripts/\_\_init \_\_.py
      * The latter command creates an empty file used by python for “housekeeping”
      * More documentation on \_\_init\_\_.py can be found with an online search but it is no longer needed in python 3.3.
    - In the /ProjectName/scripts folder, write your python script program
      * e.g., in /dj4e-samples/scripts/cats\_load.py

import csv # https://docs.python.org/3/library/csv.html

# https://django-extensions.readthedocs.io/en/latest/runscript.html

# python3 manage.py runscript cats\_load

from cats.models import Cat, Breed

def run():

fhand = open('cats/meow.csv')

reader = csv.reader(fhand)

next(reader) # Advance past the header

Cat.objects.all().delete()

Breed.objects.all().delete()

# Name,Breed,Weight

# Abby,Sphinx,6.4

# Annie,Burmese,7.6

# Ash,Manx,7.8

# Athena,Manx,8.9

# Baby,Tabby,6.9

for row in reader:

print(row)

b, created = Breed.objects.get\_or\_create(name=row[1])

c = Cat(nickname=row[0], breed=b, weight=row[2])

c.save()

* + - * next(reader) # Advance past the header
        + Moves the CSV reader past the header row of titles
      * b, created = Breed.objects.get\_or\_create(name=row[1])
        + In the current row, get the second field.
        + If there is not a record with this value in the table, then create the record
      * c = Cat(nickname=row[0], breed=b, weight=row[2])  
        c.save()
        + Create a Cat record using the first and third fields along with the breed object, b, that we just got, and then save it (commit it to the database)

**Assignment/Instructions Week #4: Cat database CRUD**

Here are some general specifications for this assignment:

* Use the Django-provided features for login and log out just as in the provided sample code.
* X1 - The auto-grader-required **meta** tag must be in the head area for all of the pages for this assignment.
* X2 - This can be added as a new application to your **mysite** project. You do not have to remove existing applications, simply add a new **cats** application. Activate any virtual environment you need (if any) and go into your `django\_projects` folder and start a new application in your `mysite` project (this project already should have 'hello' and 'autos' applications from previous assignments):

workon django3 # as needed

cd ~/django\_projects/mysite

python3 manage.py startapp cats

* X3 - Edit the **django\_projects/mysite/mysite/settings.py** to update the INSTALLED\_APPS:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'home.apps.HomeConfig',

'autos.apps.AutosConfig',

'cats.apps.CatsConfig', <---- Add this

]

* X4 - Edit the **cats/models.py** file to add Cat and Breed models as shown below with a foreign key from Cat to Breed.

from django.db import models

from django.core.validators import MinLengthValidator

class Breed(models.Model):

name = models.CharField(

max\_length=200,

validators=[MinLengthValidator(2, "Breed must be greater than 1 character")]

)

def \_\_str\_\_(self):

return self.name

class Cat(models.Model):

nickname = models.CharField(

max\_length=200,

validators=[MinLengthValidator(2, "Nickname must be greater than 1 character")]

)

weight = models.PositiveIntegerField()

foods = models.CharField(max\_length=300)

breed = models.ForeignKey('Breed', on\_delete=models.CASCADE, null=False)

def \_\_str\_\_(self):

return self.nickname

* X5 - Run the commands to perform the migrations.
* X6 - Add a link to **django\_projects/mysite/home/templates/home/main.html** that has the text for the top-level page.

<ul>

<li><a href="/autos">Autos CRUD</a>

<li><a href="/cats">Cats CRUD</a>

<ul>

* X7 - You should add a route to your **django\_projects/mysite/mysite/urls.py** as follows:

urlpatterns = [

path('', include('home.urls')),

path('admin/', admin.site.urls),

path('accounts/', include('django.contrib.auth.urls')),

path('autos/', include('autos.urls')),

path('cats/', include('cats.urls')),

]

* X8 - Create the **cats/urls.py** file to add routes for the list, edit, and delete pages for both cats and breeds. You do not need to change the **main** or **lookup** urls in **cats/urls.py** -

You should change the 'name=' values and class name on the paths from the sample application so you don't conflict with the 'autos' application:

urlpatterns = [

path('', views.CatList.as\_view(), name='all'),

path('main/create/', views.CatCreate.as\_view(), name='cat\_create'),

path('main/<int:pk>/update/', views.CatUpdate.as\_view(), name='cat\_update'),

...

]

* X9 - Edit the **cats/views.py** file to add/edit views for the list, edit, and delete pages for both cats and breeds.
* X10???? - Add the appropriate templates to **cats/templates** following the naming conventions for the templates.
* X11 - If you have not already done so, create the necessary templates in **home/templates/registration** to support the login / log out views.
* X12 - Edit **cats/admin.py** to add the Cat and Breed models to the Django administration interface.
* If you have not already done so, create a superuser so you can test the admin interface and log in to the application.

**Owned Rows**

**Owned Rows in Django - Overview**

Owned Rows: In the database model, we specify the owner of a row of data; and in the web application we create views so that only owners are offered the option to edit or delete. All users are allowed to view records. Furthermore, we have to provide security in the application to enforce the restriction that a non-owner cannot edit or delete a row.

* To create owned rows, we will use object-oriented inheritance.
  + Recall: ‘Subclasses’ are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.
* We will develop a class that inherits a django, generic view class.
  + E.g., Django generic ListView is the parent (or super) class
  + ArticleListView(ListView) is created by an application developer. It inherits the ListView class – ArticleListView is the child (or sub-class) to the ListView class. It “informs” ListView by adding the specification that “model = Article”.
  + OwnerArticleListView(ArticleListView) is created by the app developer and inherits from ArticleListView. OwnerArticleListView adds code (introduces behaviors) to restrict the editing and deleting of rows to the owner of a row.
  + Text

    Description automatically generated

**Owned Rows in Django – Generic Views Review**

When we inherit the Django Generic views, we can add code to our sub class that modifies existing or introduces new behaviors and attributes to the super class from which we inherit.

* From the Django documentation, we can get a flow chart of the methods used in Generic classes.
  + For example, the generic class ListView has the following flowchart of methods:
    1. setup()
    2. dispatch()
    3. http\_method\_not\_allowed()
    4. get\_template\_names()
    5. get\_queryset()
    6. get\_context\_object\_name()
    7. get\_context\_data()
    8. get()
    9. render\_to\_response()
  + These methods can be mapped to the following flow diagram:

Diagram

Description automatically generated with low confidence

* + WE can inherit all the methods (behaviors) in the ListView generic class and, additionally, write our own versions of the methods to modify their behaviors. This is called “extending” the behavior of the inherited class.
  + In modifying the behaviors of an inherited class, we can either:
    - completely replace a method in the parent class, or
    - augment a method in the parent class.
  + Specifically, to create an “owned” view, we will extend – i.e., write our own modified version of – get\_query\_set() and form\_valid().

**Owned Rows in Django – owner.py**

**Pattern**: Here are the steps for creating an **owned row** in Django:

* First, we create the model in models.py:
  + /projectName/appName/models.py, e.g.,
  + /dj4e-samples/myarts/models.py

from django.db import models

from django.core.validators import MinLengthValidator

from django.contrib.auth.models import User

from django.conf import settings

class Article(models.Model):

title = models.CharField(

max\_length=200,

validators=[MinLengthValidator(2, "Title must be greater than 2 characters")]

)

text = models.TextField()

owner = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

# Shows up in the admin list

def \_\_str\_\_(self):

return self.title

* + This table links to an external table that belongs to Django (i.e., table that Django defines for us)
    - This is specifically done in this line of the models.py file:  
      owner = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)
* Second, we define the views in views.py:
  + /projectName/appName/views.py, e.g.,
  + /dj4e-samples/myarts/views.py

from myarts.models import Article

from myarts.owner import OwnerListView, OwnerDetailView, OwnerCreateView, OwnerUpdateView, OwnerDeleteView

class ArticleListView(OwnerListView):

model = Article

# By convention:

# template\_name = "myarts/article\_list.html"

class ArticleDetailView(OwnerDetailView):

model = Article

class ArticleCreateView(OwnerCreateView):

model = Article

fields = ['title', 'text']

class ArticleUpdateView(OwnerUpdateView):

model = Article

fields = ['title', 'text']

class ArticleDeleteView(OwnerDeleteView):

model = Article

* + Note, in the next step we will create an “owner.py” file for our app that creates classes that inherit from the Django generic views (ListView, CreateView, etc.), so we need the following line in the views.py file
    - from myarts.owner import OwnerListView, OwnerDetailView, OwnerCreateView, OwnerUpdateView, OwnerDeleteView
  + Keep in mind that the view.py classes consider: 1. what is referenced from the database and 2. what appears on the screen.
  + In the view classes, we specify the model that we are referencing:
    - model = Article
  + We also need to specify how we want the form to appear, so we have a line specifying the displayed fields:
    - fields = ['title', 'text']
    - For the ArticleCreateView() and ArticleUpdateView() classes, we are specifying that the form to be displayed will have two fields: “title” and “text” – the “owner” field is purposefully omitted (along with the “created\_at” and “updated\_at” fields)
* Third, we extend the Django generic views in owner.py:
  + /projectName/appName/owner.py, e.g.,
  + /dj4e-samples/myarts/owner.py

from django.views.generic import CreateView, UpdateView, DeleteView, ListView, DetailView

from django.contrib.auth.mixins import LoginRequiredMixin

class OwnerListView(ListView):

"""

Sub-class the ListView to pass the request to the form.

"""

class OwnerDetailView(DetailView):

"""

Sub-class the DetailView to pass the request to the form.

"""

class OwnerCreateView(LoginRequiredMixin, CreateView):

"""

Sub-class of the CreateView to automatically pass the Request to the Form

and add the owner to the saved object.

"""

# Saves the form instance, sets the current object for the view, and redirects to get\_success\_url().

def form\_valid(self, form):

# print('form\_valid called')

object = form.save(commit=False)

object.owner = self.request.user

object.save()

return super(OwnerCreateView, self).form\_valid(form)

class OwnerUpdateView(LoginRequiredMixin, UpdateView):

"""

Sub-class the UpdateView to pass the request to the form and limit the

queryset to the requesting user.

"""

def get\_queryset(self):

# print('update get\_queryset called')

""" Limit a User to only modifying their own data. """

qs = super(OwnerUpdateView, self).get\_queryset()

return qs.filter(owner=self.request.user)

class OwnerDeleteView(LoginRequiredMixin, DeleteView):

"""

Sub-class the DeleteView to restrict a User from deleting other

user's data.

"""

def get\_queryset(self):

print('delete get\_queryset called')

qs = super(OwnerDeleteView, self).get\_queryset()

return qs.filter(owner=self.request.user)

* + OwnerUpdateView() Notes:
    - In this class we inherit the Django generic view, UpdateView(), and augment its get\_queryset() method
    - class OwnerUpdateView(LoginRequiredMixin, UpdateView) is a guardian pattern that ensures a user is logged into the application (by specifying LoginRequiredMixin) before we allow them to even attempt to edit a data set.
    - By using “super”, we are specifying that the variable “qs” get loaded using the parent version (i.e., Django generic version) of get\_queryset()
      * qs = super(OwnerUpdateView, self).get\_queryset()
    - We are returning “qs” after first filtering to ensure that the record(s) owner is the user – otherwise the user will get a 404. error
      * return qs.filter(owner=self.request.user)
    - These notes for OwnerUpdateView() also apply to OwnerDetailView() – the same pattern is followed in both classes.
  + OwnerCreateView() Notes
    - In this class we inherit the Django generic view, CreateView(), and augment its form\_valid() method
    - The generic form\_valid() method in the Django generic CreateView() looks at the data in a form, and makes sure that the data follows the rules for each field – e.g., field type, length or size, any developer defined criteria.
    - In order to create an owned record/row in the database, the OwnerCreateView() must note the user who is requesting the new record, and assign them to the owner field. We do this while the form is being validated – i.e., we modify the form\_valid() generic method as follows.
      * The line, object = form.save(commit=False) , saves the form with its content to the variable “object”
      * The line, object.owner = self.request.user , adds the “owner” field into the variable “object” and form (which both point to the same object?)
      * Then we save the form with object.save()
      * And finally we call on the parent/super version of form\_valid() using the parent CreateView() class and return:
        + return super(OwnerCreateView, self).form\_valid(form)
  + OwnerListView() and OwnerDetailView() Notes:
    - These are just placeholder classes – they don’t have any code (methods or attributes) because owned rows don’t apply to them. Anyone -- owners and non-owners -- is allowed to read a list of records or a detailed record.
* Fourth, we create the HTML templates for the extended generic views:
  + The HTML template files are created and stored in /projectName/appName/templates/appName using the following naming convention:
    - modelName\_confirm\_delete.html
      * e.g.: ad\_confirm\_delete.html
    - modelName\_detail.html
      * e.g.: ad\_detail.html
    - modelName\_form.html
      * e.g.: ad\_form.html
    - modelName\_list.html
      * e.g.: ad\_list.html

**Walkthrough: DJ4E Crispy Forms (crispy) Sample Code**

Crispy forms provide improved formatting and styling for a form within a Django webpage app.

Here are the steps to applying Crispy forms:

* First, when setting up your virtual python environment
  + Edit a requirements.txt file to include latest version of django-crispy-forms
  + The file will reside in the project folder, e.g. /dj4e-samples or /projName

# To activate this run

#

# pip install -r requirements.txt

# python -m django --version

#

# On a Macintosh this should be python3 and pip3

#

Django>=3.1.4

django-crispy-forms>=1.8.1

django-filter>=2.2.0

djangorestframework>=3.10.3

Markdown>=3.0.1

social-auth-app-django>=3.1.0

social-auth-core>=3.3.3

django-extensions>=2.2.5

django-taggit>=1.3.0

mysqlclient>=1.4.6

* + While in your Django virtual environment,
    - workon django3
  + and located at the project folder that contains requirements.txt,
    - cd /projName
  + run pip, python’s package manager, to read requirements.txt and get the latest version of django-crispy-forms
    - pip install -r requirements.txt
* Second, in settings.py
  + add an extension for crispy\_forms in the settings.py file

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

…

# Extensions - installed with pip3 / requirements.txt

'django\_extensions',

'crispy\_forms',

…

]

….

* Third, in views.py:
  + Make a class in the views.py file that extends View(),
  + Set the attribute variable template\_name = none -- you will set the template name in urls.py.

from django.views import View

…

class MyView(View):

template\_name = None # so we can override in urls.py

….

* Fourth, in urls.py:
  + Set template\_name to html templates in which Crispy will be applied.
  + Note: in the following example, the appName is “crispy” but it could be any name for an app – the use of “crispy” here is not a requirement of the pattern.

from django.urls import path

from . import views

from django.views.generic import TemplateView

# https://docs.djangoproject.com/en/3.0/topics/http/urls/

app\_name='crispy'

urlpatterns = [

path('', TemplateView.as\_view(template\_name='crispy/main.html'), name="main"),

path('boring', views.MyView.as\_view(template\_name='crispy/boring.html')),

path('awesome', views.MyView.as\_view(template\_name='crispy/awesome.html')),

]

* Fifth, in the form that use crispy.
  + we load the crispy\_form\_tags to activate Crispy
  + and then, we apply the Crispy filter to the table – this runs Crispy code to render the table with all the html/css styling
    - If we hadn’t applied crispy, then we would have had some boring styling using the Django Template Language line {{ form.as\_table }}

{% extends "base\_bootstrap.html" %}

{% block title %}DJ4E Crispy Forms{% endblock %}

{% block welcome %}

<h1>This is a crispy (a.k.a awesome) form</h1>

{% endblock %}

{% block content %}

<p>

<!-- see INSTALLED\_APPS in dj4e-samples/settings.py -->

{% load crispy\_forms\_tags %}

<form action="" method="post">

{% csrf\_token %}

{{ form|crispy }}

<input type="submit" value="Submit">

<input type="submit"

onclick="window.location='{% url 'crispy:main' %}'; return false;" value="Cancel">

</form>

</p>

{% endblock %}

**Walkthrough: DJ4E Bootstrap Menu (menu) Sample Code**

We can inherit or extend html template files using Django Template Language

* As the basis of all our HTML templates, we can start with /projectName/appName/home/template/base\_bootstrap.html file
  + We can place the file in any folder with any name, as long as we use the correct path and name to reference it
  + However, the naming convention above, can help organize the app’s file structure

Graphical user interface, text, application

Description automatically generated

App Tags: We can create our own app tags that are recognized by Django Template Language in our template files.

* app tags are tags that we, as developers, can create and then we can do things in it.
* Create an app tag python file and load your code – like displaying a gravatar on your webpage nav bar
* The file with your app tags can be named anything and placed anywhere, as long as you reference the correct path and name, however, the following convention works well:
  + /projectName /home/templatetags/app\_tags.py
* /dj4e\_samples/home/templatetags/app\_tags.py

from hashlib import md5

from django import template

# https://brobin.me/blog/2016/07/super-simple-django-gravatar/

# A "gravatar" is a globally recognized avatar that is based on email address

# People must register their email address and then upload a gravatar

# If an email address has no gravatar, a generic image is put in its place

# To use the gravatar filter in a template include

# {% load app\_tags %}

register = template.Library()

@register.filter(name='gravatar')

def gravatar(user, size=35):

email = str(user.email.strip().lower()).encode('utf-8')

email\_hash = md5(email).hexdigest()

url = "//www.gravatar.com/avatar/{0}?s={1}&d=identicon&r=PG"

return url.format(email\_hash, size)

* In our template file we must include:
  + {% load app\_tags %}
* And, then in the file we can reference the “gravatar” function that we wrote in the app\_tags.py file:

<ul class="nav navbar-nav navbar-right">

{% if user.is\_authenticated %}

<li class="dropdown">

<!-- gravatar depends on app\_tags being loaded -->

<!-- https://brobin.me/blog/2016/07/super-simple-django-gravatar/ -->

<a href="#" data-toggle="dropdown" class="dropdown-toggle">

<img style="width: 25px;" src="{{ user|gravatar:60 }}"/><b class="caret"></b></a>

<ul class="dropdown-menu">

<li><a href="{% url 'logout' %}?next={% url 'menu:main' %}">Logout</a></li>

</ul>

</li>

{% else %}

<li><a href="{% url 'login' %}?next={% url 'menu:main' %}">Login</a></li>

{% endif %}

</ul>

**Many to Many Models**

**Many to Many Overview**

Up until this point, we have been looking at data models where tables have a one-to-many relationship. Now we will look at database models which have tables that have a many-to-many relationship.

For example:

Timeline

Description automatically generated

**A Simple Many-To-Many Example in Django**

We cannot directly create a many-to-many relationship within a relational database, so it is instead portrayed as two one-to-many relationships.

* This is something that can be portrayed in a Django model
* Django allows us to create a “through” (AKA “join” or “junction” or “connection” or “bridge”) table between the two ends of the many-to-many table relationship.
* The “through” table has two outbound foreign keys with each pointing to one of the two ends of the many-to-many relationship.
* In the example below, “Authored” is the “through” table and it has a foreign key that points to the table “Book” (specifically to its auto-incrementing “id” attribute field) and a foreign key that points to the table “Author” (also to its auto-incrementing “id” attribute).

Graphical user interface, diagram

Description automatically generated

* Django creates a virtual field which it will compute upon request - in the example above, “authors” is the virtual attribute of the Book table and it will contain all the authors of a given book. Likewise, “books” is a virtual attribute of the Author table and contains all the books associated with one author.
* By default, Django will effectively “hide” the through table from the developer. However, you can override this and name and define the through table in your data model – which is what happens in the example below.
* Example: /dj4e-samples/bookmany/models.py

from django.db import models

class Book(models.Model):

title = models.CharField(max\_length=200)

authors = models.ManyToManyField('Author', through='Authored')

class Author(models.Model):

name = models.CharField(max\_length=200)

books = models.ManyToManyField('Book', through='Authored')

class Authored(models.Model):

book = models.ForeignKey(Book, on\_delete=models.CASCADE)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

* + The tables Book and Author have a many-to-many relationship …
  + … and, the model is built with Authored as the through table.
  + In the Book table, the attribute “authors” is a many-to-many link to the Author table and it is connected via the virtual through table Authored.
    - The attribute “authors” is not data that's stored in each row of the Book table. Rather, it is data which is derived for you if you request it when you retrieve a book – basically, asking ''Who are the authors of this book?''
    - Note that the Book table uses “authors” to point to the through table, and the Authored through table has a foreign key pointing right back to the Book table.
    - Also note that the “author” attribute in the Authored through table performs “on delete cascade” – this means Django will clean up this through table if rows, (e.g., an Author or a Book record) are deleted from the “outside to” table (i.e., an Author or Book table record).

**Many-To-Many Data Models for Courses and Membership**

One pattern of a “through” table is the “member” table. A member through table adds a role to each many to many relationship.

* In this example, educational systems are modeled where we have a many-to-many relationship between people and courses.
* A course can have many people and a person can be in many courses.
* However, people have roles within that many-to-many relationship: some people are student, others are teachers, and other people are administrators.

Diagram

Description automatically generated

* In Django, the /dj4e-samples/many/models.py file looks like:

from django.db import models

class Person(models.Model):

email = models.CharField(max\_length=128, unique=True)

name = models.CharField(max\_length=128, null=True)

courses = models.ManyToManyField('Course', through='Membership')

def \_\_str\_\_(self):

return self.email

class Course(models.Model):

title = models.CharField(max\_length=128, unique=True)

members = models.ManyToManyField('Person', through='Membership')

def \_\_str\_\_(self):

return self.title

class Membership(models.Model):

person = models.ForeignKey(Person, on\_delete=models.CASCADE)

course = models.ForeignKey(Course, on\_delete=models.CASCADE)

LEARNER = 1

IA = 1000

GSI = 2000

INSTRUCTOR = 5000

ADMIN = 10000

MEMBER\_CHOICES = (

( LEARNER, 'Learner'),

( IA, 'Instructional Assistant' ),

( GSI, 'Grad Student Instructor' ),

( INSTRUCTOR, 'Instructor' ),

( ADMIN, 'Administrator' ),

)

role = models.IntegerField(

choices=MEMBER\_CHOICES,

default=LEARNER,

)

def \_\_str\_\_(self):

return "Person "+ str(self.person.id) + " <--> Course " + str(self.course.id)

* The Course and Person tables have a many-to-many relationship, and their Django data model is like the previous example
* The Membership table is a variation on the earlier example, but it is an often-used pattern
  + In the Membership table, the developer created a set of integers and associated those integers with meaning so they could be used in the admin interface.
  + The “role” attribute in the Membership table is defined as an integer for efficient storage and fast retrieval performance.

role = models.IntegerField(

choices=MEMBER\_CHOICES,

default=LEARNER,

)

* + However, “role” has to be an integer from the list MEMBER\_CHOICES. If an integer number is not provided for the record, then its default value will be 1 (LEARNER=1).

**Building a Django Batch Script to Load Data from CSV**

Once the database model has been constructed and the migrations have been completed, the database is ready for use. Records can be created through the web application, the Django shell, or through a batch script.

The following python file is a script to read the rows of a CSV file and iteratively create records:

* A good file and folder convention is to load your script files in the folder: /projectName/scripts
* For a script to load, you must do the following steps when setting up your Django environment:
  1. Edit requirements.txt: call the Django extensions in /projectName/requirements.txt by including this line:
     + - django-extensions>=2.2.5
  2. Perform a pip install with requirements.txt – do this from a console while in your /projectName folder:
     + - pip install -r requirements.txt
  3. After you install it, edit the /projectName/projectName/settings.py folder and add django\_extensions into INSTALLED\_APPS:

INSTALLED\_APPS = [

…

# Extensions – installed with pip3 / requirements.txt

‘django\_extensions’,

‘crispy\_forms’,

‘rest\_framework’,

‘social\_django’,

‘taggit’,

‘home.apps.HomeConfig’,

…

]

* 1. Reload the web server on PythonAnywhere
  2. Make a /scripts folder from the /projectName folder in the console and create the file /projectName/scripts/\_ Init\_.py -- this file designates that the folder which it resides in contains modules that are suitable for importing:
     + - mkdir scripts
       - touch scripts/\_\_init\_\_.py
* /dj4e-samples/scripts/many\_load.py

import csv # https://docs.python.org/3/library/csv.html

# https://django-extensions.readthedocs.io/en/latest/runscript.html

# python3 manage.py runscript many\_load

from many.models import Person, Course, Membership

def run():

fhand = open('many/load.csv')

reader = csv.reader(fhand)

next(reader) # Advance past the header

Person.objects.all().delete()

Course.objects.all().delete()

Membership.objects.all().delete()

# Format

# email,role,course

# jane@tsugi.org,I,Python

# ed@tsugi.org,L,Python

for row in reader:

print(row)

p, created = Person.objects.get\_or\_create(email=row[0])

c, created = Course.objects.get\_or\_create(title=row[2])

r = Membership.LEARNER

if row[1] == 'I':

r = Membership.INSTRUCTOR

m = Membership(role=r, person=p, course=c)

m.save()

* This is the CSV file contents that is being read and inserted into the database using the script:

email,role,course

jane@tsugi.org,I,Python

ed@tsugi.org,L,Python

sue@tsugi.org,L,Python

ed@tsugi.org,I,Django

sue@tsugi.org,L,Django

ed@tsugi.org,I,SQL

[jane@tsugi.org,L,SQL](mailto:jane@tsugi.org,L,SQL)