**Creating a Django project**

**Getting started**

1. **Open pycharm and create a project folder**
   1. This will contain only the venv to begin
2. **Install django**
   1. pip install django
3. **create a django project**
   1. use the django-admin CLI to create project
   2. run django-admin –help to view available commands
   3. run django-admin startproject <project\_name> to create project
   4. This will create a directory containing the main django project files under the name of <project\_name> that was used
4. **manage.py**
   1. Now that the project was created, a manage.py file was also created and placed in the main directory (outside of the django project directory). This file will be used from now on to run commands instead of django-admin. The manage.py file runs commands specifically for the project it was built for.
5. **Initialize a git repo**
   1. git init
   2. create a .gitignore file in the root folder of the project (where manage.py is)
   3. to get a default .gitignore file, go to gitignore.io and type in the framework/language
   4. git add . (dot is all) && git commit -m “Initial commit”
   5. (dot is to add all, -m is to be followed by notes about the commit taking place)
   6. After entering command from step 4, all files (excluding files in .gitignore) are pushed to the local repo.
   7. Add a remote repo
      1. git remote add origin <https://github.com/beezkneez406/btre.git>
   8. git branch -M main
   9. Push to remote repo
      1. git push -u origin main
      2. then add user name for github (jbies)
      3. then type in PAT (personal access token)
      4. ghp\_PJJpHPYyaA0ZkCwRpVjPGwulXUVGFZ1c8LY1 (for btre project, no expiration)
6. **Run the development server**
   1. To run the dev server, type python manage.py runserver into the terminal
   2. With the server now running we can go to localhost:8000 to see our website
7. **Overview of django project files**
   1. **settings.py**
      1. BASE\_DIR, base directory of project
      2. SECURITY\_KEY, used for production deployment. This needs to be hidden eventually.
      3. DEBUG, set to True for development but needs to be set to False once deployed
      4. ALLOWED\_HOSTS, list of host domains that the website can serve
      5. INSTALLED\_APPS, default apps are here to begin but the apps we build or need for other functionality will be added here
      6. MIDDLEWARE, list of middleware django uses. A lot of what is listed here is for security.
      7. ROOT\_URLCONF, string that represents the full import path to url file
      8. TEMPLATES, used to generate html for application. Tells django where to look for templates and which options to use.
      9. WSGI\_APPLICATION, path that the wsgi object that djangos built in servers use
      10. DATABASES, where we configure our database parameters. Default is sqlite.
      11. AUTH\_PASSWORD\_VALIDATORS, set rules for the types of passwords used
      12. STATIC\_URL, path used for static files (CSS, javascript, etc.)
   2. **urls.py**
      1. routing file
      2. urlpatterns list is where the path and view methods link should go
   3. **wsgi.py**
      1. Stands for web server gateway interface
      2. how web servers interact with web applications.
      3. deals with hosting site so other people can access it

**Creating an app**

1. **Create a django app**
   1. To create an app use the manage.py script
   2. python manage.py startapp <app\_name>
   3. This creates a new directory named <app\_name>, this will include:
      1. a directory called migrations for any migrations that are created
      2. an *\_\_init\_\_.py,* so the app is treated as a package
      3. *admin.py* for adding pieces of the app to the admin area
      4. *apps.py,* inculdes the class of the configuration for the app
      5. *models.py,* this is where the models for the app is created. Models are a description of the data to be stored (more or less).
      6. *tests.py,* used for running tests
      7. *views.py,* this is where we will create methods/functions that will be linked to urls
2. **Add the app to settings**
   1. navigate to settings.py under the project folder.
   2. Under the INSTALLED\_APPS list, add a new entry ‘<app\_name>.apps.<app\_name>Config’
   3. The new entry consists of the app name, then apps, then the class of the configuration from the apps.py file within the app we are adding.
3. **Create urls.py file in the app**
   1. Inside the app we need to create a urls.py file. When we modify the urls.py file under the project folder, we will include this file to point to the urls for this app.
   2. Make necessary imports
      1. from django.urls import path
      2. from . Import views
   3. Add urlpatterns list
      1. urlpatterns = []
   4. inside the urlpatterns, we need to add some paths. Using the path method that is imported add some urls to the list.
      1. Path(‘<path>’, views.<name\_of\_method\_in\_views>, name=’<name\_to\_access\_this\_path’>)
4. **Create a method inside views.py to link to**
   1. Add a function inside the views.py file. The name of the function needs to match the parameter views.<name\_of\_method\_in\_views> that is passed into the path function in urls.py.
5. **Add urls.py from app to urls.py inside project folder**
   1. make sure the “include” method is imported from django.urls
   2. Add a path to the urlpatterns
      1. path(‘<url>’, include(‘<app\_name>.urls’))
      2. leave <url> inside path function blank if you want the home page to be [www.website.com/](http://www.website.com/)<home>

**Templates and Base layout**

1. **Tell django where to look for templates**
   1. First step is to let django know where it can find our templates.
   2. Inside the settings.py file under the project folder, navigate to the TEMPLATES dictionary.
   3. Under DIRS we need to add the path to the directory where we will keep templates.
   4. Templates will be kept in the main folder or root foler (where manage.py is).
   5. To point to that location add the following to the DIRS list
      1. os.path.join(BASE\_DIR, ‘templates’)
      2. BASE\_DIR is defined at the top of settings.py as the path to the root directory
      3. may need to import os before step 1
2. **Add the templates directory**
   1. Inside the root folder, add a directory named ‘templates’
   2. inside the templates directory it is a good idea to create subdirectories with the names of each app to store templates separately.
3. **Add templates (html files) to these directories** 
   1. Once the templates have been created or added to the templates directory/s, we need to add the paths to the urlpatterns in the url.pys file for that specific app.
   2. Add index.html, and about.html templates
4. **Edit the urls.py file**
   1. Add the path() functions for the templates
      1. path(‘’, views.index, name=’index’)
      2. path(‘about’, views.about, name=’about’)
5. **Add view method for templates**
   1. Any newly created templates that were added to the urls.py file need view methods to go with them.
   2. Add/edit the views.py file to include view methods for the new templates
   3. Rather than return Httpresponse, we want to return the ‘render’ method
   4. render takes in two parameters (for now), request and the location of the template to render
6. **Create a base.html file**
   1. In the root of the templates folder, create base.html
   2. This will be “extended” on all other templates
   3. Inside the body tag of the base.html file, add jinja block {% block content %} {% endblock %}
      1. If pycharm project wasn’t created using django flag, the settings of the project will need to be modified to enable django support and also use jinja2 as the template language.
   4. Add the {% extends ‘base.html’ %} line to the top of all templates.

**Static files**

1. **Create static folder**
   1. Inside the django project folder (PycharmProjectFolder > django project) create a new folder named ‘static’
   2. Add subdirectories to the static folder for any static content (css, js, fonts, img)
2. **Add STATIC\_ROOT, and STATIC\_DIRS**
   1. navigate to settings.py inside the django project folder and add the following above/below STATIC\_URL.
      1. STATIC\_ROOT = os.path.join(BASE\_DIR, ‘static’)
      2. STATICFILES\_DIRS = [os.path.join(BASE\_DIR, ‘<project\_dir\_name>/static’)
3. **Run collectstatic**
   1. using manage.py run the collectstatic command, this will find any static files and create a static folder wherever we said was the STATIC\_ROOT directory.
   2. python manage.py collectstatic
4. **Add static to .gitignore**
   1. We do not want to push static files to our repo
   2. open .gitignore and add “/static”

**Bootstrap**

1. **Open the bootstrap theme in a separate text editor (sublime text)**
   1. Opening in a separate text editor isn’t crucial, but it helps to not get confused between project html and bootstrap theme html.
2. **Start by copying css and script links**
   1. These can be found in the header tag and the bottom of the body tag
   2. open the base.html file and copy these over
3. **Change href for links**
   1. the href’s need to be changed to point to the correct location to find the css and scripts
   2. We need to load the static files by adding some jinja
   3. At the very top of the base.html file, insert {% load static %}
   4. Change all href attribute for each we need to also add some jinja
      1. href=”{% static ‘<location\_of\_css\_or\_script>’ %}”
      2. ex → href=”{% static ‘css/all.css’ %}”
4. **Add the top bar and nav bar**
   1. From the bootstrap theme index.html, copy the nav bar and top bar (any other html pieces that will be shared across entire site) and paste them into base.html.
5. **Create partials**
   1. Inside the templates folder, create a directory named partials
   2. create separate html files for anything we want as partials (this will declutter the base.html file)
   3. the convention for creating partial html files is to precede the name with an underscore \_
   4. Create the following
      1. \_topbar.html
      2. \_navbar.html
      3. \_footer.html
   5. From base.html, copy out each partial into their own files.
   6. Using jinja, bring in each partials
      1. {% include ‘partials/\_topbar.html’ %}
      2. {% include ‘partials/\_navbar.html’ %}
      3. {% include ‘partials/\_footer.html’ %}

**Setting up the markup (html)**

1. **Next step is adding html for each page**
2. **We do not want to edit base html, we need to edit individual templates**
   1. open index.html (home page) and add any needed html for the block content.
   2. Open any other html pages and copy any html needed for the block content
3. **Setting up page links**
   1. inside the href anchor tag href attribute <a href=””> we need to insert some jinja2 syntax to point to the templates we are wanting to link
   2. inside the href use {% url ‘<name\_of\_template>’ %}
   3. The <name\_of\_template> comes from whatever name is specified in the urls.py file for the specific app.
   4. In the urls.py file we used the path() method where we specify the name using the name=’’ parameter.
4. **Dynamic markup content**
   1. to display certain html based on conditions we can use if statements using jinja2
   2. For this example we are changing the highlighting of items (active not active attribute) in the navbar based on the request path.
   3. Edit the navbar html partial by editing the list items where the anchor tags are. We want certain list items to use the “active” class when that page is the current page and the other items to be grayed out when they are not the current page.
   4. Edit the list items as follows

{% if request.path == ‘<url path>’ %}

class=”nav-item active mr-3”

{% else %}

class=”nav-item mr-3”

{% endif %}

1. <url\_path> should be whatever the path is set to in urls.py
2. For this example index (‘/’), about (‘/about’), etc.

**Adding more apps**

1. **Add another app** 
   1. Adding more apps is just as easy as creating the first one. Just use the manage.py file and the ‘startapp’ command.
   2. If you are going to use this app for just setting up a model skip the next steps regarding adding urls/views
2. **create a urls.py file inside the new app directory**
   1. Now that the new app is created and we have a urls.py file, lets register it with django.
   2. First lets add any urls we are going to need to the newly created urls.py file inside the new apps directory.
      1. Make imports
      2. *from django.urls import path*
      3. *from . Import views*
   3. For standard urls we just use text, if we want to use a parameter for the url we can pass that in with ‘<dtype:parameter\_name>’ for this example we are using an ID field so it would look like the example below.
      1. *Path(‘<int:listing:id>’, views.listing, name=’listing’)*
3. **Next we need to edit the views.py file**
   1. Add a method for each url we created in the previous steps inside urls.py
      1. *def <method\_name>(request):*
         1. *return render( request, ‘template.html’, context)*
4. **Now we need to register this with the django project**
5. **Open urls.py under main project directory**
   1. Add the urls created from the apps urls.py file using the path function
      1. *path(‘<app\_path>/’, include(‘<app\_name>.urls’))*
6. **Open the settings.py file in main project**
   1. in the INSTALLED\_APPS list, add the Config class from the apps.py file in the newly created apps directory.
      1. *INSTALLED\_APPS = [ ‘<app\_name>.apps.<app\_name>Config’ ]*
7. **Create the html files for each of the newly added urls/views**

**Installing and setting up Postgres DB**

**Ubuntu**

**Installing postgresql**

1. Open a terminal and install postgresql using the apt repo
   1. sudo apt install postgresql
2. Configure postgres, setup root user
   1. Open the psql CLI by switching to the postgres user
      1. sudo -u postgres psql
   2. change the postgres user passwords
      1. ALTER USER postgres PASSWORD ‘<new\_password>’;
         1. (#Mustangs33)
      2. We can also use \password <user> to alter passwords
      3. exit postgres (\q)
   3. log back into the psql CLI
      1. psql -U postgres -h localhost
      2. enter password we just changed in previous steps
3. We now have the default setup for postgresql and have changed the password for the postgres user. We could now add more users/databases.

**Installing PgAdmin4**

1. Go to the pgadmin4 website.
2. Click on the downloads link
3. Select APT (for debian based)
4. Open a terminal and follow the instructions on the downloads page for how to setup the repository
5. Once the repo is setup, install pgadmin desktop and web using the following.
   1. Sudo apt install pgadmin4 (both web and desktop)
   2. Sudo apt install pgadmin4-desktop (desktop only)
   3. Sudo apt install pgadmin4-web (web only)
6. configure the pgadmin user account
   1. sudo /usr/pgadmin4/bin/setup-web.sh
7. Add server
   1. Create a name for the server
   2. Click the connection tab
   3. enter host (localhost)
   4. leave port at default (5432)
   5. enter the postgres username and password
8. Once the databases are listed, right click on database and select properties
   1. Go to the security tab and grant all priviledges

**Create new database**

1. login to postgres shell
   1. psql -U postgres -h localhost
2. Create a new database
   1. CREATE DATABASE <db\_name> OWNER <psql\_user>;

**Django postgresql setup and Migrate**

**Setting up django for postgres**

1. First we need to install the pyscopg2 python package so we can work with postgres from python
   1. pip install pyscopg2
   2. pip install pyscopg2-binary
2. Open settings.py under the main project directory
3. Scroll down to DATABASES dictionary
4. The dictionary looks like this to begin

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.sqlite3',  
 'NAME': BASE\_DIR / 'db.sqlite3',  
 }  
}

1. Edit the dictionary as follows for a postgres database configuration

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.postgresql',  
 'NAME': 'btredb',  
 'USER': 'postgres',  
 'PASSWORD': '#Mustangs33',  
 'HOST': 'localhost'  
 }  
}

**Running initial migrations**

1. Every time we have run our django server we have gotten error messages about unapplied migrations. Now we are going to run those migrations.
2. From the terminal run
   1. python manage.py migrate
3. If no error messages occur, the previous steps for setting up django for postgres were successful.

**Planning schemas**

Before building models using python it is wise to plan the schema (structure of data) of the database

1. Create a schemas.txt file and begin by listing out all tables/models and the fields that will go with them.

**Creating models**

1. **open the models.py file for the app we are creating a model for**
2. **create a new class with the singular version of the name of the app. For this example we are creating a model for listings on a real estate website.**
   1. class Listing(models.Model):
3. **Inside this class we will add all of the properties or fields for this table (model)**
4. **Once the models are setup with their respective properties/fields we need to migrate the changes into the database itself.**
   1. python manage.py makemigrations
   2. the makemigrations command creates a file for each model and places them into the migrations folder under the specific app. That file will then be used to migrate changes into the database.
   3. **NOTE**: if image fields were used in the model/s there may be a need to install a python package named pillow to handle the images.
5. **If we want to create migration files for a single app we need to omit that argument into the makemigrations commands**
   1. python manage.py makemigrations <name\_of\_app\_to\_migrate>
6. **After the migration files have been created we have the option to view the SQL query that will take place (if we want to).**
   1. python manage.py sqlmigrate <name\_of\_app> <migration\_file\_version>
7. **make the migrations**
   1. python manage.py migrate

**Django admin area**

1. **Now that we have a database created and registered, and we have models created and migrated into the database, we need to setup the django admin area so we can perform CRUD operations (CREATE, READ, UPDATE, DELETE)**
2. **to get to the admin area, with the server running, open the url below.**
   1. Localhost:8000/admin
3. **Now we need to create a superuser for the admin area**
   1. python manage.py createsuperuser
   2. Create a username for the superuser, this example we are going to use ‘admin’
   3. Add an email address for the superuser, this example we will use a fake email admin@btre.com
   4. Create a password for the superuser, for this example we will use ‘#Mustangs33’
4. **Next we need to register our models from our apps with the admin area, this way we can modify the models/tables from the admin area.**
   1. Open the admin.py file under the specific app
   2. add import of model
      1. from .models import <model\_name>
   3. Add the following line to the admin.py file
      1. admin.site.register(<model\_name\_we\_imported>)

**Adding a media folder**

1. **open settings.py under the main project directory**
2. **under the static folder settings we need to add some lines of code to define the media folder**
3. **add a comment # Media folder settings**
4. **Add media root path**
   1. MEDIA\_ROOT = os.path.join(BASE\_DIR, ‘media’)
5. **Add media URL**
   1. MEDIA\_URL = ‘/media/’
6. **Add media URL to urls.py under main project directory**
   1. open urls.py
   2. add imports
      1. from django.conf import settings
      2. from django.conf.urls.static import static
   3. after urlpatterns list add the following
      1. + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

**Customizing the admin area**

1. **To customize the template (HTML) for the admin area we need to add an admin template folder**
   1. under the templates folder create a directory named “admin”
   2. inside the admin templates folder create an html file named “base\_site.html”
   3. Extend the admin template at the top of the newly created base\_site.html file
      1. {% extends ‘admin/base.html’ %}
   4. To bring in a logo from the static files we need to load static as well
      1. {% load static %}
   5. To remove the default ‘Django Admin’ text from the top bar we need to utilize the branding block and place whatever we like in there.
      1. {% block branding %}
   6. To modify the css of the admin area we need to utilize the extrastyle block
      1. {% block extrastyle %}
      2. Inside the extrastyle block, add a link tag and change the href to {% static ‘css/admin.css’ %}
      3. We will need to add/create the admin.css file.
         1. **NOTE**: Using the google chrome inspector tool is a great way to pick through the html and find the correct selectors for the css styles
2. **Customize the data displayed in the admin area**
   1. to customize how the models/data is displayed in the admin area we need to edit some python code.
   2. Open the admin.py file under the specific app.
   3. Add a class that is named the same as the app and followed by admin (class <AppNameAdmin>)
   4. This class inherits from admin.ModelAdmin. The below example uses the listings app.
      1. class ListingAdmin(admin.ModelAdmin):
   5. Now from within the newly created class we can add some properties to display certain pieces of data, filter data, etc.
   6. Using the list\_display property we can list out everything we want displayed.
   7. Using list\_display\_links we can edit which fields are clickable links to edit that row.
   8. Using list\_filter we can add a filter object to sort data
   9. Using list\_editable we can make certain list items editable from the main page rather than needing to enter each individual item to edit.
   10. We can add the search\_fields property to list any fields we would like to be searchable.
   11. Using list\_per\_page we can manage how many rows are displayed at once.
   12. **NOTE**: Do not forget to pass in the class we created to the line that registers our model with the admin area.
       1. Admin.site.register(<Model\_Name>, <ModelAdmin\_Class>)

**Pull data from models into front end**

1. **Open the views.py file for the specific application**
2. **We now need to import the model for this application**
   1. from .models import <Model\_Name>
3. **To pass data into a template we need to add a context parameter to the render function.**
   1. Return rend(request, <template.html>, context)
   2. The cleaner way of doing this is to create a context variable that contains a dictionary of the data we want to pass to the template. Rather than inserting the dictionary into the function parameters.
      1. Context = { ‘data\_identifier’: ‘data’}
   3. We can pass in our model data by doing the following
      1. model\_data = <Model>.objects.all()
      2. we can order the data by doing the following
         1. model\_data = <Model>.objects.order\_by(‘db\_field’)
            1. to make the order descending preceed the db\_field with ‘-’
      3. we can also filter the data using .filter()
         1. model\_data = <Model>.objects.filter(db\_field=<True/value>)
      4. Then pass the model\_data into the context dict
         1. context = { ‘model\_data’: model\_data }
4. **Now from the template side we can bring in that data using {{ }} jinja tags.**
   1. {{ model\_data.data }}
   2. For images we need the url of the photo from the media folder
      1. {{ model\_data.model\_mage.url }}
   3. We can also use the pipe character to filter the data
      1. {{ model\_data.model\_datetime\_field | timesince }}
   4. For more template filter/format options like above, we need to add the humanize app which is not installed by default.
      1. Go to settings.py
      2. Under INSTALLED\_APPS add the following
         1. ‘django.contrib.humanize’
      3. To then use humanize in the template we need to load it. Add the following line underneath the extends tag.
         1. {% load humanize %}
5. **For multiple instances we want to list we can use a for loop to run through the model rows**
   1. {% for model in models %}
      1. {{ do stuff }}
   2. {% endfor %}
6. **Utilize the ID field from the model inside the url**
   1. Inside the urls.py file, setup the path as follows.
      1. Path(‘<data\_type:model\_id>’ views.<view\_method>, name=’<url\_reference\_name>’)
   2. Then from the template side, setup the href as follows
      1. href=”{% url ‘<url\_reference\_name>’ <model\_data.model\_id> %}”
   3. For step 2 to work properly the model\_data would need passed to the template using the context parameter in the view function.
   4. The view function for this url also needs the model\_id field passed into it for the url to work properly.

**Pagination**

1. To create pagination on a webpage we can use a paginator in our view method
2. Start by importing necessary objects
   1. from django.core.paginator import EmptyPage, PageNotAnInteger, Paginator
3. Inside the view function add the paginator
   1. paginator = Pagnitor(<model>, <models\_per\_page>)
   2. ex. paginator = Paginator(listings, 3)
4. Next we need to create a page variable
   1. page = request.GET.get(‘page’)
5. and finally create a paged\_<models> variable
   1. paged\_<model> = paginator.get\_page(page)
6. We can then pass in the paged\_<model> variable into our context dictionary to pass this to the template.
   1. Context = { ‘<model>’: <paged\_model> }
7. Next we need to include the pagination into our template. We will use a mix of the django documentation for pagination and the bootstrap pagination.
8. Use an if statement to check if the paged\_model variable has other pages
   1. {% if model.has\_other\_pages %}
9. Next we need to check if there are previous page numbers, if so we will display left facing arrows using the laquo;
   1. {% if paged\_model.has\_previous %}
      1. <li class=”page-item”>
         1. <a href=”?page= {{ paged\_models.previous\_page\_number }}” class=”page-link”>&laquo;</a>
      2. </li>
   2. {% else %}
      1. <li class=”page-item disabled”>
         1. <a class=”page-link”>&laquo;</a>
      2. </li>
   3. {% endif %}
10. Now we want to loop through the page range and list out each page number. If the current page matches the iteration value, make that list item active, otherwise just print out the number and create a link to the previous or next page.
    1. {% for i in paged\_model.paginator.page\_range %}
       1. {% if paged\_model.number == i %}
          1. <li class=”page-item active”>
             1. <a class=”page-link> {{ i }} </a>
          2. </li>
       2. {% else %}
          1. <li class=”page-item”>
             1. <a href=”?page= {{ i }} class=”page-link”>{{ i }}</a>
          2. {% endif %}
11. Next we need to do the same thing we did in step 9 but for the next page instead of previous page. Copy the code from step 9 and adjust the previous to next and the &laquo; to &raquo;

**Get single object from db to display in templates**

1. If we want to display just one object from the db in a template we can use the following.
   1. <model> = get\_object\_or\_404(<Model>, pk=<model\_id\_field>)
      1. we need to import the get\_object\_or\_404 from django.shortcuts
      2. we also need to import our model from <package>.models import Model
      3. the pk parameter is passed in through the view function via url. The below example uses a single listing url parameter from the udemy course.
         1. Def listing(request, listing\_id):
            1. listing = get\_object\_or\_404(Listing, pk=listing\_id)
            2. Where ‘Listing’ is the model itself being passed in and ‘listing\_id’ is the id field from the model.

**Iterating through dictionary in templates**

1. We can pass anything we want/need into the context dictionary from the views.py file. If we pass an iterable object to the template we can use loops to unpack them.
2. To unpack an iterable object from a dictionary do the following.
   1. {% for key, value in <dictionary>.items %}
      1. <do stuff>
   2. {% endfor %}

**Search form filtering (Querysets)**

1. Using a search form from the template we can sort/filter through our database and grab only objects that match our search.
2. Create a queryset\_list variable in the view method. Open views.py and under the specific view method add the following.
   1. Def view\_method(request):
      1. queryset\_list = <Model>.objects.order\_by(‘<model\_field>’)
3. Now we need to add some filters to the queryset\_list. The example below uses a parameter ‘keywords’ from the url to search for keywords in the description field of the model.
   1. If ‘keywords’ in request.GET:
      1. keywords = request.GET[‘keywords’]
         1. if keywords: # check that keywords isnt an empty string
            1. queryset\_list = queryset\_list.filter(description\_\_icontains=keywords)
4. The above example from step 3 looks for something in the description field that contains anything from keywords. This example below displays how to search for an exact match.
   1. If ‘city’ in request.GET:
      1. city = request.GET[‘city’]
         1. if city: # check that keywords isnt an empty string
            1. queryset\_list = queryset\_list.filter(city\_\_iexact=city)
5. **NOTE:** The difference between icontains → contains or iexact → exact is the one with I preceeding is case insensitive where without the I it is case sensitive.
6. We can also search a field in a model using conditional operators. An example would be searching for home prices below or above a certain range.
   1. # price  
      *if* 'price' *in request*.GET:  
       price = *request*.GET['price']  
       *if* price:  
       queryset\_list = queryset\_list.filter(price\_\_lte=price)
   2. The above example searches prices of homes that are less than or equal to the price that is passed in from the request.
      1. To search a field with conditional operators (<, >, <=, >=)
      2. <, (price\_\_lt=price)
      3. <=, (price\_\_lte=price)
      4. >, (price\_\_gt=price)
      5. >=, (price\_\_gte=price)

**Retaining form input values (Section 47 from udemy btre course)**

1. To retain values that are passed into a form we need to pass the values from the request.GET back to the template.
2. Open views.py for the specific app and add the key/value pair *‘values’: request.GET* into the context dictionary.
3. Now from the template side we need to see if that information is there and use it for the value/selected form fields.
   1. Value=”{{ values.city }}”
   2. {% if key == values.state %}
      1. selected
   3. {% endif %}

**User authentication or login**

1. We are going to begin by creating a new app and naming it ‘accounts’
2. We do not need to create any models for user accounts. Django has already done that for us. When we did our initial migrations django created the necessary tables.
3. Create the html files needed. For this example we will create login, register, and dashboard html files. (for logging out we will create a view function but no html file. We will use the redirect function for that instead. When a user clicks logout they will be redirected to the home page and a message will appear informing them they have been logged out.)
4. Before continuing, add the urls.py file to the new app “accounts”, add the AccountsConfig to the INSTALLED\_APPS in settings.py, add the view methods to views.py, and finally add the urlpatterns from urls.py to the main project urls.py file.
5. In the register and login templates we need to modify the form action and method. Change the form tags for each template as follows.
   1. <form action=”{% url ‘register’ %}” method=”POST”>
   2. <form action=”{% url ‘login’ %}” method=”POST”>
6. **NOTE**: an important step to take when using the POST method with forms is to include a csrf token to prevent cross site forgery requests. To do this add jinja tags directly below the form tag s from above as follows.
   1. {% csrf\_token %}
7. For the register view method we need to add some logic. We want to check if the request method is POST or GET.
   1. If request.method == ‘POST’:
      1. #register the user then redirect
   2. else:
      1. return render(‘template.html’)
8. Next we need to add messages functionality. Django includes a messages framework by default. It is already in the INCLUDED\_APPS but we need to configure a few things. Open up the settings.py file and do the following.
   1. At the top add the following import
      1. from django.contrib.messages import constants as messages
   2. At the bottom of settings.py add the following.
      1. MESSAGE\_TAGS = { messages.ERROR: ‘danger’, }
9. Once we have our messages configured in the settings.py file, we are going to create an alerts partial inside ‘templates/partials’. Name the file \_alerts.html.
10. Inside the \_alerts partial we want to do the following
    1. check if there are any messages {% if messages %}
    2. loop through all messages (if more than one) {% for message in messages %}
    3. Use bootstrap message block inside of the for loop.
    4. Check the message level with {% if message.level == DEFAULT\_MESSAGE\_LEVELS.ERROR %}
    5. If it is an error print out error or use an else statement to print out the title using {{ message.tags | title }}
    6. After that print out the message itself {{ message }}
11. Now from our view method we need to add the message functionality
    1. add the messages object by importing
       1. from django.contrib import messages
    2. then inside the view method we need to add something like the following
       1. messages.error(request, ‘<message to be displayed>’)
12. For the message to actually be displayed we need to add the \_alerts.html partial to the template where it will be displayed.
    1. {% include ‘partials/\_alerts.html’ %}

**User registration**

1. to begin we want to take any form data that is submitted from the template via POST request and place them into variables. This will be done from within the view method.
   1. If request.method == ‘POST’:
      1. first\_name = *request*.POST['first\_name']  
         last\_name = *request*.POST['last\_name']  
         username = *request*.POST['username']  
         email = *request*.POST['email']  
         password = *request*.POST['password']  
         password2 = *request*.POST['password2']
2. Next we want to make sure that the passwords match and that the user does not already exist
   1. *if* password == password2:  
       # Check that username is unique  
       *if* User.objects.filter(username=username).exists():  
       messages.error(*request*, 'The username already exists!')  
       *return* redirect('register')  
       *else*:  
       # Check that the email is not used  
       *if* User.objects.filter(email=email).exists():  
       messages.error(*request*, 'That email account is in use!')  
       *return* redirect('register')  
       *else*:  
       # Everything checks out  
       # now create a user  
       user = User.objects.create\_user(first\_name=first\_name, last\_name=last\_name,  
       username=username, email=email, password=password)  
       user.save()  
       messages.success(*request*, 'You are now registered and may log in')  
       *return* redirect('login')  
      *else*:  
       # if passwords do not match, throw an error  
       # and redirect back to the register page  
       messages.error(*request*, 'Passwords do not match!')  
       *return* redirect('register')
3. In the above code snippet we did the following
   1. check to make sure passwords match, if they don’t throw an error and redirect back to register page.
   2. If they matched, we then check that the username is unique, if it isn’t, again we throw an error and redirect back to the register page.
   3. If the username checked out, we then want to make sure the email doesn’t already exist. If it does, again we throw an error and redirect back to the register page.
   4. If the passwords matched, the username was unique, and the email did not already exist, we create a user variable/object
      1. user = User.create\_user(<fields>)
   5. We then save the user and display a success message, then redirect them to the login page.

**User Login**

1. Inside the view method for login we need to first make sure it is a POST request we are receiving.
   1. If request.method == ‘POST’:
2. From there we need to create variables for username and login.
   1. Username = request.POST[‘username’]
3. after we gather the information from the template for username and password we need to authenticate the user with django.
   1. User = auth.authenticate(username=username, password=password)
   2. NOTE: auth is imported from django.contrib
4. Next thing is make sure the user variable/object is actually there
   1. if user is not None:
5. If the user checks out, lets log them in.
   1. auth.login(request, user)
   2. messages.success(request, ‘You successfully logged in!’)
   3. return redirect(‘<template\_after\_login>’)
6. If the user does not checkout, throw and error and redirect back to login
   1. messages.error(request, ‘credentials are no good, check username or password’)
   2. return redirect(‘login’)
7. Full code example below (view method for logging in)
   1. *def* login(*request*):  
       *if request*.method == 'POST':  
       # Get login information from the login form  
       username = *request*.POST['username']  
       password = *request*.POST['password']  
        
       user = auth.authenticate(username=username, password=password)  
        
       *if* user *is not None*:  
       # user exists attempt to login  
       auth.login(*request*, user)  
       messages.success(*request*, 'You are now logged in!')  
       *return* redirect('dashboard')  
       *else*:  
       # user was unable to login  
       messages.error(*request*, 'Login unsuccessful, check username/password.')  
       *return* redirect('login')

**User Logout**

1. If the user is logged in successfully from the last steps (logging in) we redirect the user to the dashboard.
2. Once at the dashboard we need to change the navbar so login/register no longer appear. With a user logged in we want to replace login/register with dashboard/logout links.
   1. Edit the \_navbar.html partial to accomplish this.
   2. We can use {% if user.is\_authenticated %} to check if a user is logged in to determine what should appear in the navbar. If a user is logged in show the dashboard/logout links, if no user is logged in show the login/register links.
3. Once the links are setup in the html files we now need to receive the logout information.
   1. We can do this using a basic form in the navbar under the logout link.
   2. We need to use some javascript in the href for the anchor tag in the logout link.
      1. Href=”javascript:{document.getElementById(‘logout’).submit()}”
   3. Then we place a hidden form along side an info tag. Full html snippet below.
      1. {% if user.is\_authenticated %}  
          <li  
          {% if 'dashboard' in request.path %}  
          class**="nav-item active mr-3"**  
         {% else %}  
          class**="nav-item mr-3"**  
         {% endif %}  
          >  
          <a class**="nav-link"** href**="**{% url 'dashboard' %}**"**>**Welcome** {{ user.username }} **(Dashboard)**</a>  
          </li>  
          <li>  
          <a class**="nav-link"** href**="javascript:{document.getElementById('logout').submit()}"**>  
          <i class**="fas fa-sign-out-alt"**></i> **Logout**  
         </a>  
          <form id**="logout"** action**="**{% url 'logout' %}**"** method**="POST"**>  
          {% csrf\_token %}  
          <input type**="hidden"**>  
          </form>  
          </li>  
         {% else %}
   4. Once that is done we can setup or view method to accept the POST request and log the user out.
      1. *def* logout(*request*):  
          *if request*.method == 'POST':  
          auth.logout(*request*)  
          messages.success(*request*, 'You are now logged out.')  
          *return* redirect('index')  
          *return* redirect('index')