Managing

Starting: > python manage.py startapp

Starting shell > python manage.py shell

Updating db

>python manage.py makemigrations

>python manage.py migrate

Administration

Basics

Create Superuser to use admin tools

>python manage.py createsuperuser

Import and register models in admin.py to have control over them

Use admin tools from /admin

Logins

Pretty much just a login form with username and password

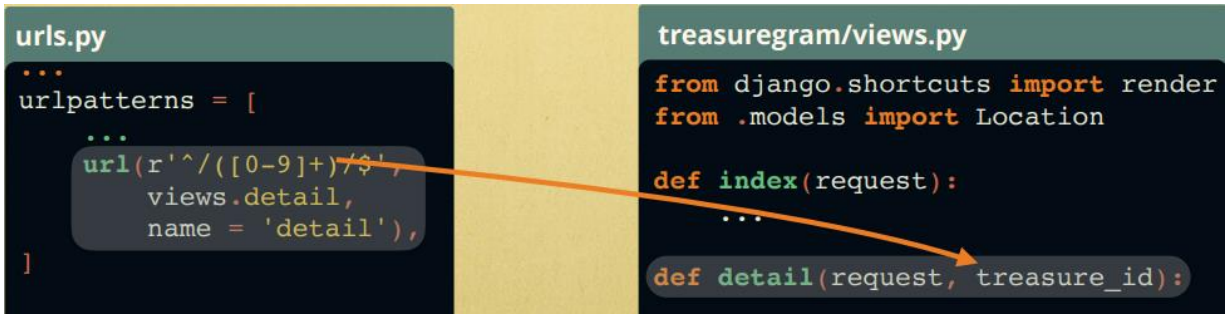
Settings

Register each app in settings.py's INSTALLED_APPS

URL Dispatcher

Basic Usage
imports: mainapp / views, django.conf.urls / url, django.contrib / admin
urlpatterns = […]
url(r'^<regex>', views.<view function>)
]

Passing Arguments



Routing to main_app
route from the base directory URLs to the project by inserting
from django.conf.urls import include, url
…
url(r'^', include('main_app.urls')):

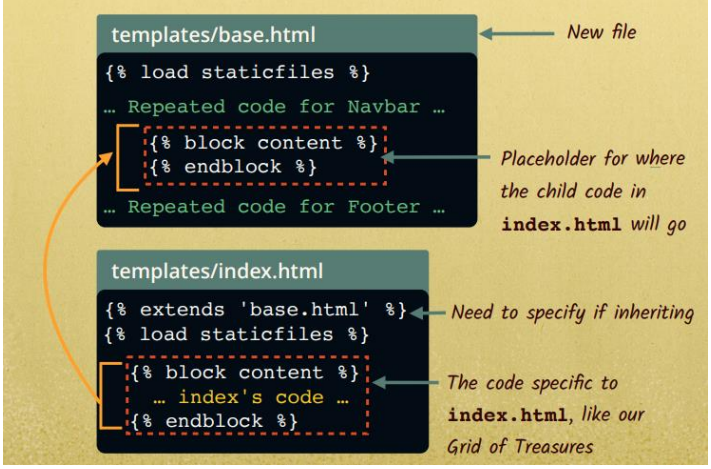
Tempaltes

Basic Usage

Mostly HTML with a template language https://docs.djangoproject.com/en/1.11/ref/templates/builtins/
Called by render in Views, which usually gives it a context dictionary
Useful stuff:
cycle: alternates between arguments, inserted anywhere
evalutate dictionary: {{ key }}

Inheritance

Use a base for everything not in the block, then define the block in another file



Views

Basic Example Usage

```
def index(request):
        treasures = Treasure.objects.all()
        form = TreasureForm()
        return render(request, 'index.html',
                      {'treasures': treasures, 'form':form})
```

needs import of .models / *

Has template to render, context, call model-defined Treasure, and form (which is is forms.py)
Can also receive parameters from URLs (see URLs)

Parameters

Always takes a request as a parameter, which may contain information?

HTTPResponse

Using django.http / HttpResponse, send HTTP
> return HttpResponse(…)

Template Renders

Can return renders of Templates

Post Requests

Usually for use in conjunction with AJAX

Models

Simple models

models.py

```
from django.db import models

class Treasure(models.Model):
    name = models.CharField(max_length=100)
    value = models.DecimalField(max_digits=10,
                                decimal_places=2)
    material = models.CharField(max_length=100)
```

Fields can be Char-, Integer-, Float-, Decimal-

More defined at https://docs.djangoproject.com/en/1.11/ref/models/fields/

Nice to have a toString

> def _ _Str_ _(self): return self.name

All will have primary key (“id”) with an integer when it was created

ANY CHANGES TO MODEL YOU HAVE TO MIGRATE so it goes thru the Python layer to the SQL

>python manage.py makemigrations

>python manage.py migrate

Default data

Just have an object further down that calls the constructors or uses other objects!

Static Resources

Basics

Put a static dir in your main_app dir for CSS, Javascript, and Static Images

A subdirectory for images

Load in templates with {% load staticfiles %} at very top

Download libraries like bootstrap and jquery

Forms

Basics

Define a format of data

Give it a url and receiving view so it can be inputted

Then in the front end, give it to a template and its view render function so the user can use it

-Defining the data: The form sets the parameters for the whole process

main_app/forms.py

```
from django import forms

class TreasureForm(forms.Form):
    name = forms.CharField(label='Name', max_length=100)
    value = forms.DecimalField(label='Value', max_digits=10,
                               decimal_places=2)
    location = forms.CharField(label='Location', max_length=100)
    img_url = forms.CharField(label='Image URL', max_length=255)
```

Notice the fields are almost identical to the model fields, but we can also define the label.

-The input end: Receiving and writing the data

Assign it a URL and a receiving function in the view

main_app/urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.index, name = 'index'),
    url(r'^([0-9]+)/$', views.detail, name = 'detail'),
    url(r'^post_url/$', views.post_treasure, name='post_treasure'),
]
```

The regex that will match **localhost/post_url**

Our view that will handle the posted data

main_app/views.py

```
...
from .forms import TreasureForm
...
def post_treasure(request):
    form = TreasureForm(request.POST)
    if form.is_valid():
        treasure = Treasure(name = form.cleaned_data['name'],
                            value = form.cleaned_data['value'],
                            material = form.cleaned_data['material'],
                            location = form.cleaned_data['location'],
                            img_url = form.cleaned_data['img_url'])
        treasure.save()
    return HttpResponseRedirect('/')
```

We'll save our new treasure.

Then we'll redirect back to the homepage.

-The front end:

Pass the format to a template via the view that renders that page

Need a csrf_token for security purposes

Can do {{form.as_p}} for paragraph format or {{form}} for inline

main_app/views.py

```
...
from .forms import TreasureForm
...
def index(request):
        treasures = Treasure.objects.all()
        form = TreasureForm()
        return render(request, 'index.html',
                      {'treasures': treasures, 'form':form})
```

main_app/templates/index.html

```
<main ...>
  ...
  <form action="post_url/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
  </form>
</main>
```

Widgets

For more complex types of data input (date/password/etc) use a widget

forms.py

> forms.

Tests

Javascript

AJAX

Use a views function in conjunction with your AJAX function

Best practice is to put js in main_app/static/js, along with jquery-3.0.0.min.js

Link them in templates

<script src="{% static 'js/jquery-3.0.0.min.js' %}"></script>

IMPORTANT: for post, we need to copy/past ajax code getCookie and csrfSafeMethod

AJAX: add a link to the object when it's clicked, getting the ID from the local part of the DOM

main_app/static/js/main.js

```
...
$('button').on('click', function(event){
  event.preventDefault();
  var element = $(this);
  $.ajax({
    url : '/like_treasure/',
    type : 'POST',
    data : { treasure_id : element.attr("data-id")},

    success : function(response){
      element.html(' ' + response);
    }
  });
});
```

Register the URL, with this object ID passed via the AJAX function

```
...
url(r'^like_treasure/$', views.like_treasure, name='like_treasure' ),
...
```

Views: Receive the data and save

main_app/views.py

```
...
def like_treasure(request):
    treasure_id = request.POST.get('treasure_id', None)

    likes = 0
    if (treasure_id):
        treasure = Treasure.objects.get(id=int(treasure_id))
        if treasure:
            likes = treasure.likes + 1
            treasure.likes += likes
            treasure.save()

    return HttpResponse(likes)
```

User Authentication

Logging In

Pretty much just a post form (must define LoginForm)

But we can combine the POST and render view functions with some special code

main_app/views.py

```
from .forms import TreasureForm, LoginForm
from django.contrib.auth import authenticate, login, logout
...
def login_view(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            u = form.cleaned_data['username']
            p = form.cleaned_data['password']
            user = authenticate(username = u, password = p)
            if user is not None:
                if user.is_active:
                    login(request, user)
                    return HttpResponseRedirect('/')
    else:
        form = LoginForm()
        return render(request, 'login.html',
                      {'form': form})
```

If we have an active user, we can use the built-in **login** *function to log in.*

Then we'll redirect to the homepage.

… and logging out

```
    url(r'^login/$', views.login_view, name='login'),
    url(r'^logout/$', views.logout_view, name='logout'),
```

```
def logout_view(request):
    logout(request)
    return HttpResponseRedirect('/')
```

Template language has a handy " {% if user.is_authenticated %}

Python Language Features

Queries of Objects defined in the Model

Get all instances of object

> <Object Name>.objects.all()

Subset matching field

> <Object Name>.objects.filter(<field> = <value>)

Get by unique value