django_basics

February 15, 2018

1 Django Tutorial

by Kenneth Love

Django is web framework designed to be out of the box ready, however it does have a lot of great third party libraries to choose from.

What are CMS's?

- CMS stands for Content Management Systems.
- Their development was driven by the need publish newspaper content online.
- Django was orginally developed to be a CMS at a newspaper.

Django is used by:

- Instagram
- Mozilla
- Pinterest

Installing Django `pip install django`

Checking Version of Django `$ python -m django --version`

Other Django Tutorials Django Basic Poll Application

1.1 Starting the Project

`django-admin.py startproject learning_site` or `django-admin startproject learning_site` will get the project started (Creates a project skeleton essentially). We won't need django-admin again for the rest of our work in this course.

What is manage.py?

- Used to run commands for a project, its kind of like Django Admin.
  - Could be used to create a new application
  - Could be used to migrate a database
  - etc.

1.1.1 Inside of learning_site (or the name you specified after `startporject`) Directory:

What is settings.py?

- Just holds all the settings for your applicaton.

What is urls.py?

- Will contain all the base URLs for the application you build.

What is wsgi.py?

- Controls how your application will be served on the internet, such as wether your using AWS or Heroku.

1.2 Running the Server

- `python manage.py runserver 0.0.0.0:8000` will run the server for Workspaces. On your own computer, you probably don't need the `0.0.0.0:8000` part.
  - You'll probably get something like the following error message when run this command for the first time:
    - `You have 13 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions. Run 'python manage.py migrate' to apply them.`
    - This is normal, and wil go away once you run `python manage.py migrate`
  - When this command is run for the first time a sqlite3 database 'db.sqlite3' is created at the root. You can configure another db to be setup by default like MySQL or PostGreSQL. You'll definitly want to switch to over to one of those (or any other pro db system) once you take your project live.
- `python manage.py migrate` will apply all pending migrations from all apps. More on apps later.

What are migrations? Migrations are a way of moving a database from one design, a specific set of tables and columns, to a new one. Migrations are reversible, too. The fact that they can be done backwards and forwards is what gives them their name.

1.3 Hello World in Django

Django is a MVC, or Model View Controller Framework.

- Django refers to templates as templates and functions that return rendered templates as views.

`HttpReponse` is the class that represents an HTTP response back to the client. The way we used it in this video is the absolute simplest way of generating a response. It's not the most useful tool in the shed, but it's the gateway to all of the other HTTP tools we have.

More information about `HttpResponse` and about Django's view functions.

`url()` is a function that constructs a special object that Django uses to join URLs to view functions.

- Django's URLs are created with Regular Expressions.

Instructions:

- Create a views.py file inside the stub directory (a dir inside the root dir).
- Type the following in views.py:

```
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse('Hello World')
```

The `request` argument is a variable name convention for incoming http requests. It is a requirement for all views.

- In urls.py type the following imports below the Django imports:

```
from . import views
```

The dot just means import such & such from the current directory.

- Add the following to the `urlpatterns` list:

```
url(r'^$', views.hello_world)
```

The `url` function can actually take up to five arguments:

```
* Regular Expression pattern: `r' '`
* View (function) to send the request to: `<views_py_file>.<view_name>`
* kwargs for the view
* name for the route
* A prefix
```

- Run `python manage.py runserver`, and check localhost at port 8000 for the 'Hello World' text.

1.4 Our First App

Django projects contain multiple Django apps. Each app generally encompasses a specific area of functionality.

`python manage.py startapp` creates the skeleton of an app including the views, models, and tests files. * Creates a few `__init__.py` files, which mark a directory a python module FYI. They are what allows us to import things from inside those directories.

`INSTALLED_APPS` is a list of all apps that Django should consider installed and active for the current project. These apps will be used to find templates, tests, models, and migrations.

`TIME_ZONE` is the setting for what time zone your server is running in. The docs explain a little more and there's a list of time zones you can use.

Instructions

- From the root of your project run `python manage.py startapp courses`
- Inside `settings.py` add the string 'courses' to the end of the INSTALLED_APPS list.

1.5 What are models?

Think of model classes as a way for python to make a database table, and the attributes of that class are the table's columns.

`django.db.models` has most of the model functionality you'll use to create models and their fields.

- `DateTimeField` holds datetime objects.
- `CharField` holds strings.
- `TextField` holds an unspecified amount of text.
- More Django model field types.

`python manage.py makemigrations [app]` will make the migrations for a specific app.

`python manage.py migrate [app]` will run the pending migrations for a specific app. If you leave off the app name, any pending migrations for any apps will be run.

Here is the list of `manage.py` commands.

Instructions:

- Open models.py in Course stub directory.
- Write the following a few lines down from the import statements.

```
class Course(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=255)
    description = models.TextField()
```

auto_now_add is essentially using datetime.datetime.now, but with the current time zone set in settings.py

- Since we've created a new Model class we need to update our table. We need to specify what has changed by running the following command:

```
$ python manage.py makemigrations course
```

- That command should return with the following:

```
Migrations for 'courses':
  courses/migrations/0001_initial.py
    - Create model Course
```

If you examine '0001_initial.py', this is what migration syntax looks like for Django.

- To put the migration into practice we run:

```
$ python manage.py migrate courses
```

- The following should be returned:

```
Operations to perform:
  Apply all migrations: courses
Running migrations:
  Applying courses.0001_initial... OK
```

1.6 Adding Instances (Adding records to the database)

`python manage.py shell` opens a Python shell with Django's configuration already loaded.

`Model.save()` will save an in-memory instance of a model to the database.

`Model.create()` will save an in-memory instance of a model to the database and return the newly-created object.

`Model.filter(attribute=value)` will get a QuerySet of all instances of the Model that match the attribute values. You can change these values with other comparisons, too, like gte or in. Find out more here

Instructions:

- Run `python manage.py shell`
- In the shell type `from courses.models import Course`
  - When we want to query Django's ORM, we have to use the model name (`courses`) and its objects attribute (`models`).
  - Objects points to what's called a model manager, which is a class that controls access to the model's instances and other things.
- In the shell type `Course.objects.all()`
  - Should return an empty QuerySet, `<QuerySet []>`, as there is nothing currently in the database.
- To create a Course Object to be put in the database type the following:

```
>>> c = Course()
>>> c.title = "Python Basics"
>>> c.description = "Learn the basics of Python"
```

- To save the Course Object to the database:

```
>>> c.save()
```

- Lets check out our new entry with:

```
>>> Course.objects.all()
[<Course: Course object>]
```

- That was a lot of steps to create a single database entry. Lets try doing this in one line:

```
>>> Course(title="Python Collections", description="Learn about list, dict, and tuple").save()
```

- Now we have two objects in our QuerySet:

```
>>> Course.objects.all()
<QuerySet [<Course: Course object>, <Course: Course object>]>
```

- Another way to do a one liner for creating database entries:

```
>>> Course.objects.create(title="Object-Oriented-Python", description="Learn about Python's cl
<Course: Course object>
```

Did you notice this time we got an Course Object back? This will be helpful in the future.

- Exit the shell and jump into 'models.py'.
- We need to fix how are courses are turned into strings.
  - Write the following beneath the Course Class's attributes:

`python  def __str__(self):  return self.title`

- Now re-enter the shell and use `Course.objects.all()`:

```
>>> Course.objects.all()
<QuerySet [<Course: Python Basics>, <Course: Python Collections>, <Course: Object-Oriented-Pyt
```

1.7 First App View

`include()` allows you to include a list of URLs from another module. It will accept the variable name or a string with a dotted path to the default urlpatterns variable.

```
include('<dir_name>.<module_name>')
```

If you don't have `include()` in your `urls.py` (more recent versions of Django have removed it), add it to the `import` line that imports `url`. That line will now look like from `django.conf.urls import url, include`.

This is the Comprehensions Workshop that's mentioned in the video. Comprehensions are a great way of doing quick work with iterables.

Instructions: Creating a course list view.

- Open views.py inside the courses folder.
- Add the following import: `from django.http import HttpResponse`
- Below the django imports add: `from .models import Course`
  - The dot before models is referencing the current directory.
- Add the following view function:

```
def course_list(request):
    courses = Course.objects.all()
    output = ', '.join(courses)
    return HttpResponse(output)
```

- Create a urls.py within the courses directory.
- Add the following to urls.py:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.course_list)
]
```

When we add a new view in views.py, we need to add it's route to the urlpatterns List in urls.py

- Now we need to tell Django about our new view and its associated url. We do this by adding the following urlpattern, `url(r'^courses/', include('courses.urls'))` to the urls.py inside the project folder, for us thats 'learning_site'.

```
from django.conf.urls import url, include
from django.contrib import admin

from . import views

urlpatterns = [
    url(r'^courses/', include('courses.urls')),
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.hello_world)
]
```

Be sure that `include` has been imported from `django.conf.urls`

- Open views.py inside of courses, and make the following change to the output variable:

```
output = ', '.join([str(course) for course in courses])
```

We need to make this change because only string objects can be rendered in the browser. (This is true of both Django and Flask)

- Now run the server on your local machine with `python manage.py runserver`.
- In the address bar add '/courses/' and press enter.
  - You should now see the list of courses we created earlier.

1.8 Django's Admin

`python manage.py createsuperuser` will create a new superuser, or a user that's allowed to log into the admin area with all permissions.

`admin.site.register(Model)` will register a model with the default admin site, which allows you to edit instances of that model in the admin.

Instructions:

- Be sure server is running.
- In the address bar, add '/admin/' to the root of the site.
  - You should see Django's auto-generated admin panel appear.
- Shutdown the server (Ctrl-C)
- To create an account we use the `python manage.py createsuperuser` command.
  - Run through the prompts to provide a username, email, and password.
  - my credentials were: user: admin, email: example@example.com, password: kennethlove.

- Turn the server back on, and type in the credentials you just provided.
  - Feel free to explore the admin pages
- Now lets add our Courses app to the admin. First open up the 'admin.py' inside of the courses directory.
- Add the following:

```
from .models import Course

admin.site.register(Course)
```

- Save 'admin.py'
- Go to admin Home page, and you should see a Courses Panel.
- Click the Add Button
  - Fill in the title with `Python Testing`and the description with `Learn to test your Python applications with unittests and doctests.`

1.9 Templates

App-specific templates are best kept in a structure like `app_name/templates/app_name` because Django looks in app directories for a directory named `templates` and makes those templates automatically available.

`{{ and }}` are used to mark a variable you want printed out.

`{% and %}` mark template tags, or special bits of Python that Django's template engine knows how to run. Unlike Jinja2 templates, you can't just run arbitrary Python in a template.

`render()` turns a request object, a template, and an optional context dictionary into a generated string. More about render.

Instructions:

- Create a new directory named templates in your 'courses' app folder.
- Within that directory create a directory that is the same as your app, in our case that's 'courses'.
  - Why we do this: So that we have our app specific templates inside this namespaced directory, courses. And then if we need to let people override them, they just make their own template directory named courses. Or, if we want to have templates that are for multiple sections, we could name them different names, whatever.
- Within that directory create a file named course_list.html, open it, and type the following:

```
{% for course in courses %}
<h2>{{ course.title }}</h2>
{{ course.description }}
{% endfor %}
```

- Open views.py, and change the following:

```
def course_list(request):
    courses = Course.objects.all()
    return render(request, 'courses/course_list.html', {'courses': courses})
```

*The arguments given to the render function are as follows: request object (the object passed to the function), the template to be rendered, and finally the context dictionary which is how were referencing our courses variable in our template.

- Run the server and check out '/courses/'.
  - You should see a list of courses with thier descriptions.
- Lets create a template for our index page (home page) which exists outside our app. To do this start by creating a directory named 'templates' inside the project's root directory, the first 'learning_site' directory in our case.
- Inside of the 'learning_site' child directory of the root 'learning_site' directory, open settings.py
  - A few notes about the TEMPLATES LIST:
    - Each item TEMPLATES is a dictionary, and each of those dictionaries describes one way of rendering templates.
    - The BACKEND is the template renderer for Django. If you wanted to use Jinja2 templates, you would change BACKEND's value to do that.
    - APP_DIRS, when set true tells Django to look for templates directories is App directories.
    - DIRS allows us to tell Django where else it should look to find template directories.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

- Inside of `DIRS` add 'templates'

```
'DIRS': ['templates',],
```

- Now inside of the templates directory nearest the root of the project, add the file 'home.html'
- Add `<h1>Welcome</h1>` to 'home.html'.
- Open the views.py inside the second 'learning_site' directory, and change the following:

```
from django.shortcuts import render


def hello_world(requeset):
    return render(request, 'home.html')
```

1.10 Template Inheritance

`{% extends <template> %}` - Template tag to specify which template the current one should inherit from.

`{% block <name> %}` and `{% endblock %}` - Template tag pair to mark a section of a parent template as overridable. You can tack on `<name>` at the end of endblock for clarity if you wish.

Instructions:

- Create 'layout.html' inside the templates folder.
- Add the following to 'layout.html'

```
<!doctype html>
<html>
    <head>
        <title>{% block title %}{% endblock %}</title>
    </head>
    <body>
        {% block content %}{% endblock %}
    </body>
</html>
```

- Add `{% extends "layout.html" %}` to the top of 'home.html', & add the following.

```
{% block title %}Well Hello There!{% endblock %}

{% block content %}<h1>Welcome</h1>{% endblock %}
```

- Start app to check if the H1 Welcome is there.

1.11 Static Assets

`STATICFILES_DIRS` is a setting for where to find static files. These files will either be served during development or will end up being collected by the `collectstatic` command during deployment.

`staticfiles_urlpatterns()` is a function that returns the URL patterns for static files based on `DEBUG` and a few other settings. You import it from `django.contrib.staticfiles.urls`.

`{% load static from staticfiles %}` - The way to import the `{% static %}` tag for use.

`{% static "<path/to/asset>" %}` - How to use the `{% static %}` tag to properly point to a static asset.

Instructions:

- Create an the following folder paths in your root project directory: `/assets/css/`.
- Inside settings.py add the following under `STATIC_URL = '/static/'`.

```
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'assets'),
)
```

- You don't want a live site have python serve static files, but rather whatever server your using like nginx, or Apache. However, for testing purpose you do need to have Django's server serve those files, as well as tell Django where they are, and how.
- Inside of learning_site/learning_site/urls.py add the following:

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

 # place below url_patterns list
urlpatterns += staticfiles_urlpatterns()
```

staticfiles_urlpatterns() checks to see if we are in debug mode, if we are it creates a static path which is where all are static files are located.

- Inside layout.html add the following to the top of the file:

```
{% load static from staticfiles %}
```

- Add this inside the head tags:

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

1.12 Step by Step (Adding the steps for each course)

`IntegerField` is a field that holds integers, or whole numbers.

An *inline* is a smaller form inside of a larger form. The smaller form represents a related record in the database.

`StackedInline` is an inline where each field takes up the full width of the form. Fields are *stacked*.

`TabularInline` is an inline where each field is part of a single row for the form.

More docs on inlines.

Instructions: *Making a model for step, making a migration for it, then migrate it.

- Add the following to the bottom '/course/models.py':

```
class Step(models.Model):
    title = models.CharField(max_length=225)
    description = models.TextField()
    order = models.IntegerField(default=0)
    course = models.ForeignKey(Course)

    def __str__(self):
        return self.title
```

- In Termial type `python manage.py makemigrations courses`
- In Termial type `python manage.py migrate courses`
- Start-up server.
- In /courses/admin.py change your import statement to `from .models import Course, Step`.
- Add `admin.site.register(Step)` to the file as well.
- Enter the site admin, Add a step.
  - Title: 'What's the deal with strings?'
  - Description: 'Strings are more than just a bunch of letters. Find out why!'
  - Order: 0
  - Course: Python Basics
- Inside of '/courses/admin.py' add/CHANGE the following:

```
class StepInline(admin.StackedInline):  # This class by itself does nothing, but it will creat
    model = Step                        # the inline with the following code.


class CourseAdmin(admin.ModelAdmin):  # This is the admin for handling our courses, this will
    inlines = [StepInline,]           # us to specify special custom things.


admin.site.register(Course, CourseAdmin)  # Notice that we did not set up a model for CourseAd
                                          # doing that here.
admin.site.register(Step)
```

1.13 Add a Detail View

`{% for step in course.step_set.all %}` Notice that we don't use the `()` on `all()`. You don't call functions in Django's template tags, the template engine will handle that for you.

Also, step_set is automatically generated from the foreign key. Handy!

`Model.get(attribute=value)` lets you get a single Model instance by a given attribute's value.

Here is more info on `prefetch_related` and `select_related`. Don't bother too much with these until you're comfortable with Django's ORM.

Instructions:

- In '/course/views.py' add the following:

```
def course_detail(request, pk):  # Primary Key (pk), is the id of the course in the db.
    course = Course.objects.get(pk=pk)
    return render(request, 'courses/course_detail.html', {'course': course})
```

- In '/courses/urls.py': add the following to the urlpatterns list:

```
url(r'(?P<pk>\d+)/$', views.course_detail)
```

- In 'courses/templates/courses/' create 'course_detail.html'

- Add the following:

```
{% extends 'layout.html' %}

{% block title %}{{ course.title }}{% endblock %}

{% block content %}
<article>
    <h2>{{ course.title }}</h2>
    {{ course.description }}

    <section>
            {% for step in course.step_set.all %}
            <h3>{{ step.title }}</h3>
            {{ step.description }}
        {% endfor %}
    </section>
</article>
{% endblock %}
```

- Check to make sure it worked by going to '/courses/1' in the address bar.

1.14 Ordering and 404s

```
class Meta:
    ordering = ['field1', 'field2']
```

This will cause the model to be ordered by field1, then `field2` if there are any conflicts on `field1` (two instances having the same `field1` value). Finally, they'll be sorted by `id` if a conflict still exists.

`get_object_or_404(Model, [selectors])` - Gets an object of `Model` by using whatever selection arguments have been given. For example: get_object_or_404(User, username='kennethlove')`would try to get a User with an`username' set to "kennethlove". If that User didn't exist, a 404 error would be raised.

What's the long way? Consider this view:

```
from django.http import Http404

from .models import Course

def course_detail(request, pk):
    try:
        course = Course.objects.get(pk=pk)
    except Course.DoesNotExist:
        raise Http404()
    else:
        return render(request, 'courses/course_detail.html', {'course': course})
```

It's definitely more work!

If you want, you can customize your error views.

Instructions:

- Add the following to the 'Step' Model in models.py:

```
class Meta():
    ordering = ['order', ]  # Order our steps by their order fields, if that fails use thier i
```

- Add a new Step to Python Basics using the Django Admin:
  - Title: Using the shell
  - Description: What's the deal with strings?
- Set the order of the previous step to '1'.
- Re-Render /courses/1 to check that the new step we created is first.
- If Django is routed to an URL that it doesn't have, it throws a '500 Interal Server Error', instead of a '400 Page Not Found Error'. Lets Fix That.
- In /courses/views.py, change the Django imports to `from django.shortcuts import get_object_or_404, render`.
- Change course_detail function as follows:

```
def course_detail(request, pk):
    course = get_object_or_404(Course, pk=pk)
    return render(request, 'courses/course_detail.html', {'course': course})
```

1.15 Step Detail View

`linebreaks` is a filter that turns two returns in a row into HTML paragraph tags.
You apply filters to a variable with the pipe (|) character.

Instructions:

- Add the following to /courses/views.py:

```
def step_detail(request, course_pk, step_pk):
    step = get_object_or_404(Step, course_id=course_pk, pk=step_pk)
    return render(request, 'courses/step_detail.html', {'step': step})
```

- Add the following to urlpatterns list in courses/urls.py:

```
# Be sure to add this above course_detail, so that both the step pks, and the course pks are g
url(r'(?P<course_pk>\d+)/(?P<step_pk>\d+)/$', views.step_detail)
```

- Create a new template in courses/templates called step_detail.html, and enter the following:

```
{% extend 'layout.html' %}

{% block title %}{{ step.title }} - {{ step.course.title }}{% endblock %}

{% block content %}
<article>
    <h2>{{ step.course.title }}</h2>
```

```
    <h3>{{ step.title }}</h3>
    {{ step.content|linebreaks }}
</article>
{% endblock %}
```

`linebreaks` is a filter that turns two returns in a row into HTML paragraph tags.
You apply filters to a variable with the pipe (|) character.

1.16 Content Field

Instructions:

- Inside courses/models.py add the following to the Step model:

```
content = models.TextField()
```

- Create a migration for courses with `python manage.py makemigrations courses`
- You'll be presented with the following text:

```
You are trying to add a non-nullable field 'content' to step without a default; we can't do tha
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this
 2) Quit, and let me add a default in models.py
Select an option:
```

- Enter 2, and press Enter
- Change content in Step model to `content = models.TextField(blank=True, default='')`

blank=True - A field can be blank (not filled in) in the admin and any other forms based on the model.
default='something' - If no value is supplied for the field, the default 'something' will be put into the record.

- Try doing the makemigration again, and then migrate it with `python manage.py migrate courses`
- Add some content to one of the Steps using Django Admin, and then check it.

1.17 Code Challenge

Challenge Task 1 of 1

- We need to be able to associate a Writer with an Article. Add a ForeignKey field to the Article model to link it to the Writer model. Give the attribute the name writer. IMPORTANT Since our Article model comes first, you'll need to quote the Writer model in the foreign key. So use "Writer" instead of Writer.

```
from django.db import models


class Article(models.Model):
```

```
    headline = models.CharField(max_length=255)
    publish_date = models.DateTimeField()
    content = models.TextField()
    writer = models.ForeignKey("Writer")

    def __str__(self):
        return self.headline


class Writer(models.Model):
    name = models.CharField(max_length=255)
    email = models.EmailField()
    bio = models.TextField()
```

1.18 url Tag

{% url 'path.to.view' %} to link to a view who's URL doesn't have a name.

Note: This has been removed in Django 1.10 and beyond. If you want to use this feature, be sure to install Django 1.9 or below. You can do that with pip install django<1.10. Better yet, name all of your URLs as shown below.

url(r'pattern', views.view, name='list') to name an URL
{% url 'list' %} to link to a named URL
include('courses.urls', namespace='course') to namespace a group of URLs
{% url 'courses:list' %} to link to a named and namespaced URL

Instructions

- In learning_site/urls.py change the hello_world url to the following `url(r'^$', views.hello_world, name='hello_world')`
- In layout.html, add the following at the top of `div class='site-container'`:

```
<nav>
    <a href="{% url 'hello_world' %}">Home</a>
    <a href="{% url 'course_list' %}">Courses</a>
</nav>
```

- Change the following in courses/urls.py:

```
urlpatterns = [
    url(r'^$', views.course_list, name='course_list'),
    url(r'(?P<course_pk>\d+)/(?P<step_pk>\d+)/$', views.step_detail, name='step'),
    url(r'(?P<pk>\d+)/$', views.course_detail, name='detail'),
]
```

- Re-Render the page to see the new nav bar.
- Update course_list.html to the following:

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

```
{% block title %}Available Courses{% endblock %}

{% block content %}
<div class='cards'>
    {% for course in courses %}
    <div class='card'>
        <header><a href="{% url 'detail' pk=course.pk %}">{{ course.title }}</a></header>
        <div class='card-copy'>
            {{ course.description }}
        </div>
    </div>
    {% endfor %}
</div>
{% endblock %}
```

- Go to the courses list page to see the courses with their hyperlinked names.
- In course_detail.html Change the following:

```
<section>
        {% for step in course.step_set.all %}
        <h3>
            <a href="{% url 'step' course_pk=step.course.pk step_pk=step.pk %}">{{ step.title
        </h3>
        {{ step.description }}
    {% endfor %}
</section>
```

- Inside step_detail.html change the following:

```
<article>
    <h2>
        <a href="{% url 'detail' pk=step.course.pk %}">{{ step.course.title }}</a>
    </h2>
    <h3>{{ step.title }}</h3>
    {{ step.content|linebreaks }}
</article>
```

- Inside learning_site/urls.py add the namespace argument to courses url: `url(r'^courses/', include('courses.urls', namespace='courses'))` *This a preventive method used to avoid name conflicts with urls from other apps (most likely created by other users)*
- In layout.html change the following: `<a href="{% url 'courses:course_list' %}">Courses</a>`
- In course_list.html change the following: `<header><a href="{% url 'courses:detail' pk=course.pk %}">{{ course.title }}</a></header>`
- In course_detail.html change the following: `<a href="{% url "courses:step" course_pk=step.course.pk step_pk=step.pk %}">{{ step.title }}</a>`
- In step_detail.html change the following: `<a href="{% url 'course:detail' pk=step.course.pk %}">{{ step.course.title }}</a>`

1.19 Model Tests

`assertLess(a, b)` checks that a is less than b.

`django.utils.timezone` is Django's timezone utility that takes the `TIME_ZONE` setting into account.

`python manage.py test` runs all of the tests for your apps.

Instructions:

- In /courses/test.py add the following:

```
from django.test import TestCase
from django.utils import timezone

from .models import Course


class CourseModelTests(TestCase):

    def test_course_creation(self):
        course = Course.objects.create(
            title="Python Regular Expressions",
            description="Learn to write regular expressions in Python."
        )
        now = timezone.now()
        self.assertLess(course.created_at, now)
```

- Run the test by typing `python manage.py test` in the Terminal.

1.20 Code Challenge

Now add a test that creates an instance of the Writer model and, using self.assertIn, make sure the email attribute is in the output of the mailto() method.

```
from django.test import TestCase

from .models import Writer


class WriterModelTestCase(TestCase):
    '''Tests for the Writer model'''

    def test_writer_creation(self):
        writer = Writer.objects.create(
            name='Joe Smith',
            email='jsmith@example.com',
            bio='Joe is a writer'
        )

        self.assertIn(writer.email, writer.mailto())
```

1.21 View Tests

django.core.urlresolvers.reverse() takes a URL name and reverses it to the correct URL string. More information

self.client acts like a web browser and lets you make requests to URLs, both inside and outside of your Django project.

assertEqual(a, b) checks that a and b are equal to each other.

assertIn(a, b) checks that a is contained in b.

Instructions:

- In courses/tests.py update file to the following:

```
from django.core.urlresolvers import reverse
from django.test import TestCase
from django.utils import timezone

from .models import Course, Step


class CourseModelTests(TestCase):

    def test_course_creation(self):
        course = Course.objects.create(
            title="Python Regular Expressions",
            description="Learn to write regular expressions in Python."
        )
        now = timezone.now()
        self.assertLess(course.created_at, now)


class StepModelTests(TestCase):

    def setUp(self):
        self.course = Course.objects.create(
            title='Python Testing',
            description='Learn to write tests in Python.'
        )

    def test_step_creation(self):
        step = Step.objects.create(
            title="Introduction to Doctests",
            description="Learn to write docstrings",
            course=self.course
        )
        self.assertIn(step, self.course.step_set.all())


class CourseViewsTests(TestCase):
```

```
def setUp(self):
    self.course = Course.objects.create(
        title='Python Testing',
        description='A new course'
    )
    self.course2 = Course.objects.create(
        title='New Course',
        description='A new course'
    )
    self.step = Step.objects.create(
        title='Introduction to DocTests',
        description='Learn to write tests in your docstrings.',
        course=self.course
    )

def test_course_list_view(self):
    resp = self.client.get(reverse('courses:course_list'))
    self.assertEqual(resp.status_code, 200)
    self.assertIn(self.course, resp.context['courses'])
    self.assertIn(self.course2, resp.context['courses'])
```

FYI: Urls are tested while testing views, because in order to get to the view, a url must be used.

1.22 Template Tests

`assertTemplateUsed(response, 'template/name.html')` checks that a given template is used somewhere in the response of the view.

`assertContains(response, 'string')` checks that a given string is somewhere in the text of a response.

Instructions:

- Add the following to the bottom courses/tests.py:

```
def test_course_detail_view(self):
    resp = self.client.get(reverse('courses:detail',
                           kwargs={'pk': self.course2.pk}))
    self.assertEqual(resp.status_code, 200)
    self.assertEqual(self.course2, resp.context['course'])
    self.assertNotEqual(self.course, resp.context['course'])

def test_step_detail_view(self):
    resp = self.client.get(reverse('courses:step', kwargs={
                'course_pk': self.course2.pk,
                'step_pk': self.step2.pk}))
    self.assertEqual(resp.status_code, 200)
    self.assertEqual(self.step2, resp.context['step'])
    self.assertNotEqual(self.step, resp.context['step'])
```

- Add the following to the bottom of test_course_list_view:

```
self.assertTemplateUsed(resp, 'courses/course_list.html')
self.assertContains(resp, self.course.title)
```