Django Tutorial-1

Create a venv:

```
python3 -m venv myenv
```

myenv being the name of the vnev

Activate your venv:

```
source myenv/bin/activate
```

Deactivate your venv:

```
Deactivate
```

Note: you have to be in the correct directory (root directory of your project) for all of the above.

Remove a venv:

```
deactivate
rm -rf myenv/
```

Install Django on your Venv/computer:

```
python3 -m pip install Django
```

Check Django version on your computer:

```
python3 -m django --version
```

Create a Django project:

```
django-admin startproject mysite
```

This creates mysite directory in the directory you in:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

These files are:

- The outer **mysite/** root directory is a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- **manage.py**: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about **manage.py** in django-admin and manage.py.
- The inner **mysite/** directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. **mysite.urls**).
- **mysite/__init__.py**: An empty file that tells Python that this directory should be considered a Python package. If you're a Python beginner, read more about packages in the official Python docs.
- **mysite/settings.py**: Settings/configuration for this Django project. Django settings will tell you all about how settings work.
- **mysite/urls.py**: The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in URL dispatcher.
- **mysite/asgi.py**: An entry-point for ASGI-compatible web servers to serve your project. See How to deploy with ASGI for more details.
- **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project. See How to deploy with WSGI for more details.

The development server:

```
python3 manage.py runserver
```

You've started the Django development server, a lightweight web server written purely in Python. Now that the server's running, visit http://127.0.0.1:8000/ with your web browser. You'll see a "Congratulations!" page, with a rocket taking off. It worked!

In order to run the server in a different port (default port is 8000):

```
python3 manage.py runserver 8080
```

If you want to change the server's IP address:

```
python manage.py runserver 0.0.0.0:8000
```

Creating an app (poll app):

Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.
Your apps can live anywhere on your Python path. In this tutorial, we'll create our poll app in the same directory as your **manage.py** file so that it can be imported as its own top-level module, rather than a submodule of **mysite**. To create your app, make sure you're in the same directory as **manage.py** and type this command:

```
python3 manage.py startapp polls
```

This will create the directory `polls`:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

Write your first view:

In your `polls/views.py` put the following code:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

To call the view, we need to map it to a URL - and for this we need a `URLconf`. To create a `URLconf` in the polls directory, create a file called `urls.py`.

In your `polls/urls.py`, insert the following code:

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"),
]
```

The next step is to point the root URLconf at the **polls.urls** module. In **mysite/urls.py**, add an import for **django.urls.include** and insert an **include()** in the **urlpatterns** list, so you have:

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("polls/", include("polls.urls")),
    path("admin/", admin.site.urls),
]
```

What this does is that whenever there's a request with `polls/` the server looks into `mysite/urls.py`, and if it finds `poll/` in the `urlpatterns`, then it is referred to `polls/urls.py` which is the `urls` for the app `polls/.`

You should always use **include()** when you include other URL patterns. **admin.site.urls** is the only exception to this.

You have now wired an **index** view into the URLconf. Verify it's working with the following command:

```
python3 manage.py runserver
```

And now go to: http://localhost:8000/polls/

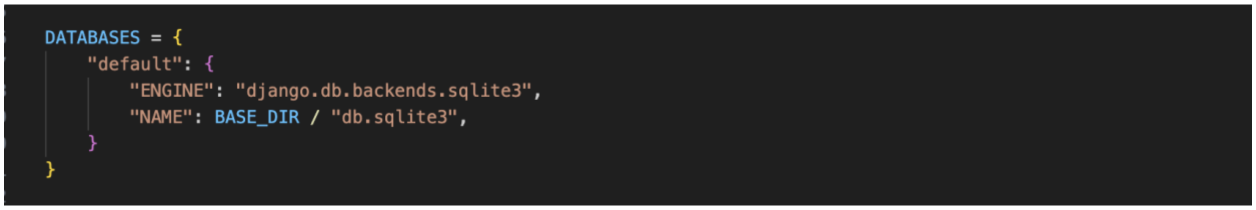
`Path()` has 4 arguments: route and view are required, and `kwargs` and `name` are optional:

- **route** is a string that contains a URL pattern, patterns don't search GET and POST parameters, or domain names
- **view**: when Django finds a matching pattern it calls the specified view function with an HttpRequest object as the first argument and any "captured" values from the route as keyword arguments.
- **Kwargs**: Arbitrary keyword arguments can be passed in a dictionary to the target view. We aren't going to use this feature of Django in the tutorial.
- **Name**: Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This powerful feature allows you to make global changes to the URL patterns of your project while only touching a single file.

Django Tutorial-2

Database setup

Open up `mysite/settings.py.` It's a normal Python module with module-level variables representing Django settings. By default, the configuration uses SQLite.



```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

This is the default configurations for database in the settings.py file.

Engine – Either **'django.db.backends.sqlite3'**, **'django.db.backends.postgresql'**, **'django.db.backends.mysql'**, or **'django.db.backends.oracle'**.

NAME – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, **NAME** should be the full absolute path, including filename, of that file. The default value, **BASE_DIR / 'db.sqlite3'**, will store the file in your project directory.

If you are not using SQLite as your database, additional settings such as **USER**, **PASSWORD**, and **HOST** must be added. For more details, see the reference documentation for **DATABASES**.

If you're using a database besides SQLite, make sure you've created a database by this point. Do that with "**CREATE DATABASE database_name;**" within your database's interactive prompt.

Also make sure that the database user provided in **mysite/settings.py** has "create database" privileges. This allows automatic creation of a test database which will be needed in a later tutorial. This is already taken care of in SQLite.

While you're editing **mysite/settings.py**, set **TIME_ZONE** to your time zone.
Also, note the **INSTALLED_APPS** setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.
By default, **INSTALLED_APPS** contains the following apps, all of which come with Django:

- **django.contrib.admin** – The admin site. You'll use it shortly.
- **django.contrib.auth** – An authentication system.
- **django.contrib.contenttypes** – A framework for content types.
- **django.contrib.sessions** – A session framework.
- **django.contrib.messages** – A messaging framework.
- **django.contrib.staticfiles** – A framework for managing static files.

Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

```
python3 manage.py migrate
```

The **migrate** command looks at the **INSTALLED_APPS** setting and creates any necessary database tables according to the database settings in your **mysite/settings.py** file and the database migrations shipped with the app (we'll cover those later).

To view the tables (in sqlite) cd to the BASE_DIR, and then type `sqlite3 db.sqlite3`
This launches sqlite. Within sqlite prompt then type: `.tables` to view the tables created by the above step. Once done you can type `.exit` to quit sqlite.

Note: Like we said above, the default applications are included for the common case, but not everybody needs them. If you don't need any or all of them, feel free to comment-out or delete the appropriate line(s) from **INSTALLED_APPS** before running **migrate**. The **migrate** command will only run migrations for apps in **INSTALLED_APPS**.

Creating models:

Now we'll define your models – essentially, your database layout, with additional metadata.

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Django follows the DRY Principle (Don't repeat yourself – Every distinct concept and/or piece of data should live in one and only one pace. Redundancy is bad, normalization is good). The goal is to define your data model in one place and automatically derive things from it.

In our poll app, we create two models: Question (question and publication date) and Choice (text of choice and vote tally). Each choice is associated with a question. Therefore in the file `polls/models.py` we type in:

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField("date published")


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Here, each model is represented by a class that subclasses **`django.db.models.Model`**. Each model has a number of class variables, each of which represents a database field in the model. Each field is represented by an instance of a **Field** class – e.g., **CharField** for character fields and **DateTimeField** for datetimes. This tells Django what type of data each field holds.

You can use an optional first positional argument to a **Field** to designate a human-readable name. That's used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn't provided, Django will use the machine-readable name. In this example, we've only defined a human-readable name for **Question.pub_date**. For all other fields in this model, the field's machine-readable name will suffice as its human-readable name.

```
pub_date = models.DateTimeField("date published")
```

Note: on_delete=models.CASCADE means that when a question object is deleted, also delete all related **Choice** objects that have a foreign key to that **Question**.

Activating models

Using the models, Django:

- Create a database schema (**CREATE TABLE** statements) for this app.
- Create a Python database-access API for accessing **Question** and **Choice** objects.

We need to tell our project that polls app is installed by adding polls to `mysite/settings.py` file by adding `polls.apps.PollsConfig` to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    "polls.apps.PollsConfig",
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
]
```

Since we made changes to models (or made new ones) we need to tell Django to store these changes as *migration* by:

```
python3 manage.py makemigrations polls
```

Migrations are how Django stores changes to you models and thus your databse schema. You can read the migration for your new model if you like. They are in the file: `polls/migrations/0001_initial.py`

The **`sqlmigrate`** command takes migration names and returns their SQL:

```
python3 manage.py sqlmigrate polls 0001
```

which shows the following (reformatted for readability – also generated for PostgreSQL – it will look deferent based on your database enging):

```
BEGIN;
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL,
    "question_id" bigint NOT NULL
);
ALTER TABLE "polls_choice"
  ADD CONSTRAINT
"polls_choice_question_id_c5b4b260_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
    DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice"
("question_id");

COMMIT;
```

Note:

- Table names are automatically generated by combining the name of the app (**polls**) and the lowercase name of the model – **question** and **choice**. (You can override this behavior.)
- Primary keys (IDs) are added automatically. (You can override this, too.)
- By convention, Django appends **"_id"** to the foreign key field name. (Yes, you can override this, as well.)
- The foreign key relationship is made explicit by a **FOREIGN KEY** constraint. Don't worry about the **DEFERRABLE** parts; it's telling PostgreSQL to not enforce the foreign key until the end of the transaction.

- The **sqlmigrate** command doesn't actually run the migration on your database - instead, it prints it to the screen so that you can see what SQL Django thinks is required. It's useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

Now run migrate again to create those model tables in your database:

```
python3 manage.py migrate
```

The **migrate** command takes all the migrations that haven't been applied (Django tracks which ones are applied using a special table in your database called **django_migrations**) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database.

To reiterate, remember the three-step guide to making model changes:

- Change your models (in **models.py**).
- Run `python3 manage.py makemigrations` to create migrations for those changes
- Run `python3 manage.py migrate` to apply those changes to the database.

Playing with the API:

Let's hop into the interactive python shell and play around with the free API Django gives you, type:

```
python3 manage.py shell
```

We're using this instead of simply typing "python", because **manage.py** sets the **DJANGO_SETTINGS_MODULE** environment variable, which gives Django the Python import path to your **mysite/settings.py** file.

Once you are in the shell let's explore the database API:

```
>>> from polls.models import Choice, Question  # Import the model
classes we just wrote.

# No questions are in the system yet.
>>> Question.objects.all()
<QuerySet []>

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use
timezone.now()
```

```
# instead of datetime.datetime.now() and it will do the right thing.
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save()
explicitly.
>>> q.save()

# Now it has an ID.
>>> q.id
1

# Access model field values via Python attributes.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217,
tzinfo=datetime.timezone.utc)

# Change values by changing the attributes, then calling save().
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>
```

Since `[<Question: Question object (1)>]` is not very descriptive we can adjust our models by adding a donder method to our classes.

```
from django.db import models


class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text


class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text
```

Let’s also add a custom method to this model:

```
import datetime

from django.db import models
from django.utils import timezone


class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

More shell playing aroud:

```
>>> from polls.models import Choice, Question

# Make sure our __str__() addition worked.
>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith="What")
<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides
a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
```

```
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a
new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django
creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text="Not much", votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text="The sky", votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text="Just hacking again", votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just
hacking again>]>
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just
hacking again>]>

# Let's delete one of the choices. Use delete() for that.
>>> c = q.choice_set.filter(choice_text__startswith="Just hacking")
>>> c.delete()
```

Notes:

- `.get()` is used to match one and only one object. If no object is found it raises **DoesNotExist,** while if more than one object is matched **MultipleObjectsReturned** is raised.
- If you expect more than object to be returned, use `.filter()`
- **.save()** vs **.commit()** (in flask-sqlalchemy): **.save()** is called on individual model instances and immediately writes changes to the database. On the other hand, **.commit()** is called on the session object, allowing you to batch multiple changes together into a single transaction before committing them to the database.
- In Django's QuerySet API, double underscores (__) are used to separate field names from query operations. E.g. `__startwith` is a string-based filter that is case-sensitive by default.

  ```
  >>> Question.objects.filter(question_text__startswith="What")
  >>> Question.objects.get(pub_date__year=current_year)
  ```

- In Django, when you define a **ForeignKey** relationship like in your **Choice** model, Django automatically creates a "reverse" relation from the model that is being pointed to (**Question** in this case) back to the model that defines the foreign key (**Choice**). By default, this "reverse" relation is named **<model_name>_set**, where **<model_name>** is the name of the model that defines the foreign key, in lowercase. In here:

  ```
  c = q.choice_set.create(choice_text="Just hacking again", votes=0)
  ```

  where **q** is assumed to be an instance of **Question** and **q.choice_set** accesses the set of all **Choice** objects that belong to this particular question **q**.

  Therefore you can access all of any instances questions choices, and you can also access the question related to an instance of a choice:

  ```
  c.question
  ```

- Note that while using `.get()` in Django, if the entry is not found in the database, you won't get a None return, instead a **DoesNotExist** exception is raised.

Introducing the Django Admin

Generating admin sites for your staff or clients to add, change, and delete content is tedious work that doesn't require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.
The admin isn't intended to be used by site visitors. It's for site managers.

Creating an admin user

First we create a user who can login to the admin site. Run the following command:

```
python3 manage.py createsuperuser
```

You will be prompted for a admin username – enter your desired username and press enter.

```
Username: admin
```

Then you will be prompted for your desired email:

```
Email address: admin@example.com
```

The final step is to enter your password:

```
Password: **********
Password (again): *********
Superuser created successfully.
```

Start the development server

The Django admin site is activated by default. Let's start the development server and explore it. If the server is not running start it like so:

```
python manage.py runserver
```

Now open a web browser and go to "/admin/" on your local domain,
i.e. http://127.0.0.1:8000/admin/, which should take you to the admin's login screen.

Enter the admin site

Once logged in as superuser account, you should see a few types of editable content: **groups** and **users**. These are provided by **django.contrib.auth**, the authentication framework shipped by Django.

Make the poll app modifiable in the admin

Only one more thing to do: we need to tell the admin that **Question** objects have an admin interface. To do this, open the **polls/admin.py** file, and edit it to look like this:

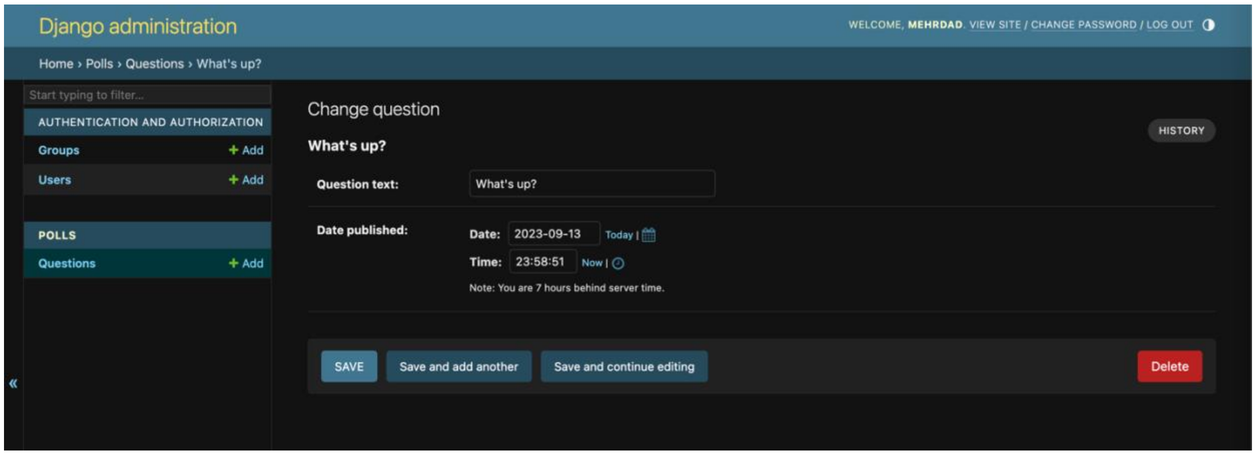
```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

Explore the free admin functionality

Now that we've registered **Question**, Django knows that it should be displayed on the admin index page.
Click "Questions". Now you're at the "change list" page for questions. This page displays all the questions in the database and lets you choose one to change it. There's the "What's up?" question we created earlier.
Click the "What's up?" question to edit it…

Notes:

- The form is automatically generated from the Question model
- The different model field types (**DateTimeField**, **CharField**) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each **DateTimeField** gets free JavaScript shortcuts. Dates get a "Today" shortcut and calendar popup, and times get a "Now" shortcut and a convenient popup that lists commonly entered times.


Django administration
WELCOME, MEHRDAD. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Polls › Questions › What's up?
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POLLS
Questions + Add
HISTORY
SAVE
Save and add another
Save and continue editing
Delete

Change question

What's up?

Question text: What's up?

Date published: Date: 2023-09-13 Today | Time: 23:58:51 Now |

Note: You are 7 hours behind server time.

The bottom part of the page gives you a couple of options:

- Save – Saves changes and returns to the change-list page for this type of object.
- Save and continue editing – Saves changes and reloads the admin page for this object.
- Save and add another – Saves changes and loads a new, blank form for this type of object.
- Delete – Displays a delete confirmation page.

You can view the change history by clicking the **history** button.

Django – Tutorial 3:

Overview:

A view is a “type” of web page in your Django application that generally serves a specific function and has a specific template. For example, in a blog application, you might have the following views:

- Blog homepage – displays the latest few entries.
- Entry “detail” page – permalink page for a single entry.
- Year-based archive page – displays all months with entries in the given year.
- Month-based archive page – displays all days with entries in the given month.
- Day-based archive page – displays all entries in the given day.
- Comment action – handles posting comments to a given entry.

E.g. in our poll app:

- Question “index” page – displays the latest few questions.
- Question “detail” page – displays a question text, with no results but with a form to vote.
- Question “results” page – displays results for a particular question.
- Vote action – handles voting for a particular choice in a particular question.

In Django, web pages and other content are delivered by views. Each view is represented by a Python function (or method, in the case of class-based views).
Django will choose a view by examining the URL that’s requested (to be precise, the part of the URL after the domain name).

A URL pattern is the general form of a URL - for example: **/newsarchive/<year>/<month>/**.
To get from a URL to a view, Django uses what are known as ‘URLconfs’. A URLconf maps URL patterns to views.

Writing more views:

Let’s add more vies to `polls/views.py`:

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." %
question_id)


def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)


def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

About request argument: the **request** argument in each view function is an instance of the class **HttpRequest**. This object contains all the details about the incoming HTTP request from the client (browser, Postman, etc.) to the server. Breakdown of what you can find in the `request` object:

- **request.method**: The HTTP method used in the request (**'GET'**, **'POST'**, etc.).
- **request.GET**: A dictionary-like object containing all GET parameters.
- **request.POST**: A dictionary-like object containing all POST parameters.
- **request.FILES**: A dictionary-like object containing all uploaded files.
- **request.user**: An instance representing the currently logged-in user. If no user is logged in, this will be an instance of **AnonymousUser**.
- **request.META**: A dictionary containing all available HTTP headers. You can access headers like **'HTTP_USER_AGENT'**, **'HTTP_REFERER'**, etc., from here.

About the '%' operator:
The **%** operator is used in Python for old-style string formatting. It's a way to substitute a set of placeholders with corresponding values. The **%s** in the string indicates a placeholder where a string representation of a value will be inserted. You can think of **%s** as a "slot" where a string will be inserted. So essentially similar to formatted string literals.

Now that we have these view we have to wire them into the `polls.urls` by adding the following `path()` calls:

```
urlpatterns = [
    # ex: /polls/
    path("", views.index, name="index"),
    # ex: /polls/5/
    path("<int:question_id>/", views.detail, name="detail"),
    # ex: /polls/5/results/
    path("<int:question_id>/results/", views.results, name="results"),
    # ex: /polls/5/vote/
    path("<int:question_id>/vote/", views.vote, name="vote"),
]
```

When somebody requests a page from your website – say, "/polls/34/", Django will load the **mysite.urls** Python module because it's pointed to by the [ROOT_URLCONF]() setting. It finds the variable named **urlpatterns** and traverses the patterns in order. After finding the match at **'polls/'**, it strips off the matching text (**"polls/"**) and sends the remaining text – **"34/"** – to the 'polls.urls' URLconf for further processing. There it matches **'<int:question_id>/'**, resulting in a call to the **detail()** view like so:

```
detail(request=<HttpRequest object>, question_id=34)
```

Write vies that actually do something:

Each view is responsible for doing one of two things: returning an **HttpResponse** object containing the content for the requested page, or raising an exception such as **Http404**. The rest is up to you.

Because it's convenient, let's use Django's own database API, which we covered in Tutorial 2. Here's one stab at a new **index()** view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

```
from django.http import HttpResponse

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by("-pub_date")[:5]
    output = ", ".join([q.question_text for q in
latest_question_list])
    return HttpResponse(output)
```

There's a problem here, though: the page's design is hard-coded in the view. If you want to change the way the page looks, you'll have to edit this Python code. So let's use Django's template system to separate the design from Python by creating a template that the view can use.

First, create a directory called **templates** in your **polls** directory. Django will look for templates in there.

Your project's **TEMPLATES** setting describes how Django will load and render templates. The default settings file configures a **DjangoTemplates** backend whose **APP_DIRS** option is set to **True**. By convention **DjangoTemplates** looks for a "templates" subdirectory in each of the **INSTALLED_APPS**.
Within the **templates** directory you have just created, create another directory called **polls**, and within that create a file called **index.html**. In other words, your template should be at
In other words, your template should be at **polls/templates/polls/index.html**. Because of how the **app_directories** template loader works as described above, you can refer to this template within Django as **polls/index.html**.

Note: Now we *might* be able to get away with putting our templates directly in polls/templates (rather than creating another polls subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a *different* application, Django would be unable to distinguish between them.

Now put the following code in index.html:

```
`{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Now let's update our **index** view in **polls/views.py** to use the template:

```
from django.http import HttpResponse
from django.template import loader

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by("-pub_date")[:5]
    template = loader.get_template("polls/index.html")
    context = {
        "latest_question_list": latest_question_list,
    }
    return HttpResponse(template.render(context, request))
```

Notes:

- the '–' before pub_date indicates descending order.
- **template = loader.get_template("polls/index.html")**: This line loads a Django template located at **polls/index.html** using Django's **loader** module. The loaded template will be used to render the HTML that gets sent back to the client.
- **context = {"latest_question_list": latest_question_list,}**: This sets up a dictionary called **context** containing the data that will be sent to the template. The key **"latest_question_list"** will be accessible in the template, allowing you to iterate over its contents, as you showed in the previous template example. **Note:** you don't need to access this data by using key to get the value.

A shortcut: render()

It’s a very common idiom to load a template, fill a context and return an **HttpResponse** object with the result of the rendered template. Django provides a shortcut. Here’s the full **index()** view, rewritten:

```
from django.shortcuts import render

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by("-pub_date")[:5]
    context = {"latest_question_list": latest_question_list}
    return render(request, "polls/index.html", context)
```

Note that once we’ve done this in all these views, we no longer need to import **loader** and **HttpResponse** (you’ll want to keep **HttpResponse** if you still have the stub methods for **detail**, **results**, and **vote**).

The **render()** function takes the request object as its first argument, a template name as its second argument and a dictionary as its optional third argument. It returns an **HttpResponse** object of the given template rendered with the given context.

Raising a 404 error

Now, let’s tackle the question detail view – the page that displays the question text for a given poll. Here’s the view:

```
from django.http import Http404
from django.shortcuts import render

from .models import Question


# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, "polls/detail.html", {"question":
question})
```

The new concept here: The view raises the **Http404** exception if a question with the requested ID doesn’t exist.

For now you can put the following in the your `polls/templates/polls/detail`

```
{{ question }}
```

A shortcut: get_object_or_404()

It's a very common idiom to use get() and raise Http404 if the object doesn't exist. Django provides a shortcut. Here's the **detail()** view, rewritten:

```
from django.shortcuts import get_object_or_404, render

from .models import Question


# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, "polls/detail.html", {"question":
question})
```

The get_object_or_404() function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the get() function of the model's manager. It raises Http404 if the object doesn't exist.

Note:
Why do we use a helper function get_object_or_404() instead of automatically catching the ObjectDoesNotExist exceptions at a higher level, or having the model API raise Http404 instead of ObjectDoesNotExist?
Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling. Some controlled coupling is introduced in the django.shortcuts module.

Use the template system

Now let's change detail template:

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

The template system uses dot-lookup syntax to access variable attributes. In the example of **{{ question.question_text }}**, first Django does a dictionary lookup on the object **question**.

Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup had failed, it would've tried a list-index lookup.

Removing hardcoded URLs in templates:

Remember, when we wrote the link to a question in the **polls/index.html** template, the link was partially hardcoded like this:

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates. However, since you defined the name argument in the **path()** functions in the **polls.urls** module, you can remove a reliance on specific URL paths defined in your url configurations by using the **{% url %}** template tag:

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

The way this works is by looking up the URL definition as specified in the **polls.urls** module. You can see exactly where the URL name of 'detail' is defined below:

```
...
# the 'name' value as called by the {% url %} template tag
path("<int:question_id>/", views.detail, name="detail"),
...
```

If you want to change the URL of the polls detail view to something else, perhaps to something like **polls/specifics/12/** instead of doing it in the template (or templates) you would change it in **polls/urls.py**:

```
...
# added the word 'specifics'
path("specifics/<int:question_id>/", views.detail, name="detail"),
...
```

Namespacing URL names