

A Project Report

on

Web API

By

Kashish Goyal

For

Project Submission

To

Jala Academy

Bonafide Certificate

This is to certify that this report entitled “**Web API**” submitted to **Jala Academy** is a bonafide record of work done by “**Kashish Goyal**”.

Date: **21/04/2022**

Place: **Delhi**

Table of Contents

1. Introduction.....	4
2. Objective and Scope.....	4
3. Software and Programming Languages used.....	4
4. Installation Guide.....	5
5. Description.....	6-32
➤ Creating project.....	6-8
➤ Creating polls app.....	8
➤ Database setup.....	8-9
➤ Creating models.....	10-15
➤ Playing with API.....	15-20
➤ Creating admin user.....	21
➤ Start development server.....	21-23
➤ Make the poll app modifiable in admin.....	23
➤ Explore the free admin functionality.....	23-25
➤ Overview.....	25
➤ Writing views that actually do something.....	26-27
➤ Customize your apps looks and feels.....	27-28
➤ Customize admin form.....	28-30
➤ Customizing project template.....	31-32
6. Output.....	32-33
7. Features	34
8. Limitations.....	34

1. Introduction

This document describes how to make Web API using python. A Web API is an application programming interface for the Web and Python is popular programming language.

The Web API will provide users a platform to answer multiple choice questions and take surveys to gather information about people's opinions. This information can be used for business purpose or keeping a record for future references. People can also share their opinion to everyone anonymously. Students can solve some multiple choice questions for revision and know what majority of other students answered.

2. Objective and Scope

- To develop a Web API where you can login as admin/super user which has admin controls.
- To develop a Web app where one can answer some multiple choice questions with ease.
- Where one can also take surveys and share their opinions.
- To provide a platform which can be easily accessible and used with ease.

3. Software and Programming Languages Used

Visual Studio Code

Commonly referred to as VS Code, is a source-code editor made by Microsoft for Windows, Linux or macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring and embedded Git.

Programming languages:

- **Python**
- **Django python**
- **HTML**
- **CSS**
- **SQL**

4. Installation Guide

- **Installing Visual Studio Code (Vs code):**

Open browser and search download Visual studio code and click on first link.

Or

Type this link to visit official site to download Vs code

<https://code.visualstudio.com/download>

Now install it according to your software preferences.

- **Installing Python:**

Open browser and search download python and click on first site which will be official site for python. This is link to official site for downloading python <https://www.python.org/> Click on download. After downloading it complete the setup step by step to complete installation of python in your system.

- **Installing Django:**

After python is installed in your system, you can simply run a command to install Django in command prompt. As we are using Visual Studio code we can simply run this code **py -m pip install django** to install django.

Or

You can download django from official site <https://www.djangoproject.com/>

- **Installing HTML,CSS:**

Install HTML and CSS from extension tab provided in Vs code.

5. Description

In this project we will be using python, django, html and css for creating a Web API.

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It is free and open source, has a thriving and active community, great documentation, and many options for free and paid-for support.

We will be using Django to create our own development server, a lightweight web server written purely in Python.

➤ **Creating a project:**

We have to first complete initial setup where we auto generate some code that establishes a Django project- a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, **cd** into a directory where you'd like to store your code, then run the following command:

django-admin startproject mysite

This will create a **mysite** directory in your current directory.

Let's look at what **startproject** created:

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.

Let's verify our Django project:

To verify the project change into outer **mysite** directory and run the following command

py manage.py runserver

You'll see the following output on the command line:

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

You have unapplied migrations; your app may not work properly until they are applied.

Run 'python manage.py migrate' to apply them.

April 20, 2022 - 15:50:53

Django version 4.0, using settings 'mysite.settings'

Starting development server at <http://127.0.0.1:8000/>

Quit the server with CONTROL-C.

Now that the server is running, visit <http://127.0.0.1:8000/> with your browser. You'll see "Congratulations!" page, with a rocket taking off.

➤ Creating Polls app:

We will create our poll app in same directory as **manage.py** so that it can be imported as its own top-level module, rather than a submodule of mysite.

To create the app make sure we are in the same directory as **manage.py** and type this command:

py manage.py startapp polls

That will create directory **polls**, which is laid out like this:

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



```
tests.py
```

```
views.py
```

This directory structure will house the poll application.

➤ Database setup:

Now, open up **mysite/settings.py**. It's a normal Python module with module-level variables representing Django settings.

By default, the configuration uses SQLite. If you're new to databases, or you're just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won't need to install anything else to support your database.

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.

These applications are included by default as a convenience for the common case.

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:

py 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. You'll see a message for each migration it applies.

➤ Creating Models:

In our poll app, we'll create two models: **Question** and **Choice**. A **Question** has a question and a publication date. A **Choice** has two fields: the text of the choice and a vote tally. Each **Choice** is associated with a **Question**.

These concepts are represented by Python classes. Edit the **polls/models.py** file so it looks like this:

polls/models.py

```
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.

The name of each **Field** instance (e.g. **question_text** or **pub_date**) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

A **Field** can also have various optional arguments; in this case, we've set the **default** value of **votes** to 0.

Finally, note a relationship is defined, using **ForeignKey**. That tells Django each **Choice** is related to a single **Question**. Django supports all the common database relationships: many-to-one, many-to-many, and one-to-one.

- **Activating models:**

That small bit of model code gives Django a lot of information. With it, Django is able to:

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

But first we need to tell our project that the **polls** app is installed.

To include the app in our project, we need to add a reference to its configuration class in the **INSTALLED_APPS** setting. The **PollsConfig** class is in the **polls/apps.py** file, so its dotted path is **'polls.apps.PollsConfig'**. Edit the **mysite/settings.py** file and add that dotted path to the **INSTALLED_APPS** setting. It'll look like this:

```
mysite/settings.py
```

```
INSTALLED_APPS = [  
    'polls.apps.PollsConfig',  
]
```

```
'django.contrib.admin',  
'django.contrib.auth',  
'django.contrib.contenttypes',  
'django.contrib.sessions',  
'django.contrib.messages',  
'django.contrib.staticfiles',  
]
```

Now Django knows to include the **polls** app. Let's run another command:

py manage.py makemigrations polls

You should see something similar to the following:

```
Migrations for 'polls':  
  polls/migrations/0001_initial.py  
    - Create model Question  
    - Create model Choice
```

By running **makemigrations**, you're telling Django that you've made some changes to your models (in this case, you've made new ones) and that you'd like the changes to be stored as a *migration*.

Migrations are how Django stores changes to your models (and thus your database schema) - they're files on disk. You can read the migration for your new model if you like; it's the file **polls/migrations/0001_initial.py**. Don't worry, you're not expected to read them every time Django makes one, but

they're designed to be human-editable in case you want to manually tweak how Django changes things.

There's a command that will run the migrations for you and manage your database schema automatically - that's called **migrate**, and we'll come to it in a moment - but first, let's see what SQL that migration would run.

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

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

You should see something similar to the following:

```
BEGIN;

--

-- Create model Question
--

CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);

--

-- Create model Choice
--

CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL,
    "question_id" integer 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 the following:

- The exact output will vary depending on the database you are using. The example above is generated for PostgreSQL.
- Table names are automatically generated by combining the name of the app (**polls**) and the lowercase name of the model – **question** and **choice**.
- Primary keys (IDs) are added automatically.
- By convention, Django appends **"_id"** to the foreign key field name.
- 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.
- It's tailored to the database you're using, so database-specific field types such as **auto_increment** (MySQL), **serial** (PostgreSQL), or **integer primary key autoincrement** (SQLite) are handled for you automatically. Same goes for the quoting of field names – e.g., using double quotes or single quotes.
- 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:

```
py manage.py migrate
```

```
Operations to perform:
```

```
  Apply all migrations: admin, auth, contenttypes, polls, sessions
```

```
Running migrations:
```

```
  Rendering model states... DONE
```

```
  Applying polls.0001_initial... OK
```

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.

Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data.

➤ **Playing with API:**

Now, let's hop into the interactive Python shell and play around with the free API Django gives you. To invoke the Python shell, use this command:

```
py 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're in the shell, 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=<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)>)]>
```

Wait a minute. **<Question: Question object (1)>** isn't a helpful representation of this object. Let's fix that by editing the **Question** model (in the **polls/models.py** file) and adding a **__str__()** method to both **Question** and **Choice**:

polls/models.py

```
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
```

It's important to add **__str__()** methods to your models, not only for your own convenience when dealing with the interactive prompt, but also because objects' representations are used throughout Django's automatically-generated admin.

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

polls/models.py

```

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)

```

Note the addition of **import datetime** and **from django.utils import timezone**, to reference Python's standard **datetime** module and Django's time-zone-related utilities in **django.utils.timezone**, respectively. Save these changes and start a new Python interactive shell by running **python manage.py shell** again:

```

>>> 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()
```

➤ Creating an admin user:

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

```
py manage.py createsuperuser
```

Enter your desired username and press enter.

```
Username: admin
```

You will then be prompted for your desired email address:

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

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

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

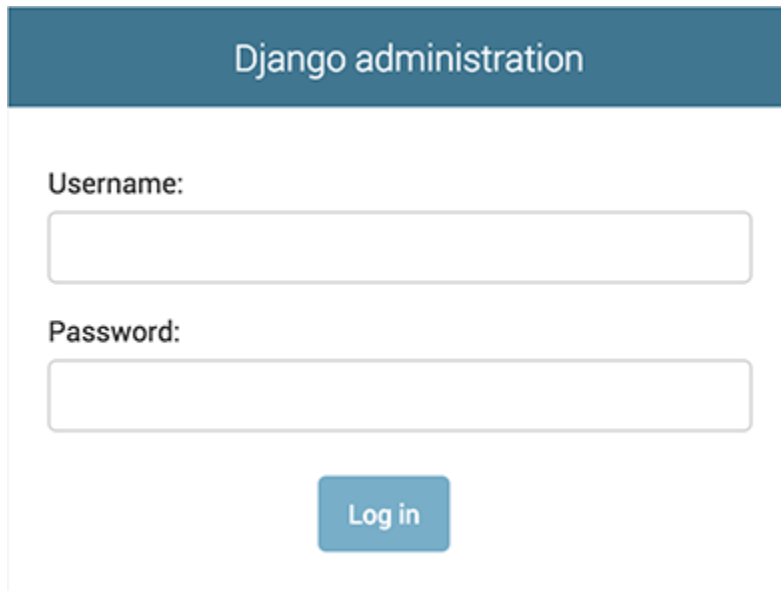
➤ Start the Development server:

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:

py manage.py runserver

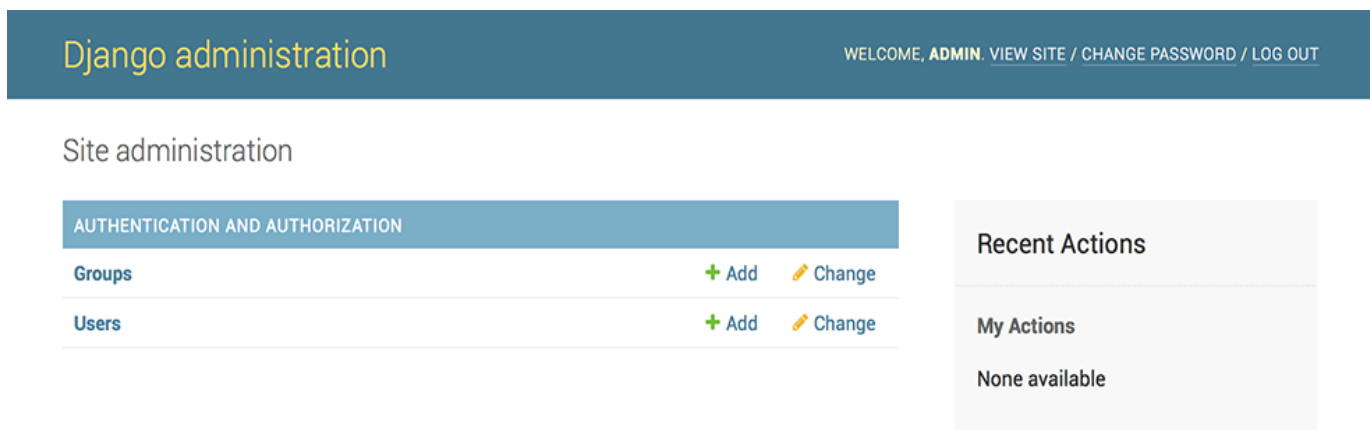
Now, open a web browser and go to “/admin/” on your local domain – e.g., <http://127.0.0.1:8000/>. You should see the admin’s login screen:

The image shows the Django administration login interface. It features a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". A blue "Log in" button is positioned below the password field.

Since translation is turned on by default, if you set **LANGUAGE_CODE**, the login screen will be displayed in the given language (if Django has appropriate translations).

- **Enter the admin site:**

Now, try logging in with the superuser account you created in the previous step. You should see the Django admin index page:

The image shows the Django administration index page. The header is dark blue with "Django administration" on the left and "WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT" on the right. Below the header, the page is titled "Site administration". There is a table with two rows: "Groups" and "Users". Each row has a "+ Add" link and a "Change" link with a pencil icon. To the right of the table is a "Recent Actions" section with a "My Actions" subsection, which currently shows "None available".

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add Change
Users	+ Add Change

Recent Actions

My Actions
None available

You should see a few types of editable content: groups and users. They are provided by **django.contrib.auth**, the authentication framework shipped by Django.

➤ Make the poll app modifiable in admin:

But where's our poll app? It's not displayed on the admin index page.

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:

```
polls/admin.py

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:

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change
POLLS		
Questions	+ Add	Change

Recent Actions
My Actions
None available

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:

The screenshot shows the Django admin interface for the 'Questions' model. At the top, a blue breadcrumb bar reads 'Home > Polls > Questions'. Below this, the page title is 'Select question to change'. In the top right corner, there is a button labeled 'ADD QUESTION +'. On the left, there is an 'Action:' dropdown menu with a dashed line, a 'Go' button, and the text '0 of 1 selected'. The main content area is a table with two rows: the first row has a checkbox and the text 'QUESTION'; the second row has a checkbox and the text 'What's up?'. Below the table, it says '1 question'.

Click the “What’s up?” question to edit it:

The screenshot shows the Django admin interface for editing a specific question. At the top, a blue breadcrumb bar reads 'Home > Polls > Questions > What's up?'. Below this, the page title is 'Change question'. In the top right corner, there is a button labeled 'HISTORY'. The form has two main sections. The first section is 'Question text:' with a text input field containing 'What's up?'. The second section is 'Date published:', which contains two sub-sections: 'Date:' with a date input field showing '2015-09-06' and a 'Today' button with a calendar icon; and 'Time:' with a time input field showing '21:16:22' and a 'Now' button with a clock icon. At the bottom of the form, there is a row of four buttons: 'Delete' (red), 'Save and add another' (blue), 'Save and continue editing' (blue), and 'SAVE' (blue).

Things to note here:

- 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.

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.

If the value of “Date published” doesn’t match the time when you created the question, it probably means you forgot to set the correct value for the **TIME_ZONE** setting. Change it, reload the page and check that the correct value appears.

➤ Overview:

In our poll application, we’ll have the following four views:

- 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).

➤ Writing views 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.

Your view can read records from a database, or not. It can use a template system such as Django's – or a third-party Python template system – or not. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is that **HttpResponse**. Or an exception.

Because it's convenient, let's use Django's own database API. 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:

polls/views.py

```
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)


# Leave the rest of the views (detail, results, vote) unchanged
```

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 **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**.

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

➤ **Customize your apps looks and feels:**

First, create a directory called **static** in your **polls** directory. Django will look for static files there, similarly to how Django finds templates inside **polls/templates/**.

Django's **STATICFILES_FINDERS** setting contains a list of finders that know how to discover static files from various sources. One of the defaults is **AppDirectoriesFinder** which looks for a "static" subdirectory in each of the **INSTALLED_APPS**, like the one in **polls** we just created. The admin site uses the same directory structure for its static files.

Within the **static** directory you have just created, create another directory called **polls** and within that create a file called **style.css**. In other words, your stylesheet should be at **polls/static/polls/style.css**. Because of how the **AppDirectoriesFinder** staticfile finder works, you can refer to this static file in Django as **polls/style.css**, similar to how you reference the path for templates.

- **Adding a background image:**

Next, we'll create a subdirectory for images. Create an images subdirectory in the polls/static/polls/ directory. Inside this directory, put an image called background.gif. In other words, put your image in polls/static/polls/images/background.gif.

Then, add to your stylesheet (polls/static/polls/style.css):

➤ **Customize admin form:**

By registering the **Question** model with **admin.site.register(Question)**, Django was able to construct a default form representation.

```
polls/admin.py

from django.contrib import admin

from .models import Choice, Question

class ChoiceInline(admin.TabularInline):
    model = Choice
    extra = 3

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date'], 'classes':
['collapse']}),
    ]
    inlines = [ChoiceInline]
    list_display = ('question_text', 'pub_date', 'was_published_recently')
    list_filter = ['pub_date']
    search_fields = ['question_text']

admin.site.register(Question, QuestionAdmin)
```

Now the question change list page looks like this:

Select question to change

Action: 0 of 1 selected

<input type="checkbox"/> QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/> What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

You can click on the column headers to sort by those values – except in the case of the **was_published_recently** header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for **was_published_recently** is, by default, the name of the method (with underscores replaced with spaces), and that each line contains the string representation of the output.

You can improve that by using the **display()** decorator on that method (in **polls/models.py**), as follows:

polls/models.py

```
from django.contrib import admin

class Question(models.Model):
    # ...

    @admin.display(
        boolean=True,
        ordering='pub_date',
        description='Published recently?',
    )
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

Edit your `polls/admin.py` file again and add an improvement to the **Question** change list page: filters using the `list_filter`. Add the following line to **QuestionAdmin**:

```
list_filter = ['pub_date']
```

That adds a “Filter” sidebar that lets people filter the change list by the `pub_date` field:

Home > Polls > Questions

Select question to change ADD QUESTION +

Action: Go 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	PUBLISHED RECENTLY?
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	✖

1 question

FILTER

By date published

- Any date
- Today
- Past 7 days
- This month
- This year

The type of filter displayed depends on the type of field you’re filtering on. Because `pub_date` is a **DateTimeField**, Django knows to give appropriate filter options: “Any date”, “Today”, “Past 7 days”, “This month”, “This year”.

This is shaping up well. Let’s add some search capability:

```
search_fields = ['question_text']
```

That adds a search box at the top of the change list. When somebody enters search terms, Django will search the `question_text` field. You can use as many fields as you’d like – although because it uses a **LIKE** query behind the scenes, limiting the number of search fields to a reasonable number will make it easier for your database to do the search.

- **Customize admin look and feel:**

Clearly, having “Django administration” at the top of each admin page is ridiculous. It’s just placeholder text.

You can change it, though, using Django’s template system. The Django admin is powered by Django itself, and its interfaces use Django’s own template system.

➤ Customizing your project's templates:

Create a **templates** directory in your project directory (the one that contains **manage.py**). Templates can live anywhere on your filesystem that Django can access. (Django runs as whatever user your server runs.) However, keeping your templates within the project is a good convention to follow.

Open your settings file (**mysite/settings.py**, remember) and add a **DIRS** option in the **TEMPLATES** setting:

mysite/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],
        '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',
            ],
        },
    ],
]
```

DIRS is a list of filesystem directories to check when loading Django templates; it's a search path.

Now create a directory called **admin** inside **templates**, and copy the template **admin/base_site.html** from within the default Django admin

template directory in the source code of Django itself (**django/contrib/admin/templates**) into that directory.

Then, edit the file and replace **{{ site_header|default:_('Django administration') }}** (including the curly braces) with your own site's name as you see fit. You should end up with a section of code like

```
{% block branding %}

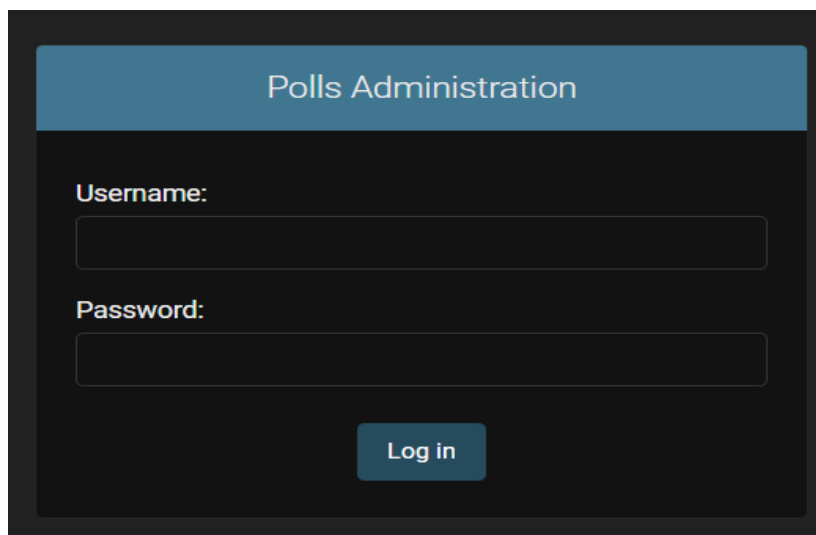
<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls
Administration</a></h1>

{% endblock %}
```

We use this approach to teach you how to override templates.

This template file contains lots of text like **{% block branding %}** and **{{ title }}**. The **{%}** and **{{** tags are part of Django's template language. When Django renders **admin/base_site.html**, this template language will be evaluated to produce the final HTML page.

6. Output

A screenshot of a web application's login page. At the top, there is a blue header bar with the text "Polls Administration" in white. Below the header, the page has a dark gray background. On the left side, there are two labels: "Username:" and "Password:", both in white. To the right of each label is a white rectangular input field. At the bottom center of the page, there is a blue button with the text "Log in" in white.

Polls Administration

WELCOME, ADMIN VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups

+ Add

Change

Users

+ Add

Change

POLLS

Questions

+ Add

Change

Recent actions

My actions

Whats up

Question

+ Which is your fav video game?

Question

+ Which games you like playing?

Question

+ What do you prefer?

Question

+ Are you Currently Working?

Question

+ Are you a Student?

Question

+ Where do you live?

Question

+ Which anime you like the most among these?

Question

Whats up

Question

What is your Age?

Question

- [Which is your fav video game?](#)
- [Which games you like playing?](#)
- [What do you prefer?](#)
- [Are you Currently Working?](#)
- [Are you a Student?](#)

Are you a Student?

☐ Yes

☐ No

Vote

7. Features

- It has an admin login page where you can login as admin/superuser.
- Admin has access to bunch of features like admin can block user for accessing site or grant users special permissions.
- Admin can also edit questions and choices, add more questions and choices and delete them.
- Admin can also review recent activities.
- Web app can be used to know about people opinions and interest by answering the questions.
- It doesn't require any login or personal information, anyone can use it.

8. Limitations

- It will take atleast a week to create this project with intermediate knowledge and experience in python language.
- Project is free of cost, everything used is available online for free to everyone.
- Risk involved are- admin username and password getting leaked or hacked which is bad as the hacker can access and edit information displayed on app, also hacker can ban or add users to access the site.
- It is only a multiple choice question answer web app where you can only answer questions with choices given in them.