- Link to login/register
- Link to leaderboard
- About
- Start game
- Add photos

Playing with API

- **Invoke the shell.**
  - **python manage.py shell**
  - Explore the database

```
>>> 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)>]>
```

- **Create __str__ methods for objects:**
  - In <app>/models.py create str functions for the objects.

```
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
```

  - **Run the shell again: (python manage.py shell)**

```
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()
```

Part 1

- ## Create a project in the current directory.
  - **django-admin startproject** <projectname>

- Inside your <projectname> folder:
  - **manage.py**: command line utility that lets you interact with Django projects in various ways.
  - **Inner <projectname>/ directory:** is the actual Python package for your project.
  - **<project>/__init__.py:** an empty file that tells python that is a directory should be considered a Python package.
  - **<project>/settings.py:** settings/configuration for this Django project.
  - **<project>/urls.py**: The URLs declarations for this Django project, a 'table of contents' of your site.
  - **<project>/wsgi.py**: an entry point for WSGI-compatible web servers to serve your project.

- ## Development Server:
  - cd into <projectname> directory.
  - Run **python manage.py runserver**
  - Should see:

```
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.

February 01, 2021 - 15:50:53
Django version 2.2, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

  - Extra:
    - Change port: `python manage.py runserver 8080`
    - Change servers IP: `python manage.py runserver 0:8000`

- ## Creating app:
  - Project vs Apps:

    What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app. A project is a collection of

configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.

- Make sure you're in the same directory as 'manage.py'.
- Run **python manage.py startapp <appname>**.
- This will create a <appname> directory that will house the application.

- Write your first View:

- Open ***<app>/views.py*** and insert the code:

```
from django.http import HttpResponse


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

- To call this we need to map it to a URL- we need a **URLconf**.
- **Create URLconf:**
  - Inside <app> directory.
  - Create file called **urls.py** and insert the code:

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),

]
```

  - Next, point the root URLconf at the **<app>.urls** module.
  - Go into **<project>/urls.py**, add an import for **django.urls.include** and insert an *include()* in urlpatterns list.

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

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

  - Verify it works with **python manage.py runserver**.
  - Open **http://localhost:8000/<project>/**

Part 2

- Database setup.

- Open **<project>/settings.py**, by default it uses SQLite.
- If you wish to change database:
  - **ENGINE**: 'django.db.backends.<database name> (eg .sqlite3, .mysql)

- **NAME**: name of your database, in SQLite the database is a file on your computer, in this case, the **NAME** should be the absolute path of this file.

- Migrate:

- In **<project>/settings.py,** INSTALLED_APPs holds the name of all Django applications that are activated in this Django instance.
- Default applications:
  - `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.
- We need to create tables in the database for these applications before we can use them: **python manage.py migrate**.
- Migrate looks at INSTALLED_APPS and creates any necessary database tables according to database settings in <project>/settings.py.

- Creating Models:

- Models = essentially your database layout, with metadata.
- In **<project>/models.py** we can create models as python classes.

```
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)
```

- Each model is represented by a class that inherits **django.db.models.Model**. Each class variable is a database field in the model.

- Activation Models:

- This model code gives Django a lot of information. That django can use to:
  - *Create database schema* (CREATE TABLE statements) *for this app.*
  - *Creates a Python database-access API for accessing **Question** and **Choice** objects.*
- But first we must tell our project that <app> is installed.
- To include the app in our project, we need to add to **INSTALLED_APPS**.

- <app>Config is in the **<app>/apps.py**:
  - so its path is **"<app>.apps.<app>Config"**
- Go to **<project>/settings.py** and add this path.

```
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 <app>.
- Run **python manage.py makemigrations** to create migrations for the changes.
- Run **python manage.py migrate** to apply changes to database.

- Creating Admin User:

- Create a user who can login to the admin site.
  - **Python manage.py createsuperuser**
  - Enter your username and details:
    - **Username: admin**
    - **Email address:** admin@example.com
  - Final step is to enter your password, you will be asked twice.
    - **Password: ************
    - **Password (again): ************
    - Superuser created successfully
- Start the development server
  - **Python manage.py runserver**
  - Open http://127.0.0.1:8000/admin/ to see admin login

- Make poll app modifiable in Admin

- Our app not on the admin index page?
- We need to tell the admin that Question objects have an admin interface.
  - Open **<app>/admin.py** and enter:

```
from django.contrib import admin
from .models import Question

admin.site.register(Question)
```

- Overview

- A **view** is a type of 'web page' in your django application that serves a specific function and has a specific template.
- 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.

- In our poll application, we have the following four views:
  1) Question “index” page – displays the latest few questions.
  2) Question “detail” page – displays a question text, with no results but with a form to vote.
  3) Question “results” page – displays results for a particular question.
  4) Vote action – handles voting for a particular choice in a particular question.

- Writing more views

- Add more views to **<app>/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)
```

  - These are a bit different as they take arguments. (input)
- Wire these new views into the **<app>.urls** by adding paths():

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

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'),

    ]
```

- Run server and enter URL:

- http://127.0.0.1:8000/polls/34/ - run the detail() method as this url path is mapped to views.detail.
- http://127.0.0.1:8000/polls/34/results/ - run the results method.
- http://127.0.0.1:8000/polls/34/vote/ - run the vote method in views.py.#

- When someone requests a page from your site, eg. http://127.0.0.1:8000/polls/34/ , Django will load the **<project>.urls** which will then search the urlpatterns and find the path pointing to **<app>/urls**. After finding the match at **'<app>/'** in **<project>.urls** it strips off ***'http://127.0.0.1:8000/polls/'*** leaving just ***'/34/'*** which is sent to **<app>.urls** where it searches *urlpatterns* for a match. It will find **'<int:question_id>/'** that calls the detail() function in **<app>.views.**
- **So the <project>.urls points to the <app>.urls that then calls the <app>.views function to show the webpage.**

- 1) Write views that actually do something:

- Each view responsible for returning two things:
  - **HttpResponse** object containing *content* for requested page.
  - Raising an **exception** such as *Http404*.
- Create a new index() view that displays the latest 5 poll questions.
  - Go to **<app>/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
```

  - Problem!! Pages design is hard-coded in this view, if you wish to change the design, create a **template**.
    - Create directory **'templates'** in **<app>** directory.
    - The project's ***TEMPLATES*** in **<project>/settings.py** describes how Django will load and render templates.
    - Within the **'templates'** directory created in step 1, create another directory called '<**appname**>'.
    - Inside '<**appname**>' create a file called **index.html**. (template at <app>/templates/<app>/index.html)
    - You can refer to this file within Django as **'<app>/index.html'**.
    - Inside <app>/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 update our *index view* in **<app>/views.py** to use this:
  - This code loads the template (<app>/index.html) and passes it a context.
  - The context is a dictionary mapping template variable names to Python.

```
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))
```

- 2) Shortcut for above ^^ (render()):

- Load a template, fill a context and return an HttpResponse object with the result of the rendered template.
- Django provides a shortcut - below is the rewritten **<app>/views.py:**

```
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.
- 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.

- **1) Raising a 404 error:**
  - Go to **<app>/views.py** detail function:

```
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 view raises the Http404 if a question with the requested ID doesnt exist.
  - Inside '**<app>/detail.html**':

```
{{ question }}
```

- **2) Shortcut for above ^^ (get_object_or_404()):**
  - Within the <app>/views.py detail function:

```
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})
```

- **Removing hardcoded URLs in templates:**
  - Inside **<app>/index.html** template the link as partially hardcoded:
  - Hardcoded:

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

    {{ question.question_text }}

</a></li>
```

  - Not hardcoded:

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

  - To edit the url for detail or for any other view, go to **<app>/urls.py** and edit the path for the given view function.

- ## Namespacing URL names:
  - When you have more than one app in a project, how does Django differentiate the names when using the url function as seen in the above section^^.
  - Solution !! = add namespaces to your URLconf.
  - In the **<app>/urls.py** add on app_name to the application namespace:

```
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    ……
    ……
    ….
]
```

  - Now set **<app>/index.html** template to (polls:detail):

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

    {{ question.question_text }}

</a></li>
```

- ## Write simple Form:
  - Update **<app>/detail.html** to include a HTML form element.

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}
    <p><strong>{{ error_message }}</strong></p>
{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
    {% csrf_token %}
    {% for choice in question.choice_set.all %}
       <input type="radio" name="choice" id="choice{{ forloop.counter }}"
    value="{{ choice.id }}">
       <label for="choice{{ forloop.counter }}">{{ choice.choice_text
    }}</label><br>
    {% endfor %}
    <input type="submit" value="Vote">
</form>
```

  - The `value` of each radio button is the associated question choice's ID
  - The `name` of each radio button is `"choice"`.

        - When someone submits the form, it'll send the POST data **`choice=#`** where # is the ID of the selected choice.
    - Using **`method="post"`** (as opposed to **`method="get"`**) is very important, because the act of submitting this form will alter data server-side.
    - **`forloop.counter`** indicates how many times the **`for`** tag has gone through its loop
    - In short, all POST forms that are targeted at internal URLs should use the **`{% csrf_token %}`** template tag.
- Create a full implementation of the **vote** function in **<app>/views.py.**

```
from django.http import HttpResponse, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse

from .models import Choice, Question
# ...
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice =
question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after
successfully dealing
        # with POST data. This prevents data from being posted
twice if a
        # user hits the Back button.

     return HttpResponseRedirect(reverse('polls:results',
args=(question.id,)))
```

- **Theory ^^^:**
    - **`request.POST`** is a dictionary-like object that lets you access submitted data by key name. In this case, **`request.POST['choice']`** returns the ID of the selected choice, as a string.
    - **`request.POST['choice']`** will raise **`KeyError`** if **`choice`** wasn't provided in POST data.
    - We are using the **`reverse()`** function in the **`HttpResponseRedirect`** constructor in this example. This function helps avoid having to hardcode a URL in the view function.

- After somebody votes in a question, the vote function view redirect to the results page for the question (in **<app>/views.py**) :

```
from django.shortcuts import get_object_or_404, render

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