1. The file “polls/views.py” is where we define how data is displayed. In this case, a simple view named “index” is created that returns a basic HTTP response: "Hello, world. You're at the polls index." And then we map Views to URLs. A new file “urls.py” is created inside the `polls` directory to define URLs specific to the “polls” app. Inside “polls/urls.py”, a URL pattern is defined that maps the root URL of the app to the `index` view. This means when someone visits the root URL of the `polls` app, they'll see the "Hello, world. You're at the polls index." message. The main project (`mysite`) has its own `urls.py`. This is where we include the URL configurations of all apps in the project.

A white rectangular object with a white border

Description automatically generated with medium confidence

1. The `mysite/settings.py` file contains settings for the Django project, including database configurations. Then we create Models`Question` and `Choice` in the file polls/models.py. After defining models, we need to inform Django about them. This is done by adding the app containing the models to the `INSTALLED\_APPS` setting in `mysite/settings.py`.Once the app is added, we do migration for these new models. Migrations are Django's way of tracking changes to rmodels. After creating migrations, we run `python manage.py migrate` to apply these changes to the database.

We can use interactive Python shell to play around with the database API. This is accessed using `python manage.py shell`. We can create and save model instances, filter and query the database, access related objects and delete objects.

1. Django Admin interface is a built-in tool that allows us to manage content for our apps without having to build a custom management interface.

A screenshot of a computer

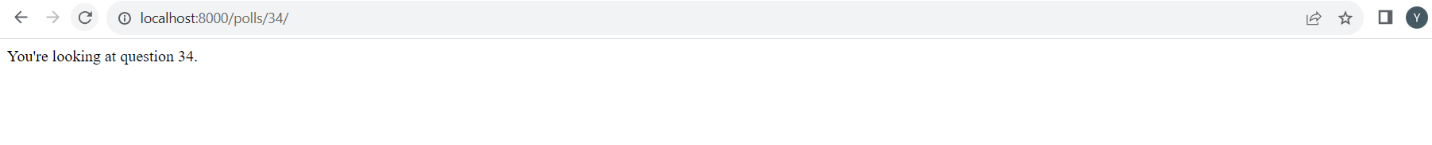
Description automatically generated

A screenshot of a computer

Description automatically generated

1. Three new view functions (detail, results, and vote) are added to the polls/views.py file.

When one of these views is called, it will return a simple HTTP response indicating which question ID the user is interacting with. Then The polls/urls.py file is updated to include paths for these new views.



A white rectangular object with a black border

Description automatically generated with medium confidence

1. A view in Django should either return an `HttpResponse` object with the content for the requested page or raise an exception (like `Http404` for not found). We added the `index` view to fetch the latest 5 poll questions from the database and display them. To separate the design (HTML) from the logic (Python code), Django's template system is introduced.A new directory structure is set up for templates (`polls/templates/polls/`). This structure ensures templates from different apps don't conflict with each other. A template (`index.html`) is created to display the list of questions or a message if no polls are available. We have a shortcut function, `render()`, to simplify the common pattern of loading a template, filling a context, and returning an `HttpResponse`.Also, instead of manually checking if an object exists and then raising a Http404 exception if it doesn't, we use the shortcut function called get\_object\_or\_404()to fetch an object from the database based on the provided arguments. If the object doesn't exist, it automatically raises a Http404 exception.
2. In this session, we know how to create a simple voting form in Django, handle its submission, and display the results. The vote view retrieves the selected choice using the request.If a choice isn't selected, an error message is displayed. If a choice is selected, its vote count is incremented.

The results view is introduced to display the voting results. It's similar to the detail view but uses a different template. The polls/results.html template displays the question, each choice, and the number of votes each choice received.

A white background with black text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

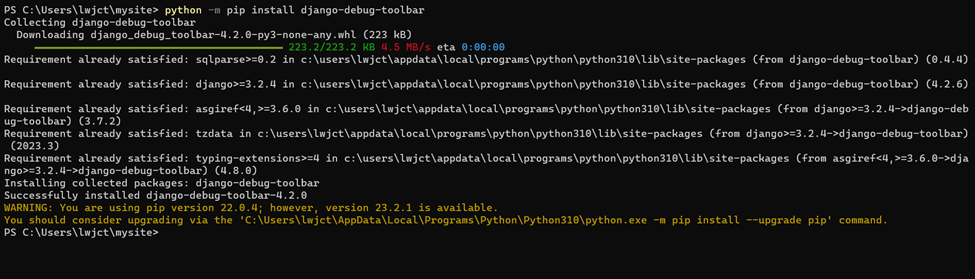
1. We can add automatic tests to the app. Tests provide a safety net, ensuring that as we add or modify code, we don't introduce new bugs

A computer screen with white text

Description automatically generated

A screen shot of a computer

Description automatically generated



1. We can customize the appearance of the Django app by adding static files, specifically CSS files, to style the app.

A screenshot of a computer screen

Description automatically generated

1. By default, Django provides a basic form representation for models registered with the admin. However, we can customize this form. We can reorder the fields on the edit form by creating a custom admin class (QuestionAdmin) and specifying the desired order in the fields attribute. We can also customize the Admin Change List and the Admin Look and Feel.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

