
```

1 class IndexView(generic.ListView):
2     template_name = 'polls/index.html'
3     context_object_name = 'latest_question_list'
4
5     def get_queryset(self):
6         # Return the last five published questions.
7         return Question.objects.order_by('-pub_date')[:5]

```

The queryset for this view is the five most recently published objects in the Question table. It then uses the template located at “templates/polls/index.html” - the root folder “templates” is implicit. Here is that template:

```

1 {% if latest_question_list %}
2     <ul>
3         {% for question in latest_question_list %}
4             <li><a href="/polls/{{ question.id }}">{{ question.
               question_text }}</a></li>
5         {% endfor %}
6     </ul>
7 {% else %}
8     <p>No polls are available.</p>
9 {% endif %}

```

Unless “latest_question_list” is empty, this will output a list of the five most recent questions, showing their names and linking to that question’s page. The URLs are defined in the aptly names urls.py.

11 Stage 1: Tasks

Given the foundations laid by my work on the tutorial, I think the best place to start development will be with task management, especially given that this is the primary important function of my product.

I need to make:

- A task model, to store that data about each task
- A task index, listing todo and done tasks separately, and linking through to view each task in more detail
- A task detail view, showing all the information about a given task and allowing editing/deletion
- A form for adding and editing tasks
- A task deletion view

11.1 Task Model

Logically, it makes sense to create the model first, as we can't even begin to think about how we will display a task until we know their attributes. Since I've already outlined the attributes and methods for it in my design, creating the model will be quite simple.

```
1 class Task(models.Model):
2     LOW = 1
3     MED = 2
4     HIGH = 3
5     PRIORITY_LIST = [
6         (LOW, "Low"),
7         (MED, "Normal"),
8         (HIGH, "High"),
9     ]
10    title = models.CharField(max_length=200)
11    description = models.CharField(max_length=1000)
12    due_date = models.DateField("due date")
13    due_time = models.TimeField("due time", default="00:00")
14    time_estimate = models.DurationField("time estimate")
15    priority = models.IntegerField("priority", choices=PRIORITY_LIST
16                                   , default=2)
17    done = models.BooleanField(default=False)
18
19    def __str__(self):
20        return self.title
21
22    def is_overdue(self):
23        return self.due_date <= timezone.now()
24
25    def mark_done(self):
26        self.done = True
27
28    def mark_todo(self):
29        self.done = False
30
31    def get_absolute_url(self):
32        return f"/task/{self.id}/"
```

Most of this is self-explanatory, and in line with my design specification. Some notable features are the `PRIORITY_LIST`, which is an attribute of `Task` but not model field - as such it is not present in the database. `PRIORITY_LIST` serves the `priority` field, which is a multiple choice field. Choice fields require a list of tuples, with each tuple containing a value which is actually stored in the database, and a human readable name for that value - what it represents. In this instance the values are `LOW`, `MED` and `HIGH`, each being a variable corresponding to the values of 1, 2 and 3 respectively, and each with a corresponding readable name. This method seems somewhat convoluted, but is Django's recommended way of handling choice fields and has a pleasant, clean feel, so I decided not to take a shortcut route.

There is also the `get_absolute_url` method, this is so it is always easy to access the corresponding detail view of any given task, which is located at `/task/[id]`, where `id` is the task's id. There are various ways to do this, here I'm using an "f-string", or formatted string, so `self.id` will be evaluated to the id of the task.

11.2 Task Index

This will consist of two parts: a view defining the data to be displayed, and an HTML template defining the layout.

```
1 class IndexView(ListView):
2     template_name = 'tasks/index.html'
3     context_object_name = 'task_list'
4
5     def get_queryset(self):
6         return Task.objects.order_by('due_date')
7
8     def get_context_data(self, **kwargs):
9         context = super(IndexView, self).get_context_data(**kwargs)
10        context['todo_tasks'] = Task.objects.filter(
11            done=False).order_by("due_date")
12        context['done_tasks'] = Task.objects.filter(
13            done=True).order_by("due_date")
14        return context
```

`IndexView` is a subclass of `ListView`, meaning it expects a list as it's queryset - in other words, it retrieves a list of objects from the database, not just one. Therefore the `get_queryset` returns a list of all the `Task` objects, ordered by when they are due. This data will be referred to as "task_list" in the HTML template, as specified by the `context_object_name` attribute.

The `get_context_data` function serves to segregate the data: I split it into the tasks which are todo and which are done, so it will be easy to show them in two separate lists.

The HTML template is specified at `templates/tasks/task_detail.html` (as a relative path from `models.py`), as indicated by the attribute `template_name`. Having the `tasks/` subfolder is somewhat redundant, this is just following Django's recommended directory layout, which would help if I ever expanded the project to need more complex namespacing. The template for this view is somewhat lengthy, so I won't put it all here; in short, It will show two lists, one of tasks which are todo, and one of tasks which are already done. Each task will show its name, doubling as a link to the detail view of the task. It will also have a button to quickly toggle the tasks todo/done status.

Here is a snippet of that part of the code:

```
1 <li><a href="/task/{{ task.id }}/">{{ task.title }}</a>
2     {% if task.done %}
3     <form action="{% url 'tasks:mark_as_todo' task.id %}" method="
4         post">
5         {% csrf_token %}
```

```
5         <input type="hidden" name="link" value="{ request.path }">
6         <input type="submit", value="Mark as todo">
7     </form>
8     {% else %}
9     <form action="{% url 'tasks:mark_as_done' task.id %}" method="
10         post">
11         {% csrf_token %}
12         <input type="hidden" name="link" value="{ request.path }">
13         <input type="submit", value="Mark as done">
14     </form>
15     {% endif %}
16 </li>
```

Most of this is fairly standard: a list item, with a hyperlink displaying the title of the task and linking to it's detail view, and a form with a button to either mark as todo or done depending on the task's status. Of note however are the dynamic functionalities provided by Django: variables provided from the queryset are surrounded by double braces, which Django replaces with the actual values when the page is visited, and this is used for the `task.id` and `task.title` here. Conditionals and loops are also available, I've used an if/else statement here and, this snippet is actually inside a for-loop, so it creates list items for all the tasks.

The form has a `csrf_token`: CSRF here stands for cross-site request forgery. Although security is far from a huge concern in my project, Django requires this to be used by all forms. and is certainly a best practice. Finally, note that the URL for the form is not a hard link, but instead makes use of the namespacing provided by Django. `mark_as_done` and `mark_as_todo` are specified in my `urls.py` file, as linking to views of the same name, which run the relevant code to the mark the task as done or todo. This is just one line of course, calling the relevant method on the Task object provided by the form. Figure 10 shows what we've made so far.

Tasks

Todo:

- [Something I need to do](#)
- [Something else I need to do...](#)

Done:

- [Something which I've done](#)

Figure 10: The task index

11.3 Task Detail

Currently, the hyperlinks on the index page produce a 404 error, because I haven't actually created the detail view for the tasks yet. It's fairly simple, it just needs to show the user all the information about a specific task.

```
1 class TaskDetail(DetailView):
2     model = Task
3     template_name = "tasks/task_detail.html"
4
5     def task_done(self):
6         Task.mark_done()
7
8     def task_todo(self):
```

`TaskDetail` is a subclass of `DetailView`. With the model specified as `Task`, this means that whenever the relevant URL is accessed, which I've specified `tasks/[task id]`, it will get the data for the task of that id. It also has two simple methods to be able to mark the task as todo or done.

The HTML template is also simple, just displaying the data about the task along with a button to either mark as todo or done, as appropriate, a button to edit the task, and a button to delete the task. The latter two are not yet functional, of course.

[<Back to tasks](#)

Something I need to do

- Due on March 8, 2020 at 4:14 p.m.
- Time estimate: 0:20:00
- Priority level: 2

Do this!

Still to-do.

Mark as done

Edit

Delete

Figure 11: Detail view of a task

11.4 Task creation/editing/deletion

Django can automatically create basic forms for creating and updating model objects. One just needs to specify what model a form is operating on, what attributes should be available to alter, and any specific widgets to be used for entering data for each field. Then it will retrieve the values from the fields in the form with the matching name, and create a new object or change an existing one to match the input.

```
1 class TaskCreate(CreateView):
2     model = Task
3     fields = [
4         "title",
5         "description",
6         "due_date",
7         "due_time",
8         "time_estimate",
```

```

9         "priority",
10     ]
11     due_date = forms.DateField(widget=forms.SelectDateWidget(attrs={
12         "type": "date"}))
13     due_time = forms.TimeField(widget=forms.TimeInput(attrs={"type":
14         "time"}))

```

Django will default to using the template at `[model]_form.html`, and I decided not to alter that, which is why no template is specified here. The template simply consists of a single form with input fields for each of the attributes.

The `TaskUpdate` view is overwhelmingly similar, with the simple addition that each field in the template has a default value, being the current value of the relevant attribute. See figure 12.

Add a new task:

Title:

Description:

Due date:

Due time:

Time estimate:



Priority level:
  
 (Higher = more important)

Figure 12: Detail view of a task

The deletion view is nothing special, the only notable feature is how it redirects the user back to the index, as going back the previous page wouldn't work since that would be the detail view of a task that was just deleted.

```
1 class TaskDelete(DeleteView):  
2     model = Task  
3     success_url = reverse_lazy("tasks:index")
```

Likewise, the HTML template for this view is just a form asking the user to confirm the deletion, with a button to do so.

At this point, it is possible to create and manage various tasks, view them together or individually in more detail, mark them as todo or done, and delete them - all the functions of a basic todo list app. The next step is to add similar abilities for events and recurring events (routines). This should mostly be straightforward, as they will be in many ways like tasks, just less dynamic, and having a fixed start and end time. As of yet tasks have no location in time - they have a due date, but no date or time specified in which they should actually be completed. Once I've added what is essentially calendaring functionality with events and routines, I'll write scheduler, which will be responsible for giving tasks that anchoring in time.