6

Tracking User Actions

In the previous chapter, you implemented AJAX views into your project using jQuery and built a JavaScript bookmarklet to share content from other websites on your platform.

In this chapter, you will learn how to build a follow system and create a user activity stream. You will also discover how Django signals work and integrate Redis's fast I/O storage into your project to store item views.

This chapter will cover the following points:

- Building a follow system
- Creating many-to-many relationships with an intermediary model
- Creating an activity stream application
- Adding generic relations to models
- Optimizing QuerySets for related objects
- Using signals for denormalizing counts
- Storing item views in Redis

Building a follow system

Let's build a follow system in your project. This means that your users will be able to follow each other and track what other users share on the platform. The relationship between users is a many-to-many relationship: a user can follow multiple users and they, in turn, can be followed by multiple users.

Creating many-to-many relationships with an intermediary model

In previous chapters, you created many-to-many relationships by adding the `ManyToManyField` to one of the related models and letting Django create the database table for the relationship. This is suitable for most cases, but sometimes you may need to create an intermediary model for the relationship. Creating an intermediary model is necessary when you want to store additional information for the relationship, for example, the date when the relationship was created, or a field that describes the nature of the relationship.

Let's create an intermediary model to build relationships between users. There are two reasons for using an intermediary model:

- You are using the `User` model provided by Django and you want to avoid altering it
- You want to store the time when the relationship was created

Edit the `models.py` file of your `account` application and add the following code to it:

```
class Contact(models.Model):
    user_from = models.ForeignKey('auth.User',
                                  related_name='rel_from_set',
                                  on_delete=models.CASCADE)
    user_to = models.ForeignKey('auth.User',
                                related_name='rel_to_set',
                                on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)

    class Meta:
        ordering = ('-created',)

    def __str__(self):
        return f'{self.user_from} follows {self.user_to}'
```

The preceding code shows the `Contact` model that you will use for user relationships. It contains the following fields:

- `user_from`: A `ForeignKey` for the user who creates the relationship
- `user_to`: A `ForeignKey` for the user being followed
- `created`: A `DateTimeField` field with `auto_now_add=True` to store the time when the relationship was created

A database index is automatically created on the ForeignKey fields. You use db_index=True to create a database index for the created field. This will improve query performance when ordering QuerySets by this field.

Using the ORM, you could create a relationship for a user, user1, following another user, user2, like this:

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

The related managers, rel_from_set and rel_to_set, will return a QuerySet for the Contact model. In order to access the end side of the relationship from the User model, it would be desirable for User to contain a ManyToManyField, as follows:

```
following = models.ManyToManyField('self',
                                   through=Contact,
                                   related_name='followers',
                                   symmetrical=False)
```

In the preceding example, you tell Django to use your custom intermediary model for the relationship by adding through=Contact to the ManyToManyField. This is a many-to-many relationship from the User model to itself; you refer to 'self' in the ManyToManyField field to create a relationship to the same model.

When you need additional fields in a many-to-many relationship, create a custom model with a ForeignKey for each side of the relationship. Add a ManyToManyField in one of the related models and indicate to Django that your intermediary model should be used by including it in the through parameter.

If the User model was part of your application, you could add the previous field to the model. However, you can't alter the User class directly because it belongs to the django.contrib.auth application. Let's take a slightly different approach by adding this field dynamically to the User model.

Edit the models.py file of the account application and add the following lines:

```
from django.contrib.auth import get_user_model

# Add following field to User dynamically
user_model = get_user_model()
user_model.add_to_class('following',
                        models.ManyToManyField('self',
                            through=Contact,
                            related_name='followers',
                            symmetrical=False))
```

In the preceding code, you retrieve the user model by using the generic function `get_user_model()`, which is provided by Django. You use the `add_to_class()` method of Django models to monkey patch the `User` model. Be aware that using `add_to_class()` is not the recommended way of adding fields to models. However, you take advantage of using it in this case to avoid creating a custom user model, keeping all the advantages of Django's built-in `User` model.

You also simplify the way that you retrieve related objects using the Django ORM with `user.followers.all()` and `user.following.all()`. You use the intermediary `Contact` model and avoid complex queries that would involve additional database joins, as would have been the case had you defined the relationship in your custom `Profile` model. The table for this many-to-many relationship will be created using the `Contact` model. Thus, the `ManyToManyField`, added dynamically, will not imply any database changes for the Django `User` model.

Keep in mind that, in most cases, it is preferable to add fields to the `Profile` model you created before, instead of monkey patching the `User` model. Ideally, you shouldn't alter the existing Django `User` model. Django allows you to use custom user models. If you want to use your custom user model, take a look at the documentation at `https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#specifying-a-custom-user-model`.

Note that the relationship includes `symmetrical=False`. When you define a `ManyToManyField` in the model creating a relationship with itself, Django forces the relationship to be symmetrical. In this case, you are setting `symmetrical=False` to define a non-symmetrical relationship (if I follow you, it doesn't mean that you automatically follow me).

When you use an intermediary model for many-to-many relationships, some of the related manager's methods are disabled, such as `add()`, `create()`, or `remove()`. You need to create or delete instances of the intermediary model instead.

Run the following command to generate the initial migrations for the `account` application:

```
python manage.py makemigrations account
```

You will obtain the following output:

```
Migrations for 'account':
  account/migrations/0002_contact.py
    - Create model Contact
```

Now, run the following command to sync the application with the database:

```
python manage.py migrate account
```

You should see an output that includes the following line:

```
Applying account.0002_contact... OK
```

The `Contact` model is now synced to the database, and you are able to create relationships between users. However, your site doesn't offer a way to browse users or see a particular user's profile yet. Let's build list and detail views for the `User` model.

Creating list and detail views for user profiles

Open the `views.py` file of the `account` application and add the following code to it:

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User

@login_required
def user_list(request):
    users = User.objects.filter(is_active=True)
    return render(request,
                  'account/user/list.html',
                  {'section': 'people',
                   'users': users})

@login_required
def user_detail(request, username):
    user = get_object_or_404(User,
                             username=username,
                             is_active=True)
    return render(request,
                  'account/user/detail.html',
                  {'section': 'people',
                   'user': user})
```

These are simple list and detail views for `User` objects. The `user_list` view gets all active users. The Django `User` model contains an `is_active` flag to designate whether the user account is considered active. You filter the query by `is_active=True` to return only active users. This view returns all results, but you can improve it by adding pagination in the same way as you did for the `image_list` view.

The `user_detail` view uses the `get_object_or_404()` shortcut to retrieve the active user with the given username. The view returns an HTTP `404` response if no active user with the given username is found.

Edit the `urls.py` file of the `account` application, and add a URL pattern for each view, as follows:

```
urlpatterns = [
    # ...
    path('users/', views.user_list, name='user_list'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

You will use the `user_detail` URL pattern to generate the canonical URL for users. You have already defined a `get_absolute_url()` method in a model to return the canonical URL for each object. Another way to specify the URL for a model is by adding the `ABSOLUTE_URL_OVERRIDES` setting to your project.

Edit the `settings.py` file of your project and add the following code to it:

```
from django.urls import reverse_lazy

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])
}
```

Django adds a `get_absolute_url()` method dynamically to any models that appear in the `ABSOLUTE_URL_OVERRIDES` setting. This method returns the corresponding URL for the given model specified in the setting. You return the `user_detail` URL for the given user. Now, you can use `get_absolute_url()` on a `User` instance to retrieve its corresponding URL.

Open the Python shell with the `python manage.py shell` command and run the following code to test it:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.latest('id')
>>> str(user.get_absolute_url())
'/account/users/ellington/'
```

The returned URL is as expected.

You will need to create templates for the views that you just built. Add the following directory and files to the `templates/account/` directory of the `account` application:

```
/user/
    detail.html
    list.html
```

Edit the `account/user/list.html` template and add the following code to it:

```
{% extends "base.html" %}
{% load thumbnail %}

{% block title %}People{% endblock %}

{% block content %}
  <h1>People</h1>
  <div id="people-list">
    {% for user in users %}
      <div class="user">
        <a href="{{ user.get_absolute_url }}">
          <img src="{% thumbnail user.profile.photo 180x180 %}">
        </a>
        <div class="info">
          <a href="{{ user.get_absolute_url }}" class="title">
            {{ user.get_full_name }}
          </a>
        </div>
      </div>
    {% endfor %}
  </div>
{% endblock %}
```

The preceding template allows you to list all the active users on the site. You iterate over the given users and use the `{% thumbnail %}` template tag from `easy-thumbnails` to generate profile image thumbnails.

Open the `base.html` template of your project and include the `user_list` URL in the `href` attribute of the following menu item:

```
<li {% if section == "people" %}class="selected"{% endif %}>
  <a href="{% url "user_list" %}">People</a>
</li>
```

Start the development server with the `python manage.py runserver` command and open `http://127.0.0.1:8000/account/users/` in your browser. You should see a list of users like the following one:

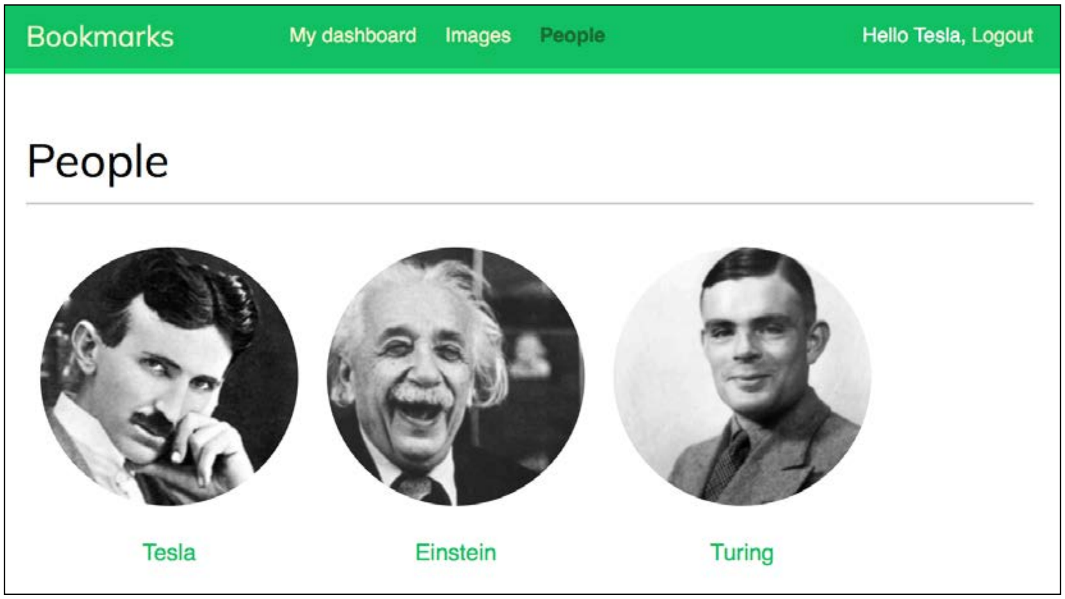


Figure 6.1: The user list page with profile image thumbnails

Remember that if you have any difficulty generating thumbnails, you can add `THUMBNAIL_DEBUG = True` to your `settings.py` file in order to obtain debug information in the shell.

Edit the `account/user/detail.html` template of the `account` application and add the following code to it:

```
{% extends "base.html" %}
{% load thumbnail %}

{% block title %}{{ user.get_full_name }}{% endblock %}

{% block content %}
  <h1>{{ user.get_full_name }}</h1>
```

```
    <div class="profile-info">
      <img src="{% thumbnail user.profile.photo 180x180 %}" class="user-
  detail">
    </div>
    {% with total_followers=user.followers.count %}
      <span class="count">
        <span class="total">{{ total_followers }}</span>
        follower{{ total_followers|pluralize }}
      </span>
      <a href="#" data-id="{{ user.id }}" data-action="{% if request.
  user in user.followers.all %}un{% endif %}follow" class="follow
  button">
        {% if request.user not in user.followers.all %}
          Follow
        {% else %}
          Unfollow
        {% endif %}
      </a>
      <div id="image-list" class="image-container">
        {% include "images/image/list_ajax.html" with images=user.
  images_created.all %}
      </div>
    {% endwith %}
  {% endblock %}
```

Make sure that no template tag is split into multiple lines; Django doesn't support multiple line tags.

In the `detail` template, you display the user profile and use the `{% thumbnail %}` template tag to display the profile image. You show the total number of followers and a link to follow or unfollow the user. You perform an AJAX request to follow/unfollow a particular user. You add `data-id` and `data-action` attributes to the `<a>` HTML element, including the user ID and the initial action to perform when the link element is clicked – `follow` or `unfollow`, which depends on the user requesting the page being a follower of this other user or not, as the case may be. You display the images bookmarked by the user, including the `images/image/list_ajax.html` template.

Open your browser again and click on a user who has bookmarked some images. You will see profile details, as follows:

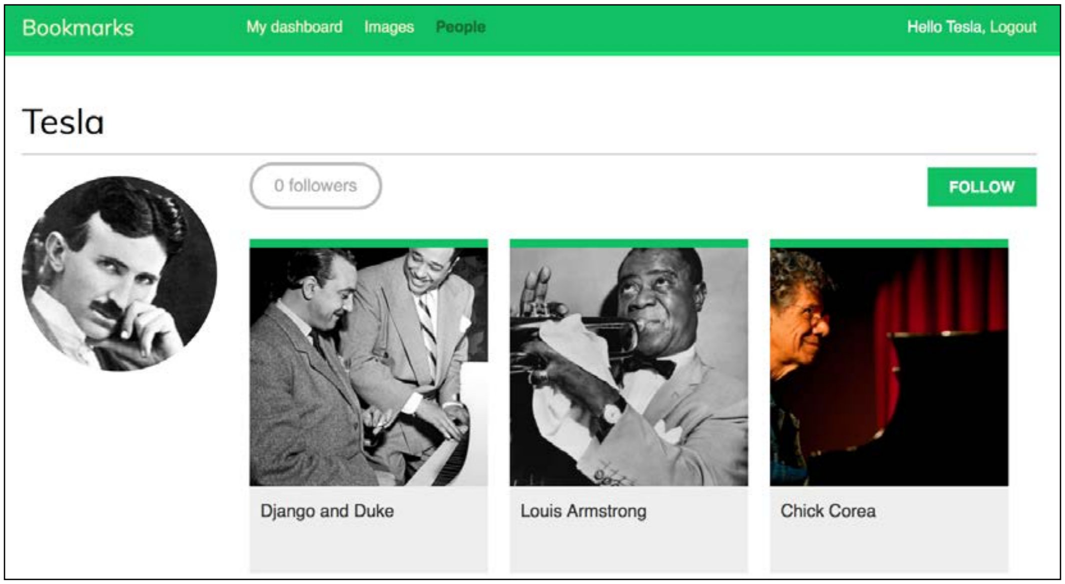


Figure 6.2: The user detail page

Building an AJAX view to follow users

Let's create a simple view to follow/unfollow a user using AJAX. Edit the `views.py` file of the `account` application and add the following code to it:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from common.decorators import ajax_required
from .models import Contact

@ajax_required
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                        user_from=request.user,
                        user_to=user)
                else:
                    Contact.objects.filter(user_from=request.user,
                                           user_to=user).delete()
                return JsonResponse({'status':'ok'})
            except User.DoesNotExist:
                return JsonResponse({'status':'error'})
        return JsonResponse({'status':'error'})
```

The `user_follow` view is quite similar to the `image_like` view that you created before. Since you are using a custom intermediary model for the user's many-to-many relationship, the default `add()` and `remove()` methods of the automatic manager of `ManyToManyField` are not available. You use the intermediary `Contact` model to create or delete user relationships.

Edit the `urls.py` file of the `account` application and add the following URL pattern to it:

```
path('users/follow/', views.user_follow, name='user_follow'),
```

Ensure that you place the preceding pattern before the `user_detail` URL pattern. Otherwise, any requests to `/users/follow/` will match the regular expression of the `user_detail` pattern and that view will be executed instead. Remember that in every HTTP request, Django checks the requested URL against each pattern in order of appearance and stops at the first match.

Edit the `user/detail.html` template of the `account` application and append the following code to it:

```
{% block domready %}
  $('a.follow').click(function(e){
    e.preventDefault();
    $.post('{% url "user_follow" %}',
      {
        id: $(this).data('id'),
        action: $(this).data('action')
      },
      function(data){
        if (data['status'] == 'ok') {
          var previous_action = $('a.follow').data('action');

          // toggle data-action
          $('a.follow').data('action',
            previous_action == 'follow' ? 'unfollow' : 'follow');
          // toggle link text
```

```
                $('a.follow').text(
                  previous_action == 'follow' ? 'Unfollow' : 'Follow');

                // update total followers
                var previous_followers = parseInt(
                  $('span.count .total').text());
                $('span.count .total').text(previous_action == 'follow' ?
                previous_followers + 1 : previous_followers - 1);
              }
            }
          );
        });
    {% endblock %}
```

The preceding code is the JavaScript code to perform the AJAX request to follow or unfollow a particular user and also to toggle the follow/unfollow link. You use jQuery to perform the AJAX request and set both the `data-action` attribute and the text of the HTML `<a>` element based on its previous value. When the AJAX action is performed, you also update the total followers count displayed on the page.

Open the user detail page of an existing user and click on the **FOLLOW** link to test the functionality you just built. You will see that the followers count is increased:

Figure 6.3: The followers count and follow/unfollow button

Building a generic activity stream application

Many social websites display an activity stream to their users so that they can track what other users do on the platform. An activity stream is a list of recent activities performed by a user or a group of users. For example, Facebook's News Feed is an activity stream. Sample actions can be *user X bookmarked image Y* or *user X is now following user Y*.

You are going to build an activity stream application so that every user can see the recent interactions of the users they follow. To do so, you will need a model to save the actions performed by users on the website and a simple way to add actions to the feed.

Create a new application named `actions` inside your project with the following command:

```
python manage.py startapp actions
```

Add the new application to `INSTALLED_APPS` in the `settings.py` file of your project to activate the application in your project:

```
INSTALLED_APPS = [
    # ...
    'actions.apps.ActionsConfig',
]
```

Edit the `models.py` file of the `actions` application and add the following code to it:

```
from django.db import models

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                             related_name='actions',
                             db_index=True,
                             on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)

    class Meta:
        ordering = ('-created',)
```

The preceding code shows the `Action` model that will be used to store user activities. The fields of this model are as follows:

- `user`: The user who performed the action; this is a `ForeignKey` to the Django `User` model.
- `verb`: The verb describing the action that the user has performed.
- `created`: The date and time when this action was created. You use `auto_now_add=True` to automatically set this to the current datetime when the object is saved for the first time in the database.

With this basic model, you can only store actions, such as *user X did something*. You need an extra `ForeignKey` field in order to save actions that involve a `target` object, such as *user X bookmarked image Y* or *user X is now following user Y*. As you already know, a normal `ForeignKey` can point to only one model. Instead, you will need a way for the action's `target` object to be an instance of an existing model. This is what the Django `contenttypes` framework will help you to do.

Using the contenttypes framework

Django includes a `contenttypes` framework located at `django.contrib.contenttypes`. This application can track all models installed in your project and provides a generic interface to interact with your models.

The `django.contrib.contenttypes` application is included in the `INSTALLED_APPS` setting by default when you create a new project using the `startproject` command. It is used by other `contrib` packages, such as the authentication framework and the administration application.

The `contenttypes` application contains a `ContentType` model. Instances of this model represent the actual models of your application, and new instances of `ContentType` are automatically created when new models are installed in your project. The `ContentType` model has the following fields:

- `app_label`: This indicates the name of the application that the model belongs to. This is automatically taken from the `app_label` attribute of the model `Meta` options. For example, your `Image` model belongs to the `images` application.
- `model`: The name of the model class.
- `name`: This indicates the human-readable name of the model. This is automatically taken from the `verbose_name` attribute of the model `Meta` options.

Let's take a look at how you can interact with `ContentType` objects. Open the shell using the `python manage.py shell` command. You can obtain the `ContentType` object corresponding to a specific model by performing a query with the `app_label` and `model` attributes, as follows:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> image_type = ContentType.objects.get(app_label='images',
model='image')
>>> image_type
<ContentType: images | image>
```

You can also retrieve the model class from a `ContentType` object by calling its `model_class()` method:

```
>>> image_type.model_class()
<class 'images.models.Image'>
```

It's also common to get the ContentType object for a particular model class, as follows:

```
>>> from images.models import Image
>>> ContentType.objects.get_for_model(Image)
<ContentType: images | image>
```

These are just some examples of using contenttypes. Django offers more ways to work with them. You can find the official documentation about the contenttypes framework at https://docs.djangoproject.com/en/3.0/ref/contrib/contenttypes/.

Adding generic relations to your models

In generic relations, ContentType objects play the role of pointing to the model used for the relationship. You will need three fields to set up a generic relation in a model:

- A ForeignKey field to ContentType: This will tell you the model for the relationship
- A field to store the primary key of the related object: This will usually be a PositiveIntegerField to match Django's automatic primary key fields
- A field to define and manage the generic relation using the two previous fields: The contenttypes framework offers a GenericForeignKey field for this purpose

Edit the models.py file of the actions application and make it look like this:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                             related_name='actions',
                             db_index=True,
                             on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    target_ct = models.ForeignKey(ContentType,
                                  blank=True,
                                  null=True,
                                  related_name='target_obj',
                                  on_delete=models.CASCADE)
    target_id = models.PositiveIntegerField(null=True,
                                            blank=True,
                                                db_index=True)
    target = GenericForeignKey('target_ct', 'target_id')
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)

    class Meta:
        ordering = ('-created',)
```

You have added the following fields to the `Action` model:

- `target_ct`: A `ForeignKey` field that points to the `ContentType` model
- `target_id`: A `PositiveIntegerField` for storing the primary key of the related object
- `target`: A `GenericForeignKey` field to the related object based on the combination of the two previous fields

Django does not create any field in the database for `GenericForeignKey` fields. The only fields that are mapped to database fields are `target_ct` and `target_id`. Both fields have `blank=True` and `null=True` attributes, so that a `target` object is not required when saving `Action` objects.

You can make your applications more flexible by using generic relations instead of foreign keys.

Run the following command to create initial migrations for this application:

```
python manage.py makemigrations actions
```

You should see the following output:

```
Migrations for 'actions':
  actions/migrations/0001_initial.py
    - Create model Action
```

Then, run the next command to sync the application with the database:

```
python manage.py migrate
```

The output of the command should indicate that the new migrations have been applied, as follows:

```
Applying actions.0001_initial... OK
```

Let's add the `Action` model to the administration site. Edit the `admin.py` file of the `actions` application and add the following code to it:

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

@admin.register(Action)
class ActionAdmin(admin.ModelAdmin):
    list_display = ('user', 'verb', 'target', 'created')
    list_filter = ('created',)
    search_fields = ('verb',)
```

You just registered the `Action` model in the administration site. Run the `python manage.py runserver` command to start the development server and open `http://127.0.0.1:8000/admin/actions/action/add/` in your browser. You should see the page for creating a new `Action` object, as follows:

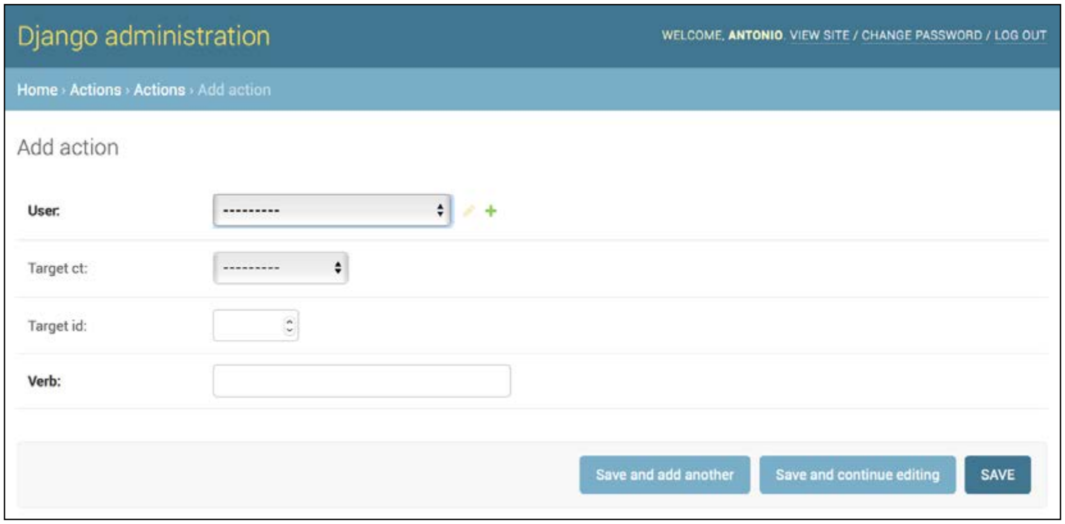

Django administration
WELCOME, ANTONIO. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Actions › Actions › Add action
Add action
User:
Target ct:
Target id:
Verb:
Save and add another
Save and continue editing
SAVE

Figure 6.4: The add action page on the Django administration site

As you will notice in the preceding screenshot, only the `target_ct` and `target_id` fields that are mapped to actual database fields are shown. The `GenericForeignKey` field does not appear in the form. The `target_ct` field allows you to select any of the registered models of your Django project. You can restrict the content types to choose from a limited set of models using the `limit_choices_to` attribute in the `target_ct` field; the `limit_choices_to` attribute allows you to restrict the content of `ForeignKey` fields to a specific set of values.

Create a new file inside the actions application directory and name it utils.py. You need to define a shortcut function that will allow you to create new Action objects in a simple way. Edit the new utils.py file and add the following code to it:

```
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    action = Action(user=user, verb=verb, target=target)
    action.save()
```

The create_action() function allows you to create actions that optionally include a target object. You can use this function anywhere in your code as a shortcut to add new actions to the activity stream.

Avoiding duplicate actions in the activity stream

Sometimes, your users might click several times on the **LIKE** or **UNLIKE** button or perform the same action multiple times in a short period of time. This will easily lead to storing and displaying duplicate actions. To avoid this, let's improve the create_action() function to skip obvious duplicated actions.

Edit the utils.py file of the actions application, as follows:

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    # check for any similar action made in the last minute
    now = timezone.now()
    last_minute = now - datetime.timedelta(seconds=60)
    similar_actions = Action.objects.filter(user_id=user.id,
                                            verb= verb,
                                            created__gte=last_minute)
    if target:
        target_ct = ContentType.objects.get_for_model(target)
        similar_actions = similar_actions.filter(
                                                  target_ct=target_ct,
                                                  target_id=target.id)
    if not similar_actions:
        # no existing actions found
```

```
            action = Action(user=user, verb=verb, target=target)
            action.save()
            return True
        return False
```

You have changed the `create_action()` function to avoid saving duplicate actions and return Boolean to tell you whether the action was saved. This is how you avoid duplicates:

- First, you get the current time using the `timezone.now()` method provided by Django. This method does the same as `datetime.datetime.now()` but returns a timezone-aware object. Django provides a setting called `USE_TZ` to enable or disable timezone support. The default `settings.py` file created using the `startproject` command includes `USE_TZ=True`.
- You use the `last_minute` variable to store the datetime from one minute ago and retrieve any identical actions performed by the user since then.
- You create an `Action` object if no identical action already exists in the last minute. You return `True` if an `Action` object was created, or `False` otherwise.

Adding user actions to the activity stream

It's time to add some actions to your views to build the activity stream for your users. You will store an action for each of the following interactions:

- A user bookmarks an image
- A user likes an image
- A user creates an account
- A user starts following another user

Edit the `views.py` file of the `images` application and add the following import:

```
from actions.utils import create_action
```

In the `image_create` view, add `create_action()` after saving the image, like this:

```
new_item.save()
create_action(request.user, 'bookmarked image', new_item)
```

In the `image_like` view, add `create_action()` after adding the user to the `users_like` relationship, as follows:

```
image.users_like.add(request.user)
create_action(request.user, 'likes', image)
```

Now, edit the `views.py` file of the `account` application and add the following import:

```
from actions.utils import create_action
```

In the `register` view, add `create_action()` after creating the `Profile` object, as follows:

```
Profile.objects.create(user=new_user)
create_action(new_user, 'has created an account')
```

In the `user_follow` view, add `create_action()`:

```
Contact.objects.get_or_create(user_from=request.user,
                              user_to=user)
create_action(request.user, 'is following', user)
```

As you can see in the preceding code, thanks to your `Action` model and your helper function, it's very easy to save new actions to the activity stream.

Displaying the activity stream

Finally, you need a way to display the activity stream for each user. You will include the activity stream in the user's dashboard. Edit the `views.py` file of the `account` application. Import the `Action` model and modify the `dashboard` view, as follows:

```
from actions.models import Action

@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # If user is following others, retrieve only their actions
        actions = actions.filter(user_id__in=following_ids)
    actions = actions[:10]

    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

In the preceding view, you retrieve all actions from the database, excluding the ones performed by the current user. By default, you retrieve the latest actions performed by all users on the platform. If the user is following other users, you restrict the query to retrieve only the actions performed by the users they follow. Finally, you limit the result to the first 10 actions returned. You don't use `order_by()` in the QuerySet because you rely on the default ordering that you provided in the `Meta` options of the `Action` model. Recent actions will come first since you set `ordering = ('-created',)` in the `Action` model.

Optimizing QuerySets that involve related objects

Every time you retrieve an `Action` object, you will usually access its related `User` object and the user's related `Profile` object. The Django ORM offers a simple way to retrieve related objects at the same time, thereby avoiding additional queries to the database.

Using select_related()

Django offers a QuerySet method called `select_related()` that allows you to retrieve related objects for one-to-many relationships. This translates to a single, more complex QuerySet, but you avoid additional queries when accessing the related objects. The `select_related` method is for `ForeignKey` and `OneToOne` fields. It works by performing a SQL `JOIN` and including the fields of the related object in the `SELECT` statement.

To take advantage of `select_related()`, edit the following line of the preceding code:

```
actions = actions[:10]
```

Also, add `select_related` to the fields that you will use, like this:

```
actions = actions.select_related('user', 'user__profile')[:10]
```

You use `user__profile` to join the `Profile` table in a single SQL query. If you call `select_related()` without passing any arguments to it, it will retrieve objects from all `ForeignKey` relationships. Always limit `select_related()` to the relationships that will be accessed afterward.

Using `select_related()` carefully can vastly improve execution time.

Using prefetch_related()

`select_related()` will help you to boost performance for retrieving related objects in one-to-many relationships. However, `select_related()` doesn't work for many-to-many or many-to-one relationships (`ManyToMany` or reverse `ForeignKey` fields). Django offers a different QuerySet method called `prefetch_related` that works for many-to-many and many-to-one relationships in addition to the relationships supported by `select_related()`. The `prefetch_related()` method performs a separate lookup for each relationship and joins the results using Python. This method also supports the prefetching of `GenericRelation` and `GenericForeignKey`.

Edit the `views.py` file of the `account` application and complete your query by adding `prefetch_related()` to it for the target `GenericForeignKey` field, as follows:

```
actions = actions.select_related('user', 'user__profile')\
                 .prefetch_related('target')[:10]
```

This query is now optimized for retrieving the user actions, including related objects.

Creating templates for actions

Let's now create the template to display a particular `Action` object. Create a new directory inside the `actions` application directory and name it `templates`. Add the following file structure to it:

```
actions/
    action/
        detail.html
```

Edit the `actions/action/detail.html` template file and add the following lines to it:

```
{% load thumbnail %}

{% with user=action.user profile=action.user.profile %}
<div class="action">
  <div class="images">
    {% if profile.photo %}
      {% thumbnail user.profile.photo "80x80" crop="100%" as im %}
      <a href="{{ user.get_absolute_url }}">
        <img src="{{ im.url }}" alt="{{ user.get_full_name }}"
         class="item-img">
      </a>
    {% endif %}
```

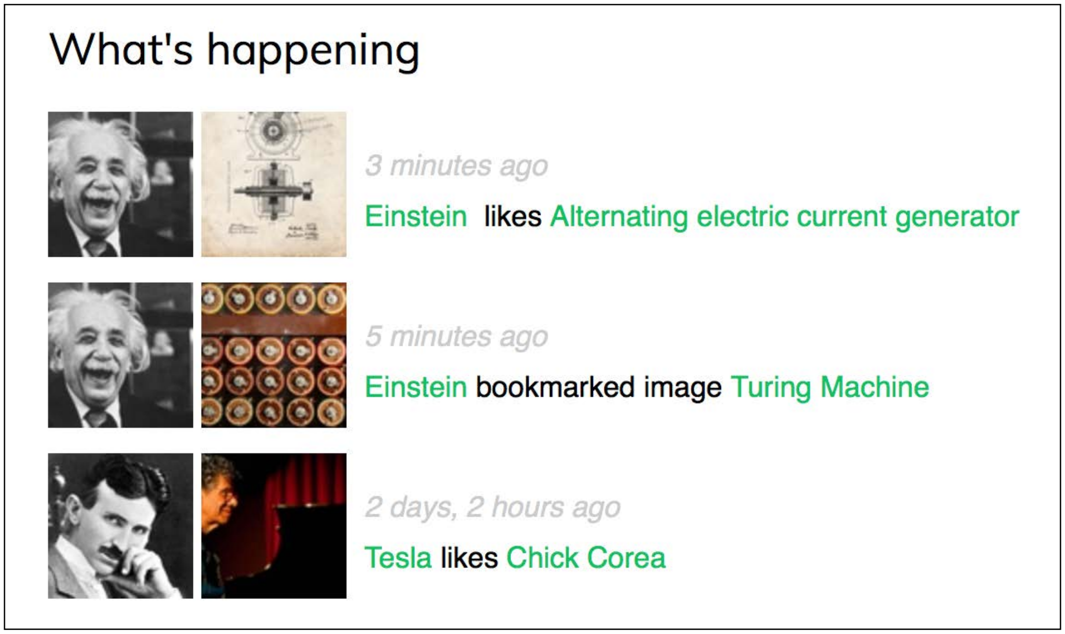
```
      {% if action.target %}
        {% with target=action.target %}
          {% if target.image %}
            {% thumbnail target.image "80x80" crop="100%" as im %}
            <a href="{{ target.get_absolute_url }}">
              <img src="{{ im.url }}" class="item-img">
            </a>
          {% endif %}
        {% endwith %}
      {% endif %}
    </div>
    <div class="info">
      <p>
        <span class="date">{{ action.created|timesince }} ago</span>
        <br />
        <a href="{{ user.get_absolute_url }}">
          {{ user.first_name }}
        </a>
        {{ action.verb }}
        {% if action.target %}
          {% with target=action.target %}
            <a href="{{ target.get_absolute_url }}">{{ target }}</a>
          {% endwith %}
        {% endif %}
      </p>
    </div>
  </div>
  {% endwith %}
```

This is the template used to display an `Action` object. First, you use the `{% with %}` template tag to retrieve the user performing the action and the related `Profile` object. Then, you display the image of the `target` object if the `Action` object has a related `target` object. Finally, you display the link to the user who performed the action, the verb, and the `target` object, if any.

Edit the `account/dashboard.html` template of the `account` application and append the following code to the bottom of the `content` block:

```
<h2>What's happening</h2>
<div id="action-list">
  {% for action in actions %}
    {% include "actions/action/detail.html" %}
  {% endfor %}
</div>
```

Open `http://127.0.0.1:8000/account/` in your browser. Log in as an existing user and perform several actions so that they get stored in the database. Then, log in using another user, follow the previous user, and take a look at the generated action stream on the dashboard page. It should look like the following:



What's happening

3 minutes ago
Einstein likes Alternating electric current generator

5 minutes ago
Einstein bookmarked image Turing Machine

2 days, 2 hours ago
Tesla likes Chick Corea

Figure 6.5: The activity stream for the current user

You just created a complete activity stream for your users, and you can easily add new user actions to it. You can also add infinite scroll functionality to the activity stream by implementing the same AJAX paginator that you used for the `image_list` view.

Using signals for denormalizing counts

There are some cases when you may want to denormalize your data. Denormalization is making data redundant in such a way that it optimizes read performance. For example, you might be copying related data to an object to avoid expensive read queries to the database when retrieving the related data. You have to be careful about denormalization and only start using it when you really need it. The biggest issue you will find with denormalization is that it's difficult to keep your denormalized data updated.

Let's take a look at an example of how to improve your queries by denormalizing counts. You will denormalize data from your `Image` model and use Django signals to keep the data updated.

Working with signals

Django comes with a signal dispatcher that allows receiver functions to get notified when certain actions occur. Signals are very useful when you need your code to do something every time something else happens. Signals allow you to decouple logic: you can capture a certain action, regardless of the application or code that triggered that action, and implement logic that gets executed whenever that action occurs. For example, you can build a signal receiver function that gets executed every time a `User` object is saved. You can also create your own signals so that others can get notified when an event happens.

Django provides several signals for models located at `django.db.models.signals`. Some of these signals are as follows:

- `pre_save` and `post_save` are sent before or after calling the `save()` method of a model
- `pre_delete` and `post_delete` are sent before or after calling the `delete()` method of a model or QuerySet
- `m2m_changed` is sent when a `ManyToManyField` on a model is changed

These are just a subset of the signals provided by Django. You can find a list of all built-in signals at `https://docs.djangoproject.com/en/3.0/ref/signals/`.

Let's say you want to retrieve images by popularity. You can use the Django aggregation functions to retrieve images ordered by the number of users who like them. Remember that you used Django aggregation functions in *Chapter 3*, *Extending Your Blog Application*. The following code will retrieve images according to their number of likes:

```
from django.db.models import Count
from images.models import Image

images_by_popularity = Image.objects.annotate(
    total_likes=Count('users_like')).order_by('-total_likes')
```

However, ordering images by counting their total `likes` is more expensive in terms of performance than ordering them by a field that stores total counts. You can add a field to the `Image` model to denormalize the total number of likes to boost performance in queries that involve this field. The issue is how to keep this field updated.

Edit the `models.py` file of the `images` application and add the following `total_likes` field to the `Image` model:

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(db_index=True,
                                              default=0)
```

The `total_likes` field will allow you to store the total count of users who like each image. Denormalizing counts is useful when you want to filter or order QuerySets by them.

> There are several ways to improve performance that you have to take into account before denormalizing fields. Consider database indexes, query optimization, and caching before starting to denormalize your data.

Run the following command to create the migrations for adding the new field to the database table:

```
python manage.py makemigrations images
```

You should see the following output:

```
Migrations for 'images':
  images/migrations/0002_image_total_likes.py
    - Add field total_likes to image
```

Then, run the following command to apply the migration:

```
python manage.py migrate images
```

The output should include the following line:

```
Applying images.0002_image_total_likes... OK
```

You need to attach a `receiver` function to the `m2m_changed` signal. Create a new file inside the `images` application directory and name it `signals.py`. Add the following code to it:

```
from django.db.models.signals import m2m_changed
from django.dispatch import receiver
from .models import Image

@receiver(m2m_changed, sender=Image.users_like.through)
def users_like_changed(sender, instance, **kwargs):
    instance.total_likes = instance.users_like.count()
    instance.save()
```

First, you register the `users_like_changed` function as a receiver function using the `receiver()` decorator. You attach it to the `m2m_changed` signal. Then, you connect the function to `Image.users_like.through` so that the function is only called if the `m2m_changed` signal has been launched by this sender. There is an alternate method for registering a receiver function; it consists of using the `connect()` method of the `Signal` object.

Django signals are synchronous and blocking. Don't confuse signals with asynchronous tasks. However, you can combine both to launch asynchronous tasks when your code gets notified by a signal. You will learn to create asynchronous tasks with Celery in *Chapter 7, Building an Online Shop*.

You have to connect your receiver function to a signal so that it gets called every time the signal is sent. The recommended method for registering your signals is by importing them in the `ready()` method of your application configuration class. Django provides an application registry that allows you to configure and introspect your applications.

Application configuration classes

Django allows you to specify configuration classes for your applications. When you create an application using the `startapp` command, Django adds an `apps.py` file to the application directory, including a basic application configuration that inherits from the `AppConfig` class.

The application configuration class allows you to store metadata and the configuration for the application, and it provides introspection for the application. You can find more information about application configurations at `https://docs.djangoproject.com/en/3.0/ref/applications/`.

In order to register your signal `receiver` functions, when you use the `receiver()` decorator, you just need to import the `signals` module of your application inside the `ready()` method of the application configuration class. This method is called as soon as the application registry is fully populated. Any other initializations for your application should also be included in this method.

Edit the `apps.py` file of the `images` application and make it look like this:

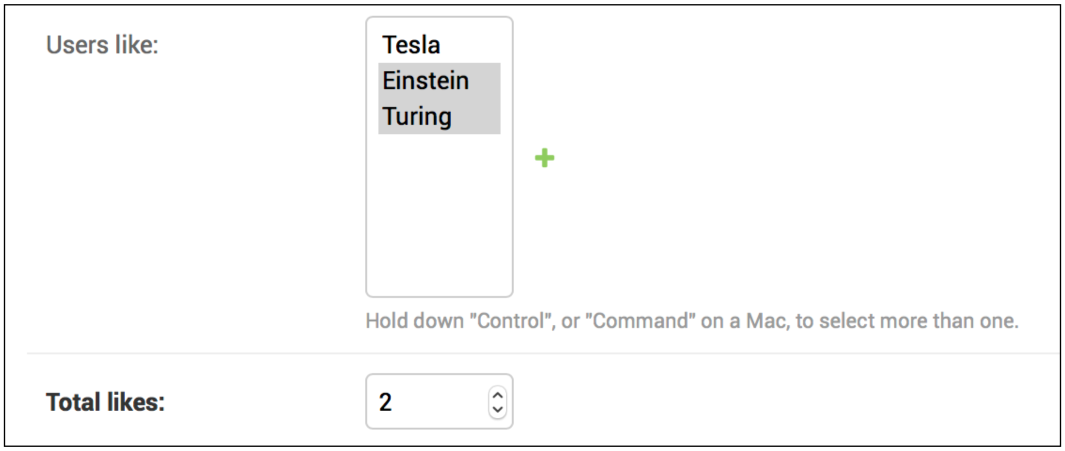
```
from django.apps import AppConfig

class ImagesConfig(AppConfig):
    name = 'images'

    def ready(self):
        # import signal handlers
        import images.signals
```

You import the signals for this application in the `ready()` method so that they are imported when the `images` application is loaded.

Run the development server with the following command:

```
python manage.py runserver
```

Open your browser to view an image detail page and click on the **LIKE** button. Go back to the administration site, navigate to the edit image URL, such as `http://127.0.0.1:8000/admin/images/image/1/change/`, and take a look at the `total_likes` attribute. You should see that the `total_likes` attribute is updated with the total number of users who like the image, as follows:



Users like: Tesla, Einstein, Turing (Einstein and Turing selected)

Hold down "Control", or "Command" on a Mac, to select more than one.

Total likes: 2

Figure 6.6: The image edit page on the administration site, including denormalization for total likes

Now, you can use the `total_likes` attribute to order images by popularity or display the value anywhere, avoiding using complex queries to calculate it. Consider the following query to get images ordered according to their likes count:

```
from django.db.models import Count

images_by_popularity = Image.objects.annotate(
    likes=Count('users_like')).order_by('-likes')
```

The preceding query can now be written as follows:

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

This results in a less expensive SQL query. This is just an example of how to use Django signals.

Use signals with caution since they make it difficult to know the control flow. In many cases, you can avoid using signals if you know which receivers need to be notified.

You will need to set initial counts for the rest of the `Image` objects to match the current status of the database. Open the shell with the `python manage.py shell` command and run the following code:

```
from images.models import Image
for image in Image.objects.all():
    image.total_likes = image.users_like.count()
    image.save()
```

The likes count for each image is now up to date.

Using Redis for storing item views

Redis is an advanced key/value database that allows you to save different types of data. It also has extremely fast I/O operations. Redis stores everything in memory, but the data can be persisted by dumping the dataset to disk every once in a while, or by adding each command to a log. Redis is very versatile compared to other key/value stores: it provides a set of powerful commands and supports diverse data structures, such as strings, hashes, lists, sets, ordered sets, and even bitmaps or HyperLogLogs.

Although SQL is best suited to schema-defined persistent data storage, Redis offers numerous advantages when dealing with rapidly changing data, volatile storage, or when a quick cache is needed. Let's take a look at how Redis can be used to build a new functionality into your project.

Installing Redis

If you are using Linux or macOS, download the latest Redis version from `https://redis.io/download`. Unzip the `tar.gz` file, enter the `redis` directory, and compile Redis using the `make` command, as follows:

```
cd redis-5.0.8

make
```

Redis is now installed on your machine. If you are using Windows, the preferred method to install Redis is to enable the **Windows Subsystem for Linux** (**WSL**) and install it in the Linux system. You can read instructions on enabling WSL and installing Redis at `https://redislabs.com/blog/redis-on-windows-10/`.

After installing Redis, use the following shell command to start the Redis server:

```
src/redis-server
```

You should see an output that ends with the following lines:

```
# Server initialized
* Ready to accept connections
```

By default, Redis runs on port `6379`. You can specify a custom port using the `--port` flag, for example, `redis-server --port 6655`.

Keep the Redis server running and open another shell. Start the Redis client with the following command:

```
src/redis-cli
```

You will see the Redis client shell prompt, like this:

```
127.0.0.1:6379>
```

The Redis client allows you to execute Redis commands directly from the shell. Let's try some commands. Enter the `SET` command in the Redis shell to store a value in a key:

```
127.0.0.1:6379> SET name "Peter"
OK
```

The preceding command creates a `name` key with the string value `"Peter"` in the Redis database. The `OK` output indicates that the key has been saved successfully.

Next, retrieve the value using the `GET` command, as follows:

```
127.0.0.1:6379> GET name
"Peter"
```

You can also check whether a key exists using the `EXISTS` command. This command returns `1` if the given key exists, and `0` otherwise:

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

You can set the time for a key to expire using the EXPIRE command, which allows you to set time-to-live in seconds. Another option is using the EXPIREAT command, which expects a Unix timestamp. Key expiration is useful for using Redis as a cache or to store volatile data:

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379> EXPIRE name 2
(integer) 1
```

Wait for two seconds and try to get the same key again:

```
127.0.0.1:6379> GET name
(nil)
```

The (nil) response is a null response and means that no key has been found. You can also delete any key using the DEL command, as follows:

```
127.0.0.1:6379> SET total 1
OK
127.0.0.1:6379> DEL total
(integer) 1
127.0.0.1:6379> GET total
(nil)
```

These are just basic commands for key operations. You can take a look at all Redis commands at https://redis.io/commands and all Redis data types at https://redis.io/topics/data-types.

Using Redis with Python

You will need Python bindings for Redis. Install redis-py via pip using the following command:

```
pip install redis==3.4.1
```

You can find the redis-py documentation at https://redis-py.readthedocs.io/.

The redis-py package interacts with Redis, providing a Python interface that follows the Redis command syntax. Open the Python shell and execute the following code:

```
>>> import redis
>>> r = redis.Redis(host='localhost', port=6379, db=0)
```

The preceding code creates a connection with the Redis database. In Redis, databases are identified by an integer index instead of a database name. By default, a client is connected to the database `0`. The number of available Redis databases is set to `16`, but you can change this in the `redis.conf` configuration file.

Next, set a key using the Python shell:

```
>>> r.set('foo', 'bar')
True
```

The command returns `True`, indicating that the key has been successfully created. Now you can retrieve the key using the `get()` command:

```
>>> r.get('foo')
b'bar'
```

As you will note from the preceding code, the methods of `Redis` follow the Redis command syntax.

Let's integrate Redis into your project. Edit the `settings.py` file of the `bookmarks` project and add the following settings to it:

```
    REDIS_HOST = 'localhost'
    REDIS_PORT = 6379
    REDIS_DB = 0
```

These are the settings for the Redis server and the database that you will use for your project.

Storing item views in Redis

Let's find a way to store the total number of times an image has been viewed. If you implement this using the Django ORM, it will involve a SQL `UPDATE` query every time an image is displayed. If you use Redis instead, you just need to increment a counter stored in memory, resulting in a much better performance and less overhead.

Edit the `views.py` file of the `images` application and add the following code to it after the existing `import` statements:

```
import redis
from django.conf import settings

# connect to redis
r = redis.Redis(host=settings.REDIS_HOST,
                port=settings.REDIS_PORT,
                db=settings.REDIS_DB)
```

With the preceding code, you establish the Redis connection in order to use it in your views. Edit the `views.py` file of the `images` application and modify the `image_detail` view, like this:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # increment total image views by 1
    total_views = r.incr(f'image:{image.id}:views')
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

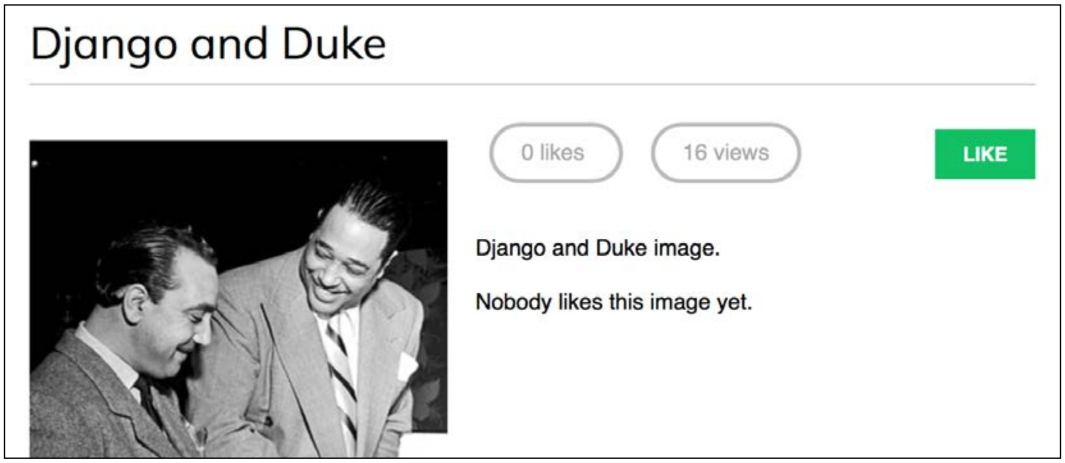
In this view, you use the `incr` command that increments the value of a given key by `1`. If the key doesn't exist, the `incr` command creates it. The `incr()` method returns the final value of the key after performing the operation. You store the value in the `total_views` variable and pass it in the template context. You build the Redis key using a notation, such as `object-type:id:field` (for example, `image:33:id`).

The convention for naming Redis keys is to use a colon sign as a separator for creating namespaced keys. By doing so, the key names are especially verbose and related keys share part of the same schema in their names.

Edit the `images/image/detail.html` template of the `images` application and add the following code to it after the existing `<span class="count">` element:

```
<span class="count">
  {{ total_views }} view{{ total_views|pluralize }}
</span>
```

Now, open an image detail page in your browser and reload it several times. You will see that each time the view is processed, the total views displayed is incremented by 1. Take a look at the following example:



Django and Duke image.

Nobody likes this image yet.

Figure 6.7: The image detail page, including the count of likes and views

Great! You have successfully integrated Redis into your project to store item counts.

Storing a ranking in Redis

Let's build something more complex with Redis. You will create a ranking of the most viewed images in your platform. For building this ranking, you will use Redis sorted sets. A sorted set is a non-repeating collection of strings in which every member is associated with a score. Items are sorted by their score.

Edit the `views.py` file of the `images` application and make the `image_detail` view look as follows:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # increment total image views by 1
    total_views = r.incr(f'image:{image.id}:views')
    # increment image ranking by 1
    r.zincrby('image_ranking', 1, image.id)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

You use the zincrby() command to store image views in a sorted set with the image:ranking key. You will store the image id and a related score of 1, which will be added to the total score of this element in the sorted set. This will allow you to keep track of all image views globally and have a sorted set ordered by the total number of views.

Now, create a new view to display the ranking of the most viewed images. Add the following code to the views.py file of the images application:

```
@login_required
def image_ranking(request):
    # get image ranking dictionary
    image_ranking = r.zrange('image_ranking', 0, -1,
                             desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # get most viewed images
    most_viewed = list(Image.objects.filter(
                           id__in=image_ranking_ids))
    most_viewed.sort(key=lambda x: image_ranking_ids.index(x.id))
    return render(request,
                  'images/image/ranking.html',
                  {'section': 'images',
                   'most_viewed': most_viewed})
```

The image_ranking view works like this:

1. You use the zrange() command to obtain the elements in the sorted set. This command expects a custom range according to the lowest and highest score. Using 0 as the lowest and -1 as the highest score, you are telling Redis to return all elements in the sorted set. You also specify desc=True to retrieve the elements ordered by descending score. Finally, you slice the results using [:10] to get the first 10 elements with the highest score.
2. You build a list of returned image IDs and store it in the image_ranking_ids variable as a list of integers. You retrieve the Image objects for those IDs and force the query to be executed using the list() function. It is important to force the QuerySet execution because you will use the sort() list method on it (at this point, you need a list of objects instead of a QuerySet).
3. You sort the Image objects by their index of appearance in the image ranking. Now you can use the most_viewed list in your template to display the 10 most viewed images.

Create a new `ranking.html` template inside the `images/image/` template directory of the `images` application and add the following code to it:

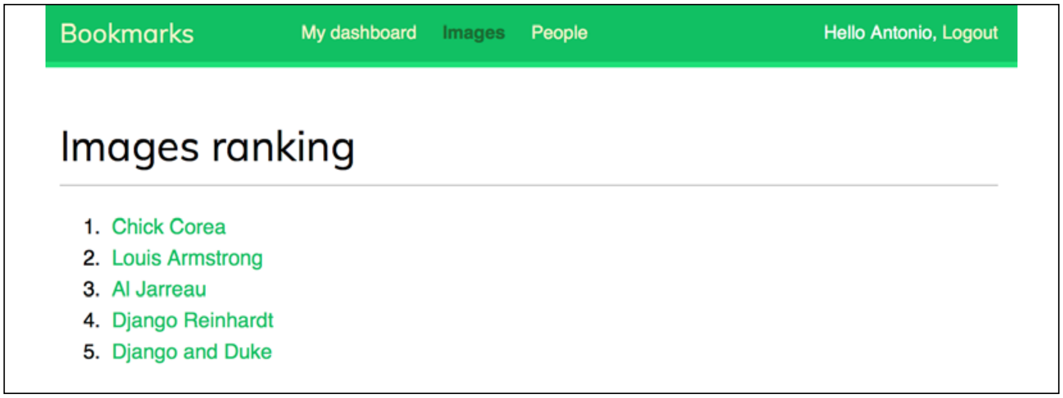
```
{% extends "base.html" %}

{% block title %}Images ranking{% endblock %}

{% block content %}
  <h1>Images ranking</h1>
  <ol>
    {% for image in most_viewed %}
      <li>
        <a href="{{ image.get_absolute_url }}">
          {{ image.title }}
        </a>
      </li>
    {% endfor %}
  </ol>
{% endblock %}
```

The template is pretty straightforward. You iterate over the `Image` objects contained in the `most_viewed` list and display their names, including a link to the image detail page.

Finally, you need to create a URL pattern for the new view. Edit the `urls.py` file of the `images` application and add the following pattern to it:

```
path('ranking/', views.image_ranking, name='ranking'),
```

Run the development server, access your site in your web browser, and load the image detail page multiple times for different images. Then, access `http://127.0.0.1:8000/images/ranking/` from your browser. You should be able to see an image ranking, as follows:



Images ranking

1. Chick Corea
2. Louis Armstrong
3. Al Jarreau
4. Django Reinhardt
5. Django and Duke

Figure 6.8: The ranking page built with data retrieved from Redis

Great! You just created a ranking with Redis.

Next steps with Redis

Redis is not a replacement for your SQL database, but it does offer fast in-memory storage that is more suitable for certain tasks. Add it to your stack and use it when you really feel it's needed. The following are some scenarios in which Redis could be useful:

- **Counting**: As you have seen, it is very easy to manage counters with Redis. You can use `incr()` and `incrby()` for counting stuff.
- **Storing latest items**: You can add items to the start/end of a list using `lpush()` and `rpush()`. Remove and return the first/last element using `lpop()` / `rpop()`. You can trim the list's length using `ltrim()` to maintain its length.
- **Queues**: In addition to `push` and `pop` commands, Redis offers the blocking of queue commands.
- **Caching**: Using `expire()` and `expireat()` allows you to use Redis as a cache. You can also find third-party Redis cache backends for Django.
- **Pub/sub**: Redis provides commands for subscribing/unsubscribing and sending messages to channels.
- **Rankings and leaderboards**: Redis sorted sets with scores make it very easy to create leaderboards.
- **Real-time tracking**: Redis's fast I/O makes it perfect for real-time scenarios.

Summary

In this chapter, you built a follow system using many-to-many relationships with an intermediary model. You also created an activity stream using generic relations and you optimized QuerySets to retrieve related objects. This chapter then introduced you to Django signals, and you created a signal receiver function to denormalize related object counts. We covered application configuration classes, which you used to load your signal handlers. You also learned how to install and configure Redis in your Django project. Finally, you used Redis in your project to store item views, and you built an image ranking with Redis.

In the next chapter, you will learn how to build an online shop. You will create a product catalog and build a shopping cart using sessions. You will also discover how to launch asynchronous tasks using Celery.