

MODULE-4: GENERIC VIEWS AND DJANGO STATE PERSISTENCE**4.0 USING GENERIC VIEWS**

Using generic views in Django is a powerful way to simplify the creation of common views for tasks such as displaying database objects, handling form submissions, and performing CRUD operations. Generic views provide pre-built functionality for common patterns, reducing the amount of code you need to write. Here's how to use generic views in Django:

Step 1: Import Generic Views

First, import the generic views you want to use in your views.py file. Django provides a variety of generic views, such as ListView, DetailView, CreateView, UpdateView, and DeleteView.

```
from django.views.generic import ListView, DetailView, CreateView, UpdateView, DeleteView
```

Step 2: Define URL Patterns

Define URL patterns for your views in your project's urls.py file. Map each view to a URL pattern using Django's path() function.

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

```
from .views import MyListView, MyDetailView, MyCreateView, MyUpdateView, MyDeleteView
```

```
urlpatterns = [  
    path('my-list/', MyListView.as_view(), name='my-list'),  
    path('my-detail/<int:pk>/', MyDetailView.as_view(), name='my-detail'),  
    path('my-create/', MyCreateView.as_view(), name='my-create'),  
    path('my-update/<int:pk>/', MyUpdateView.as_view(), name='my-update'),  
    path('my-delete/<int:pk>/', MyDeleteView.as_view(), name='my-delete'),  
]
```

Step 3: Define Views

Define your views by subclassing the appropriate generic views and specifying the model and template_name attributes. Customize the views by overriding methods such as get_queryset() or form_valid().

```
from django.views.generic import ListView, DetailView, CreateView, UpdateView, DeleteView
```

```
from .models import MyModel
```

```
class MyListView(ListView):
```

```
    model = MyModel

    template_name = 'myapp/my_model_list.html'

class MyDetailView(DetailView):
    model = MyModel

    template_name = 'myapp/my_model_detail.html'

class MyCreateView(CreateView):
    model = MyModel

    template_name = 'myapp/my_model_form.html'

    fields = ['field1', 'field2']

class MyUpdateView(UpdateView):
    model = MyModel

    template_name = 'myapp/my_model_form.html'

    fields = ['field1', 'field2']

class MyDeleteView(DeleteView):
    model = MyModel

    template_name = 'myapp/my_model_confirm_delete.html'

    success_url = '/my-list/'
```

Step 4: Create Templates

Create templates for your views if they don't already exist. These templates should match the `template_name` attributes specified in your views.

Step 5: Access URLs

Access the URLs associated with your generic views in a web browser or link to them in your templates using Django's `{% url %}` template tag.

By using generic views, you can quickly create views for common tasks without having to write a lot of boilerplate code. This can lead to cleaner, more maintainable code and faster development times.

Using Generic Views

Basic Concept

- Generic Views: Django provides built-in views that can handle common patterns, reducing the need to write repetitive view code.
- Configuration: Use configuration dictionaries in URLconf files, passing them as the third member of the URLconf tuple.

Example: Static "About" Page

1. Simple URLconf

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns("",
    (r'^about/$', direct_to_template, {'template': 'about.html'})
)
```

- **Explanation:** `direct_to_template` is used to render `about.html` without additional view code.

Advanced Example: Dynamic Static Pages

- **Goal:** Render `about/<whatever>.html` for URLs of the form `/about/<whatever>/`.

1. Modify URLconf to Point to a View Function:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
from mysite.books.views import about_pages
```

```
urlpatterns = patterns("",
    (r'^about/$', direct_to_template, {'template': 'about.html'}),
    (r'^about/(\w+)/$', about_pages),
)
```

2. Write the View Function:

```
from django.http import Http404
```

```
from django.template import TemplateDoesNotExist

from django.views.generic.simple import direct_to_template

def about_pages(request,
page): try:
    return direct_to_template(request, template="about/%s.html" % page)
except TemplateDoesNotExist:
    raise Http404()
```

Explanation:

- Function: about_pages dynamically constructs the template path.
- Error Handling: Catches TemplateDoesNotExist exceptions and raises Http404 to prevent server errors for missing templates.

1. Generic Views:

Used by creating configuration dictionaries in URLconf.

Examples include direct_to_template for rendering static templates.

Other generic views include list views, detail views, and more.

2. Advantages:

Reduces boilerplate code.

Increases readability and maintainability.

Can be reused within custom view functions.

3. Custom View Integration:

Generic views can be called within custom views, returning HttpResponse objects directly.

Custom error handling, like catching TemplateDoesNotExist, can be implemented for more robust applications.

Generic Views of Objects

Purpose: Simplifies creating views that display lists and details of database objects.

Benefits: Reduces repetitive code, leverages built-in Django functionality for common tasks.

Example: Object List View

1. Model Definition:

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province =
models.CharField(max_length=30) country =
models.CharField(max_length=50) website =
models.URLField()

    def __unicode__(self): return
        self.name

    class Meta:
        ordering = ['name']
```

2. Basic URLconf for Object List View:

```
from django.conf.urls.defaults import *
from django.views.generic import
list_detail from mysite.books.models import
Publisher publisher_info = {
    'queryset': Publisher.objects.all(),
}
urlpatterns = patterns("",
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

3. Specifying a Template:

- You can explicitly specify the template to be used.

```
from django.conf.urls.defaults import *
from django.views.generic import
list_detail from mysite.books.models import
Publisher

publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_name': 'publisher_list_page.html',
}

urlpatterns = patterns(
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

- **Default Template:** If `template_name` is not specified, Django infers the template name based on the model and app name, e.g., `books/publisher_list.html`.

4. Template Example:

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

{% block content %}
<h2>Publishers</h2>
<ul>
{% for publisher in object_list %}
    <li>{{ publisher.name }}</li>
{% endfor %}
</ul>
{% endblock %}
```

- Context Variable: `object_list` contains all publisher objects.

Customizing Generic Views

- **Info Dictionary:** The dictionary passed to the generic view can be customized to include additional options.

- **Template Context:** Additional context variables can be passed to the template by modifying the dictionary.
 - **Generic View Options:** Appendix C of the Django documentation provides detailed information on all available options for generic views.
1. **Ease of Use:** Generic views simplify the creation of common views for database objects.
 2. **Flexibility:** Options in the info dictionary allow for extensive customization without writing additional view code.
 3. **Template Inference:** Django can infer template names, but explicit specification is possible for better control.
 4. **Reusability:** Generic views promote code reusability and maintainability across projects.

Extending Generic Views of objects

Using generic views can significantly speed up development in Django, but there are times when they need to be extended to handle more complex use cases.

Here are some common patterns for extending generic views:

Making "Friendly" Template Contexts

Instead of using the default variable name `object_list`, use a more descriptive name like `publisher_list`. This can be achieved with the `template_object_name` argument.

Example:

```
from django.conf.urls.defaults import *
from django.views.generic import
list_detail from mysite.books.models import
Publisher

publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_name': 'publisher_list_page.html',
    'template_object_name': 'publisher',
}
```

```
urlpatterns = patterns(",  
    (r'^publishers/$', list_detail.object_list, publisher_info)  
)
```

Template:

```
{% extends "base.html" %}  
  
{% block content %}  
<h2>Publishers</h2>  
<ul>  
    {% for publisher in publisher_list %}  
        <li>{{ publisher.name }}</li>  
    {% endfor %}  
</ul>  
{% endblock %}.
```

Adding Extra Context

You can add extra context to the template by using the `extra_context` argument. Use a callable to ensure the context is evaluated each time the view is called.

```
publisher_info = {  
    'queryset': Publisher.objects.all(),  
    'template_object_name': 'publisher',  
    'extra_context': {'book_list': Book.objects.all}  
}
```

Viewing Subsets of Objects

Customize the queryset to filter objects.

Example

```
apress_books =  
{  
    'queryset': Book.objects.filter(publisher_name='Apress Publishing'),  
    'template_name': 'books/apress_list.html'  
}
```

```
urlpatterns = patterns(",  
    (r'^publishers/$', list_detail.object_list, publisher_info),  
    (r'^books/apress/$', list_detail.object_list, apress_books),
```

)

Complex Filtering with Wrapper Functions

Use a wrapper function to filter objects based on URL parameters.

Example

```
from django.shortcuts import
get_object_or_404 from django.views.generic
import list_detail
from mysite.books.models import Book, Publisher
def books_by_publisher(request, name):
    publisher = get_object_or_404(Publisher, name__iexact=name)
    return list_detail.object_list(request,
        queryset=Book.objects.filter(publisher=publisher),
        template_name='books/books_by_publisher.html',
        template_object_name='book', extra_context={'publisher':
        publisher}
    )

urlpatterns = patterns("",
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/(\w+)/$', books_by_publisher),
)
```

Performing Extra Work

Perform additional operations before or after calling the generic view.

Example: Updating Last Accessed Field

```
import datetime
from django.shortcuts import
get_object_or_404 from django.views.generic
import list_detail from mysite.books.models
import Author
def author_detail(request, author_id):
    response =
    list_detail.object_detail(
        request, queryset=Author.objects.all(), object_id=author_id,)

    now = datetime.datetime.now()
```

```
Author.objects.filter(id=author_id).update(last_accessed=now)
return response
```

```
urlpatterns =
    patterns("", # ...
        (r'^authors/(?P<author_id>\d+)/$', author_detail),
        # ...
    )
```

Example: Downloadable Plain-Text Version.

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        request,
        queryset=Author.objects.all(),
        mimetype='text/plain',
        template_name='books/author_list.txt'
    )

    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

MIME Types

1. Structure:

- MIME types have a format: type/subtype.
- Example: image/jpeg for JPEG images.

2. Common MIME Types:

Text

Plain Text: text/plain

- Example: A .txt file containing simple text.

HTML: text/html

- Example: An .html file for web pages.

CSS: text/css

- Example: A .css file for styling web pages.

Image

- JPEG: image/jpeg
- Example: A .jpg or .jpeg file.
- PNG: image/png
- Example: A .png file.

- GIF: image/gif
- Example: A .gif file for simple animations.

Audio

- MP3: audio/mpeg
- Example: An .mp3 music file.
- WAV: audio/wav
- Example: A .wav sound file.

Video

- MP4: video/mp4
- Example: An .mp4 video file.
- WebM: video/webm
- Example: A .webm video file.

Application

- JSON: application/json
- Example: A .json file for structured data.
- PDF: application/pdf
- Example: A .pdf document.
- ZIP: application/zip
- Example: A .zip compressed file.
- Word Document: application/vnd.openxmlformats-officedocument.wordprocessingml.document
- Example: A .docx file created by Microsoft Word.

3. Usage in HTTP:

- MIME types are specified in HTTP headers to indicate the type of content.

Example

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html; charset=UTF-8
```

- This header tells the client that the content is an HTML document.
-

4. Usage in Emails:

- MIME types are used to specify the format of email content and attachments.
- Example: An email with a plain text body and an image attachment might have:
 - Content-Type: text/plain
 - Content-Type: image/jpeg for the attached image.

5. Custom MIME Types:

- Custom or non-standard types often start with x-.
- Example: application/x-custom-type

6. Registration:

- Official MIME types are registered with IANA (Internet Assigned Numbers Authority).

Generating Non-HTML contents like CSV and PDF

1. CSV Format:

- Simple data format for table rows, separated by commas.
- Example

Year, Unruly Airline Passengers

1995,146

1996,184

1997,235

1998,200

1999,226

2000,251

2001,299

2002,273

2003,281

2004,304

2005,203

2006,134

2007,147

2. Using Python's CSV Library:

- Python's csv module handles CSV file operations.
- Example of generating CSV with

Django: `import csv`

`from django.http import HttpResponse`

```
UNRULY_PASSENGERS = [146, 184, 235, 200, 226, 251, 299, 273, 281, 304,
203, 134, 147]
```

```
def unruly_passengers_csv(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="unruly.csv"'
    writer = csv.writer(response)
    writer.writerow(['Year', 'Unruly Airline Passengers'])
    for year, num in zip(range(1995, 2008), UNRULY_PASSENGERS):
        writer.writerow([year, num])
    return response
```

MIME Type: Set Content-Type to text/csv to indicate CSV format.

Content-Disposition: Include attachment; **filename="unruly.csv"** to prompt file download.

HttpResponse as File: Use HttpResponse object with csv.writer.

Writing Rows: Use writer.writerow() to write each row in the CSV file.

Generating PDFs with Django

1. PDF Format:

- PDF (Portable Document Format) is used for printable documents with precise formatting.

2. Using ReportLab Library:

- ReportLab is a library for generating PDFs in Python.
- Installation:
 - Download from ReportLab Downloads.
 - Verify installation: `import reportlab`.

3. Example of Generating PDFs:

- Use ReportLab to create customized PDF documents dynamically.
- Installation: Install ReportLab from the official website.
- Import Test: Verify installation by importing ReportLab in Python

Syndication Feed Framework

High-level framework for generating RSS and Atom feeds.

Create multiple feeds using simple Python classes.

Feeds are conventionally accessed via the `/feeds/` URL.

1. Initialization:

- Add a URLconf to activate syndication feeds.

```
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed', {'feed_dict': feeds}),
```
- This directs all URLs starting with `/feeds/` to the RSS framework.
- Customize the URL prefix (`feeds/`) as needed.

2. URL Configuration:

- Use `feed_dict` to map feed slugs to Feed classes:

```
from django.conf.urls.defaults import *
from mysite.feeds import LatestEntries, LatestEntriesByCategory
feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
```

```
}  
urlpatterns =  
    patterns("", # ...  
        (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed', {'feed_dict':  
            feeds}), # ...  
    )
```

Example:

- feeds/latest/ for the LatestEntries feed.
- feeds/categories/ for the LatestEntriesByCategory feed.

3. Feed Class:

- Define Feed classes to represent syndication feeds.
- Subclass `django.contrib.syndication.feeds.Feed`.
- Feed classes can be placed anywhere in the code tree.

4. Feed Class Example:

- Simple feed (e.g., latest blog entries):

```
from django.contrib.syndication.views import Feed
```

```
class LatestEntries(Feed):  
    title = "Latest Blog Entries"  
    link = "/latest/"  
    description = "Updates on the latest blog entries."
```

```
    def items(self):  
        return Entry.objects.order_by('-pub_date')[:5]
```

```
    def item_title(self, item):  
        return item.title
```

```
    def item_description(self, item):  
        return item.description
```

- Add URLconf for syndication.
- Map URLs to Feed classes using feed_dict.
- Define Feed classes by subclassing Feed.
- Customize and extend feeds based on application needs.

Sitemap Framework

- A sitemap is an XML file that helps search engines index your site.
- Tells search engines how frequently pages change and their importance.
- Example

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.djangoproject.com/documentation/</loc>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.djangoproject.com/documentation/0_90/</loc>
    <changefreq>never</changefreq>
    <priority>0.1</priority>
  </url>
</urlset>
```

1. Installation:

- Add 'django.contrib.sitemaps' to INSTALLED_APPS.
- Ensure 'django.template.loaders.app_directories.load_template_source' is in TEMPLATE_LOADERS.
- Install the sites framework.

2. Initialization:

- Add this line to URLconf to activate sitemap generation
`(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})`
- The dot in sitemap.xml is escaped with a backslash.

3. URL Configuration:

- Define sitemaps dictionary mapping section labels to Sitemap classes
- ```
from django.conf.urls.defaults import *
from mysite.blog.models import Entry
from django.contrib.sitemaps import Sitemap

class BlogSitemap(Sitemap):
 changefreq = "never"
 priority = 0.5

 def items(self):
 return Entry.objects.filter(is_draft=False)

 def lastmod(self, obj):
 return obj.pub_date

 sitemaps = {
 'blog': BlogSitemap,
 }

urlpatterns = patterns("",
 (r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps':
sitemaps}),
)
```

### 4. Sitemap Class:

- Subclass `django.contrib.sitemaps.Sitemap`.
- Define methods and attributes such as `items`, `lastmod`, `changefreq`, `priority`.

### 5. Example Sitemap Class:

```
from django.contrib.sitemaps import Sitemap
from mysite.blog.models import Entry
class BlogSitemap(Sitemap):
 changefreq = "never"
 priority = 0.5

 def items(self):
 return Entry.objects.filter(is_draft=False)
```

```
def lastmod(self,
 obj): return
 obj.pub_date
```

## **6. Convenience Classes:**

- FlatPageSitemap: For flatpages defined for the current site.
- GenericSitemap: Works with generic views.

## **7. Example with GenericSitemap and FlatPageSitemap:**

```
from django.conf.urls.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.blog.models import Entry
info_dict = {
 'queryset': Entry.objects.all(),
 'date_field': 'pub_date',
}
sitemaps = {
 'flatpages': FlatPageSitemap,
 'blog': GenericSitemap(info_dict, priority=0.6),
}
urlpatterns = patterns("",
 (r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps}),
)
```

## **8. Creating a Sitemap Index:**

- Use two views in URLconf  
(r'^sitemap.xml\$', 'django.contrib.sitemaps.views.index', {'sitemaps':  
sitemaps}), (r'^sitemap-(?P<section>.+).xml\$',  
'django.contrib.sitemaps.views.sitemap',  
{'sitemaps': sitemaps})),

## **9. Pinging Google:**

- Use ping\_google function to notify Google of sitemap changes.
- Example

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
```

```
...
def save(self, *args, **kwargs):
 super(Entry, self).save(*args, **kwargs)
 try:
 ping_google()
 except
 Exception:
 pass
```

- **Command-line:**

```
python manage.py ping_google /sitemap.xml
```

### Cookies, Sessions

#### **Introduction to Cookies:**

- Cookies help overcome HTTP's statelessness by storing small pieces of information in the browser.
- Browsers send cookies back to the server with each request, allowing servers to recognize returning users.

#### **How Cookies Work:**

- **Example:**

Browser requests a page from Google:

```
GET / HTTP/1.1
```

```
Host: google.com
```

Google responds with a Set-Cookie header:

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html
```

```
Set-Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
```

```
 expires=Sun, 17-Jan-2038 19:14:07 GMT;
```

```
 path=/; domain=.google.com
```

```
Server: GWS/2.1
```

Browser stores the cookie and sends it back on subsequent

requests GET / HTTP/1.1

Host: google.com

Cookie: PREF=ID=5b14f22bdafl1e81c:TM=1167000671:LM=1167000671

### Getting and Setting Cookies in Django:

- Reading Cookies:

Use the COOKIES attribute of HttpRequest to read

```
cookies. def show_color(request):
 if "favorite_color" in request.COOKIES:
 return HttpResponse("Your favorite color is %s" %
 request.COOKIES["favorite_color"])
 else:
 return HttpResponse("You don't have a favorite color.")
```

- Writing Cookies:

Use the set\_cookie() method of HttpResponse to set

```
cookies. def set_color(request):
 if "favorite_color" in request.GET:
 response = HttpResponse("Your favorite color is now %s" %
 request.GET["favorite_color"])
 response.set_cookie("favorite_color", request.GET["favorite_color"])
 return response
 else:
 return HttpResponse("You didn't give a favorite color.")
```

- **Optional Arguments for set\_cookie():**

You can pass various optional arguments to response.set\_cookie() to control aspects of the cookie, such as:

- **max\_age:** The maximum age of the cookie in seconds.

- **expires:** The expiration date of the cookie.
- **path:** The path for which the cookie is valid.
- **domain:** The domain for which the cookie is valid.
- **secure:** If True, the cookie will only be sent over HTTPS.
- **httponly:** If True, the cookie will only be accessible via HTTP(S) and not via client-side scripts.

- **Problems with Cookies:**

Voluntary Storage:

- Clients can choose not to accept or store cookies, making them unreliable.
- Developers should verify if a user accepts cookies before relying on them.

- **Security Concerns:**

Cookies sent over HTTP are vulnerable to snooping attacks.

Never store sensitive information in cookies.

Man-in-the-Middle Attacks: Attackers can intercept and use cookies to impersonate users.

- **Tampering:**

Browsers allow users to edit cookies, making it risky to store important state information in cookies.

Example of a security mistake:

# Storing something like `IsLoggedIn=1` in a cookie can be easily tampered with.

- Use secure methods (e.g., HTTPS) to transmit cookies.
- Avoid storing sensitive information directly in cookies.
- Validate and sanitize all data received from cookies.

### Django's Session Framework

- Django's session framework addresses the limitations and potential security issues of

using cookies directly by providing a way to store and retrieve arbitrary data on a per-site visitor basis.

- The session data is stored server-side, with only a hashed session ID sent to the client, mitigating many common cookie-related issues.
- **Enabling Sessions**

Middleware and Installed Apps:

- Ensure SessionMiddleware is included in your MIDDLEWARE\_CLASSES

```
MIDDLEWARE_CLASSES = [
 ...
 'django.contrib.sessions.middleware.SessionMiddleware',
 ...
]
```

- Ensure django.contrib.sessions is in your INSTALLED\_APPS.

```
INSTALLED_APPS = [
 ...
 'django.contrib.sessions',
 ...
]
```

### Using Sessions in Views

- When SessionMiddleware is enabled, each HttpRequest object has a session attribute, which behaves like a dictionary:

Setting a session value

```
request.session["fav_color"] =
"blue"
```

Getting a session value:

```
fav_color = request.session["fav_color"]
```

Clearing a session value

```
del request.session["fav_color"]
```

Checking if a session key exists

```
if "fav_color" in request.session:
```

```
...
```

- **Session Usage Rules**

Use normal Python strings for dictionary keys on request.session.

Avoid using keys that start with an underscore, as they are reserved for internal use by Django.

Do not replace request.session with a new object or set its attributes directly.

### 1. Example Views

- Preventing Multiple

Comments: def

```
post_comment(request):
```

```
 if request.method != 'POST':
```

```
 raise Http404('Only POSTs are allowed')
```

```
 if 'comment' not in request.POST:
```

```
 raise Http404('Comment not submitted')
```

```
 if request.session.get('has_commented', False):
```

```
 return HttpResponse("You've already commented.")
```

```
 c = comments.Comment(comment=request.POST['comment'])
```

```
 c.save()
```

```
 request.session['has_commented'] = True
```

```
 return HttpResponse('Thanks for your comment!')
```

### 2. Logging In

```
def login(request):
```

```
 if request.method != 'POST':
```

```
 raise Http404('Only POSTs are allowed')
```

```
 try:
```

```
 m = Member.objects.get(username=request.POST['username'])
```

```
 if m.password == request.POST['password']:
```

```
 request.session['member_id'] = m.id
```

```
 return HttpResponseRedirect('/you-are-logged-in/')
```

```
except Member.DoesNotExist:
```

```
 return HttpResponse("Your username and password didn't match.")
```

### 3. Logging Out:

```
def logout(request):
```

```
 try:
```

```
 del request.session['member_id']
```

```
 except KeyError:
```

```
 pass
```

```
 return HttpResponse("You're logged out.")
```

- Testing Cookie Acceptance
- To test if a browser accepts cookies:

Set the test cookie

```
request.session.set_test_cookie()
```

- Check if the test cookie worked in a subsequent

view if `request.session.test_cookie_worked()`:

```
request.session.delete_test_cookie()
```

```
return HttpResponse("You're logged in.")
```

```
else:
```

```
 return HttpResponse("Please enable cookies and try again.")
```

Sessions Outside of Views

Sessions can also be managed directly through Django's session model:

```
from django.contrib.sessions.models import Session
```

```
s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceed')
```

```
session_data = s.get_decoded()
```

Session Saving Behavior

By default, Django saves the session to the database only if it has been modified:

```
request.session['foo'] = 'bar' # Modified
```

```
del request.session['foo'] # Modified
```

```
request.session['foo'] = {} # Modified
```

```
request.session['foo']['bar'] = 'baz' # Not Modified.
```

- To save the session on every request, set `SESSION_SAVE_EVERY_REQUEST` to `True`.

### Browser-Length vs. Persistent Sessions

- Persistent sessions (default): Cookies are stored for `SESSION_COOKIE_AGE` seconds (default: two weeks).
- Browser-length sessions: Set `SESSION_EXPIRE_AT_BROWSER_CLOSE` to `True` to use browser-length cookies.

### Session Settings

- SESSION\_COOKIE\_DOMAIN: Domain for session cookies.
- SESSION\_COOKIE\_NAME: Name of the session cookie.
- SESSION\_COOKIE\_SECURE: Use secure cookies (only sent via HTTPS).

### Users and Authentication

#### 1. Django Auth/Auth System Overview

- Authentication: Verify user identity using username and password.
- Authorization: Grant permissions to users to perform specific tasks.

#### 2. Components of the Auth/Auth System

- Users: Registered users on the site.
- Permissions: Flags indicating user capabilities.
- Groups: Labels and permissions for multiple users.
- Messages: Queue and display system messages.

#### 3. Enabling Authentication Support

- Ensure the session framework is installed.
- Add 'django.contrib.auth' to INSTALLED\_APPS and run manage.py syncdb.
- Include 'django.contrib.auth.middleware.AuthenticationMiddleware' in MIDDLEWARE\_CLASSES after SessionMiddleware.

# settings.py

```
INSTALLED_APPS = [
```

```
...
```

```
'django.contrib.auth',
```

```
...
```

```
]
```

```
MIDDLEWARE_CLASSES = [
```

```
...
```

```
'django.contrib.sessions.middleware.SessionMiddleware',
```

```
'django.contrib.auth.middleware.AuthenticationMiddleware',
```

```
...
```

```
]
```

**4. Accessing User Information**

- Use request.user to access the logged-in user.
- Check if a user is authenticated with request.user.is\_authenticated()

```
if request.user.is_authenticated():
 # Do something for authenticated
 users. else:
 # Do something for anonymous users.
```

**5. User Object Fields**

- Fields: username, first\_name, last\_name, email, password, is\_staff, is\_active, is\_superuser, last\_login, date\_joined.

**6. User Object Methods**

- **Methods:** is\_authenticated(), is\_anonymous(),  
get\_full\_name(), set\_password(passwd), check\_password(passwd),  
get\_group\_permissions(), get\_all\_permissions(), has\_perm(perm),  
has\_perms(perm\_list), has\_module\_perms(app\_label),  
get\_and\_delete\_messages(), email\_user(subj, msg).

**7. Managing User Groups and Permissions**

```
user.groups.add(group1, group2, ...)
user.groups.remove(group1, group2, ...)
user.groups.clear()
user.permissions.add(permission1, permission2, ...)
user.permissions.remove(permission1, permission2, ...)
user.permissions.clear()
```

**8. Logging In and Out**

- Use auth.authenticate(username, password) to verify credentials.
- Use auth.login(request, user) to log in a user.
- Use auth.logout(request) to log out a user.

```
from django.contrib import auth
def login_view(request):
 username = request.POST.get('username', "")
```

```
password = request.POST.get('password', "")
user = auth.authenticate(username=username,
password=password) if user is not None and user.is_active:
 auth.login(request, user)
 return HttpResponseRedirect("/account/loggedin/")
else:
 return HttpResponseRedirect("/account/invalid/")
def logout_view(request):
 auth.logout(request)
 return HttpResponseRedirect("/account/loggedout/")
```

### **9. Built-in View Functions for Login and Logout**

- Add URL patterns for login and logout views: (r'^accounts/login/\$', login), (r'^accounts/logout/\$', logout).
- Customize login templates and redirect URLs using hidden fields and GET parameters.

### **10. Restricting Access to Logged-in Users**

- Use request.user.is\_authenticated() to check access in views.
- Use login\_required decorator for views to ensure user authentication.

### **11. Restricting Access Based on User Permissions**

- Check permissions directly in views: request.user.has\_perm('polls.can\_vote').
- Use user\_passes\_test and permission\_required decorators to enforce permissions.

### **12. Managing Users, Permissions, and Groups via Admin Interface**

- Admin interface provides an easy way to manage users and their permissions.
- Low-level APIs available for absolute control over user management.

### **13. Creating and Managing Users Programmatically**

- Create users using User.objects.create\_user(username, email, password).
- Change passwords using user.set\_password(new\_password).

### **14. Handling User Registration**

- Implement custom views for user registration using UserCreationForm.
- Example view provided for user registration with form validation and template rendering.

### 15. Using Authentication Data in Templates

- `{{ user }}` template variable for accessing the current user.
- `{{ perms }}` template variable for checking user permissions within templates.