Django Signals, ORM, and Caching Manual for Beginners

ALX Backend Python

June 2025

Contents

1 Introduction

This manual provides a beginner-friendly guide to **Django Signals**, **Object-Relational Mapper (ORM)**, **Advanced ORM Techniques**, and **Basic Caching**. It includes a production-ready messaging app implementation, with senior-level code that is heavily commented to explain design decisions. External resources are provided for deeper learning.

2 Event Listeners Using Django Signals

2.1 What Are Signals?

Signals are like a news broadcast: when an event occurs (e.g., a message is sent), a signal is sent, and listeners (functions) react (e.g., create a notification). This decouples app components, improving modularity.

2.2 Key Signals

- `pre_save`/`post_save`: Before/after saving a model.
- `pre_delete`/`post_delete`: Before/after deleting a model.
- `m2m_changed`: For many-to-many field changes.

2.3 Best Practices

- Use lightweight receivers; delegate heavy logic to services.
- Register signals explicitly with `@receiver`.
- Avoid signal overuse to prevent debugging complexity.
- Test signals in isolation by disconnecting them.

2.4 Resources

- Django Signals Docs: https://docs.djangoproject.com/en/5.0/topics/signals/
- Blog: https://simpleisbetterthancomplex.com/tutorial/2016/07/28/how-to-create-dja
  html
- Video: https://www.youtube.com/watch?v=5rG7iE4P4Gk

3 Django ORM Basics

3.1 What is the ORM?

The ORM is like a librarian who fetches books (data) using plain English (Python) instead of library codes (SQL). It simplifies database interactions, making development

faster and safer.

3.2 Common Operations

- **Create**: `Model.objects.create(...)`
- **Retrieve**: `.get()`, `.filter()`, `.all()`
- **Update**: `.save()`, `.update()`
- **Delete**: `.delete()`

3.3 Best Practices

- Handle exceptions (e.g., `DoesNotExist`).
- Use specific queries to minimize data retrieval.
- Add indexes for frequently queried fields.

3.4 Resources

- Django ORM Docs: https://docs.djangoproject.com/en/5.0/topics/db/queries/
- Blog: https://realpython.com/django-orm-introduction/
- Video: https://www.youtube.com/watch?v=9yBXXa0dTQU

4 Advanced ORM Techniques

4.1 Why Optimize?

Inefficient queries (e.g., N+1 problem) slow down apps. Advanced techniques reduce database hits, improving performance.

4.2 Key Techniques

- `select_related()`: Fetches foreign key data in one query.
- `prefetch_related()`: Optimizes many-to-many relationships.
- `annotate()`: Adds computed fields (e.g., counts).
- **Custom Managers/Querysets**: Encapsulate reusable logic.

4.3 Best Practices

- Profile queries with Django Debug Toolbar.
- Use `.only()`/`.defer()` for selective field loading.
- Add database indexes strategically.

4.4 Resources

- Django Optimization Docs: https://docs.djangoproject.com/en/5.0/topics/db/optimization/
- Blog: https://haki.benfred.com/django-orm-optimization/
- Video: https://www.youtube.com/watch?v=8JShj6vaT-k

5 Basic Caching

5.1 What is Caching?

Caching stores results of expensive operations (e.g., database queries) for quick reuse, like keeping a pre-cooked meal ready to serve.

5.2 Methods

- `@cache_page`: Caches entire views.
- `% cache %`: Caches template fragments.
- `cache.set()/get()`: Manual caching.

5.3 Best Practices

- Use short timeouts for dynamic data.
- Invalidate cache on data changes (e.g., via signals).
- Monitor cache hit/miss ratios.

5.4 Resources

- Django Caching Docs: https://docs.djangoproject.com/en/5.0/topics/cache/
- Blog: https://realpython.com/caching-in-django-with-redis/
- Video: https://www.youtube.com/watch?v=zvQJKh0vUsA

6 Implementation Tasks

Below are senior-level implementations for the messaging app, with extensive comments and docstrings.

6.1 Task 0: Signals for User Notifications

Automate notifications for new messages.

```
# messaging/models.py
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone
```

```
5
6  class MessageQuerySet(models.QuerySet):
7      """Custom queryset for Message model to encapsulate reusable
           queries."""
8      def for_user(self, user):
9          # Filters messages where user is sender or receiver
10         return self.filter(models.Q(sender=user) | models.Q(receiver=
               user))
11     def unread(self):
12         # Filters unread messages
13         return self.filter(read=False)
14
15 class Message(models.Model):
16     """Represents a chat message with threading support."""
17     sender = models.ForeignKey(
18         User,
19         on_delete=models.CASCADE,
20         related_name='sent_messages',
21         help_text="User who sent the message."
22     )
23     receiver = models.ForeignKey(
24         User,
25         on_delete=models.CASCADE,
26         related_name='received_messages',
27         help_text="User receiving the message."
28     )
29     content = models.TextField(max_length=5000, help_text="Message
           content.")
30     timestamp = models.DateTimeField(default=timezone.now, db_index=
           True)
31     read = models.BooleanField(default=False, help_text="Read status.")
32     edited = models.BooleanField(default=False, help_text="Edit status.
           ")
33     parent_message = models.ForeignKey(
34         'self',
35         on_delete=models.CASCADE,
36         null=True,
37         blank=True,
38         related_name='replies',
39         help_text="Parent message for threaded replies."
40     )
41     objects = MessageQuerySet.as_manager()  # Custom manager
42
43     class Meta:
44         ordering = ['timestamp']
45         indexes = [
46             models.Index(fields=['sender', 'receiver', 'timestamp']),
47             # Improves query performance for user-specific message
                   lookups
48         ]
49
50 class Notification(models.Model):
51     """Stores user notifications for messages."""
52     user = models.ForeignKey(
53         User,
54         on_delete=models.CASCADE,
55         related_name='notifications',
56         help_text="Recipient of the notification."
```

```
    )
    message = models.ForeignKey(
        Message,
        on_delete=models.CASCADE,
        help_text="Related message."
    )
    created_at = models.DateTimeField(default=timezone.now, db_index=
        True)
    is_read = models.BooleanField(default=False)

    class Meta:
        ordering = ['-created_at']
        indexes = [models.Index(fields=['user', 'created_at'])]

# messaging/services.py
from django.db import transaction
from .models import Notification

class NotificationService:
    """Handles notification creation logic to keep signals lean."""
    @staticmethod
    @transaction.atomic
    def create_notification(message):
        """Creates a notification for the message receiver."""
        # Atomic transaction ensures data consistency
        Notification.objects.create(
            user=message.receiver,
            message=message
        )

# messaging/signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import Message
from .services import NotificationService

@receiver(post_save, sender=Message)
def handle_new_message(sender, instance, created, **kwargs):
    """Triggers notification creation for new messages."""
    if created:  # Only for new messages, not updates
        # Delegate to service to keep signal handler lightweight
        NotificationService.create_notification(instance)

# messaging/apps.py
from django.apps import AppConfig

class MessagingConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'messaging'
    def ready(self):
        # Imports signals to ensure registration at app startup
        import messaging.signals

# messaging/admin.py
from django.contrib import admin
from .models import Message, Notification

@admin.register(Message)
```

```python
class MessageAdmin(admin.ModelAdmin):
    list_display = ['sender', 'receiver', 'content_preview', 'timestamp']
    list_filter = ['timestamp', 'read']
    search_fields = ['content']
    def content_preview(self, obj):
        # Truncates content for admin display
        return obj.content[:50] + '...' if len(obj.content) > 50 else obj.content

@admin.register(Notification)
class NotificationAdmin(admin.ModelAdmin):
    list_display = ['user', 'message', 'created_at', 'is_read']
    list_filter = ['is_read', 'created_at']

# messaging/tests.py
from django.test import TestCase
from django.contrib.auth.models import User
from .models import Message, Notification
from .services import NotificationService

class NotificationSignalTests(TestCase):
    def setUp(self):
        # Creates test users
        self.sender = User.objects.create_user(username='alice', password='pass123')
        self.receiver = User.objects.create_user(username='bob', password='pass123')
    def test_notification_on_message_creation(self):
        """Verifies notification is created when a message is saved."""
        message = Message.objects.create(
            sender=self.sender,
            receiver=self.receiver,
            content="Hello, Bob!"
        )
        notification = Notification.objects.get(user=self.receiver, message=message)
        self.assertEqual(notification.user, self.receiver)
        self.assertFalse(notification.is_read)
```

6.2 Task 1: Signal for Logging Message Edits

Log message edits with history.

```python
# messaging/models.py (add to existing)
class MessageHistory(models.Model):
    """Stores historical versions of edited messages."""
    message = models.ForeignKey(
        Message,
        on_delete=models.CASCADE,
        related_name='history',
        help_text="Message being edited."
    )
    old_content = models.TextField(help_text="Previous content.")
    edited_at = models.DateTimeField(default=timezone.now, db_index=True)

```

```
    class Meta:
        ordering = ['-edited_at']
        indexes = [models.Index(fields=['message', 'edited_at'])]

# messaging/services.py (add to existing)
class MessageHistoryService:
    """Handles message edit logging."""
    @staticmethod
    @transaction.atomic
    def log_edit(message, old_content):
        """Logs the old content of an edited message."""
        MessageHistory.objects.create(
            message=message,
            old_content=old_content
        )

# messaging/signals.py (add to existing)
from django.db.models.signals import pre_save
from .models import Message
from .services import MessageHistoryService

@receiver(pre_save, sender=Message)
def handle_message_edit(sender, instance, **kwargs):
    """Logs old content before a message update."""
    if instance.pk:  # Only for updates
        try:
            old_message = Message.objects.get(pk=instance.pk)
            if old_message.content != instance.content:
                # Only log if content changed
                MessageHistoryService.log_edit(instance, old_message.
                    content)
                instance.edited = True
        except Message.DoesNotExist:
            pass  # Graceful handling of edge cases

# messaging/views.py
from django.shortcuts import render, get_object_or_404
from django.contrib.auth.decorators import login_required
from .models import Message

@login_required
def message_history_view(request, message_id):
    """Displays a message's edit history."""
    # Uses get_object_or_404 for clean error handling
    message = get_object_or_404(Message, id=message_id)
    # Optimizes query with select_related
    history = message.history.select_related('message').all()
    return render(request, 'messaging/history.html', {
        'message': message,
        'history': history
    })

# messaging/templates/messaging/history.html
<h1>Message History</h1>
<p><strong>Current:</strong> {{ message.content }}</p>
<ul>
{% for entry in history %}
    <li>{{ entry.old_content }} (Edited: {{ entry.edited_at }})</li>
```

```
{% endfor %}
</ul>
```

6.3 Task 2: Signals for Deleting User Data

Clean up user data on account deletion.

```
# messaging/services.py (add to existing)
class UserCleanupService:
    """Handles cleanup of user-related data."""
    @staticmethod
    @transaction.atomic
    def cleanup_user_data(user):
        """Deletes messages, notifications, and history for a user."""
        # Atomic transaction ensures consistency
        Message.objects.filter(sender=user).delete()
        Message.objects.filter(receiver=user).delete()
        Notification.objects.filter(user=user).delete()
        MessageHistory.objects.filter(message__sender=user).delete()
        MessageHistory.objects.filter(message__receiver=user).delete()

# messaging/signals.py (add to existing)
from django.db.models.signals import post_delete
from django.contrib.auth.models import User
from .services import UserCleanupService

@receiver(post_delete, sender=User)
def handle_user_deletion(sender, instance, **kwargs):
    """Triggers cleanup when a user is deleted."""
    UserCleanupService.cleanup_user_data(instance)

# messaging/views.py (add to existing)
from django.shortcuts import redirect
from django.contrib.auth.decorators import login_required

@login_required
def delete_user(request):
    """Handles user account deletion."""
    if request.method == 'POST':
        # Deletes user, triggering post_delete signal
        request.user.delete()
        return redirect('home')
    return render(request, 'messaging/delete_account.html')

# messaging/templates/messaging/delete_account.html
<h1>Delete Account</h1>
<form method="post">
    {% csrf_token %}
    <p>Are you sure? This is permanent.</p>
    <button type="submit">Delete</button>
</form>
```

6.4 Task 3: Threaded Conversations with Advanced ORM

Support threaded replies with optimized queries.

```python
# messaging/views.py (add to existing)
from django.db.models import Prefetch
from .models import Message

@login_required
def threaded_conversation_view(request, message_id):
    """Displays a message and its threaded replies."""
    # Optimizes query with select_related and prefetch_related
    message = get_object_or_404(
        Message.objects.select_related('sender', 'receiver').prefetch_related(
            Prefetch(
                'replies',
                queryset=Message.objects.select_related('sender', 'receiver').order_by('timestamp')
            )
        ),
        id=message_id
    )
    return render(request, 'messaging/thread.html', {'message': message})

# messaging/templates/messaging/thread.html
<h1>Conversation Thread</h1>
<p><strong>{{ message.sender.username }}:</strong> {{ message.content }}</p>
<h2>Replies</h2>
<ul>
{% for reply in message.replies.all %}
    <li><strong>{{ reply.sender.username }}:</strong> {{ reply.content }} ({{ reply.timestamp }})</li>
{% endfor %}
</ul>
```

6.5 Task 4: Custom ORM Manager for Unread Messages

Filter unread messages efficiently.

```python
# messaging/models.py (update Message model)
# Already included in Task 0 with MessageQuerySet and unread manager

# messaging/views.py (add to existing)
@login_required
def inbox_view(request):
    """Displays unread messages for the user."""
    # Uses custom manager and optimizes with select_related/only
    messages = Message.unread.select_related('sender').only(
        'sender__username', 'content', 'timestamp'
    ).filter(receiver=request.user)
    return render(request, 'messaging/inbox.html', {'messages': messages})

# messaging/templates/messaging/inbox.html
<h1>Inbox</h1>
<ul>
{% for message in messages %}
```

```
    <li><strong>{{ message.sender.username }}:</strong> {{ message.content }} ({{ message.timestamp }})</li>
{% endfor %}
</ul>
```

6.6 Task 5: Basic View Caching

Cache conversation view for performance.

```
# messaging_app/messaging_app/settings.py
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.redis.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        # Redis for production-grade caching
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}

# messaging/views.py (add to existing)
from django.views.decorators.cache import cache_page
from django.core.cache import cache
from .models import Message

@cache_page(60, key_prefix='conversation_view')
def conversation_view(request):
    """Displays messages, cached for 60 seconds."""
    # Generates cache key based on user for personalized caching
    cache_key = f'conversation:{request.user.id}'
    messages = cache.get(cache_key)
    if not messages:
        # Cache miss: fetch from database
        messages = Message.objects.select_related('sender', 'receiver').filter(
            models.Q(sender=request.user) | models.Q(receiver=request.user)
        ).order_by('timestamp')
        # Stores in cache for 60 seconds
        cache.set(cache_key, messages, 60)
    return render(request, 'messaging/conversation.html', {'messages': messages})

# messaging/signals.py (add to existing)
@receiver(post_save, sender=Message)
def invalidate_conversation_cache(sender, instance, **kwargs):
    """Invalidates conversation cache on new message."""
    # Clears cache for sender and receiver
    cache.delete(f'conversation:{instance.sender.id}')
    cache.delete(f'conversation:{instance.receiver.id}')

# messaging/templates/messaging/conversation.html
<h1>Messages</h1>
<ul>
{% for message in messages %}
```

```
45     <li><strong>{{ message.sender.username }}:</strong> {{ message.
           content }} ({{ message.timestamp }})</li>
46 {% endfor %}
47 </ul>
```

7 Conclusion

This manual equips you with production-ready skills in Django Signals, ORM, and Caching. The messaging app demonstrates senior-level engineering practices. Extend the app (e.g., add real-time messaging) to deepen your learning.