Collaborative Text Editor Documentation

Kenisha and Ziv

December 5, 2024

Contents

Chapter 1

Introduction

1.1 Overview

This documentation provides a comprehensive guide to the development, deployment, and usage of the **Collaborative Text Editor** project. This application allows multiple users to edit documents collaboratively in real-time, incorporating features like commenting, version control, and the ability to download documents in Word format.

1.2 Features

- **Real-time Collaborative Editing**: Multiple users can edit the same document simultaneously.
- **User Authentication and Authorization**: Secure login and registration system using Django's built-in authentication.
- **Commenting System**: Users can add comments and suggest edits on specific text ranges.
- **Version Control**: Save, restore, and delete different versions of a document.
- **Download as Word Document**: Export the document in `.docx` format.

1.3 Technologies Used

- **Django**: Web framework for building the backend.
- **Django Channels**: Extends Django to handle WebSockets for real-time communication.
- **Redis**: In-memory data structure store used as the channel layer backend.
- **QuillJS**: Rich text editor integrated into the frontend.
- **Daphne**: ASGI server used to run the application.
- **JavaScript**: For frontend interactivity and WebSocket communication.
- **HTML/CSS**: For structuring and styling the web pages.

Chapter 2

User Guide

2.1 Getting Started

2.1.1 Registration

To use the Collaborative Text Editor, you need to create an account.

1. Navigate to the registration page by clicking on `register()` on the homepage.
2. Fill in the registration form:
   - **Username**
   - **Email**
   - **Password**
   - **Confirm Password**
3. Click on `initializeAccount()` to complete the registration.

2.1.2 Login

If you already have an account:

1. Click on `login()`.
2. Enter your **Username** and **Password**.
3. Click on `authenticate()` to log in.

2.2 Dashboard

Upon logging in, you are directed to your **Workspace**. Here you can:

- View all documents you have access to.
- Create a new document by clicking `new Document()`.
- Edit or delete existing documents.

2.3 Creating a New Document

1. Click on `new Document()`.
2. Provide a **Title** for your document.
3. Optionally, add **Collaborators** by selecting users from the list.
4. Click on `createDocument()` to create the document.

2.4 Editing a Document

2.4.1 Real-time Collaboration

When editing a document:

- Changes are synchronized in real-time among all collaborators.
- Active users are displayed, each with a unique cursor color.
- A rich text editor is provided for formatting and structuring your document.

2.4.2 Adding Comments

To add a comment:

1. Highlight the text you want to comment on.
2. Click on the `comment` button in the toolbar.
3. Enter your comment and optionally suggest an edit.
4. Click `Submit` to post the comment.

2.4.3 Version Control

The editor supports version control:

- Click on the `Save` button to save the current state as a new version.
- Provide a description for the version.
- Access version history via the `Version History` button.
- Preview, restore, or delete versions as needed.

2.4.4 Sharing Documents

To share a document:

1. Click on the `Share` button.
2. Add or remove collaborators.
3. Click `Save Changes` to update the collaborators.

2.5 Downloading Documents

You can download your document as a Word file:

1. Click on the `Download as Doc` button.
2. The document will be downloaded in `.docx` format.

Chapter 3

Technical Documentation

3.1 Project Structure

The project is organized as follows:

- **collaborative_text_editor/**: Root project directory containing settings and configurations.
  - `__init__.py`
  - `asgi.py`
  - `settings.py`
  - `urls.py`
  - `wsgi.py`
- **editor/**: Django app containing models, views, forms, and routing.
  - `__init__.py`
  - `admin.py`
  - `apps.py`
  - `consumers.py`
  - `forms.py`
  - `models.py`
  - `routing.py`
  - `tests.py`
  - `urls.py`
  - `views.py`
  - `migrations/`
- **templates/**: HTML templates used by the application.
- **static/**: Static files like CSS and JavaScript.
- `manage.py`: Django's command-line utility.

3.2 Dependencies

- **Django ¿= 3.2**
- **Django Channels ¿= 3.0**
- **Channels Redis ¿= 3.0**
- **Redis**: For channel layers.
- **QuillJS**: Rich text editor.
- **Daphne**: ASGI server.
- **python-docx**: For exporting documents to Word format.

3.3 ASGI Configuration

3.3.1 ASGI Application

The `asgi.py` file configures the ASGI application for handling both HTTP and WebSocket protocols. It sets up Django and wraps the HTTP application with `ASGIStaticFilesHandler` to serve static files during development.

```
import os

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'collaborative_text_editor.settings')

import django
django.setup()

from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from django.core.asgi import get_asgi_application
from django.contrib.staticfiles.handlers import ASGIStaticFilesHandler
import editor.routing

application = ProtocolTypeRouter({
    "http": ASGIStaticFilesHandler(get_asgi_application()),
    "websocket": AuthMiddlewareStack(
        URLRouter(
            editor.routing.websocket_urlpatterns
        )
    ),
})
```

Listing 3.1: collaborative_text_editor/asgi.py

3.3.2 Channel Layers

Configured in `settings.py` to use Redis as the backend for channel layers, which enables message passing between different instances of the application.

```
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('localhost', 6379)],
            'capacity': 10000,
            'group_expiry': 60 * 60,
        },
    },
}
```

Listing 3.2: collaborative_text_editor/settings.py

3.4 Models

3.4.1 Document Model

Represents a collaborative document with fields for title, content, collaborators, and timestamps.

```
class Document(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField(blank=True, default='')  # Store content as JSON
    collaborators = models.ManyToManyField(User, related_name='documents')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

Listing 3.3: editor/models.py

3.4.2 DocumentVersion Model

Stores versions of a document for version control, including content, description, timestamps, and the user who saved the version.

```
class DocumentVersion(models.Model):
    document = models.ForeignKey(Document, on_delete=models.CASCADE, related_name='versions')
    content = models.TextField()
    description = models.CharField(max_length=255, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
```

3.4.3 Comment Model

Represents comments made on specific text ranges within a document. Supports suggesting edits and tracking whether the comment has been resolved or the edit has been applied.

```
class Comment(models.Model):
    document = models.ForeignKey(Document, on_delete=models.CASCADE, related_name='comments')
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    range = models.JSONField()
    content = models.TextField()
    suggested_edit = models.TextField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    resolved = models.BooleanField(default=False)
    edit_applied = models.BooleanField(default=False)
```

3.5 Views

3.5.1 User Registration

Handles user registration using a custom `RegisterForm`.

```
def register(request):
    if request.method == 'POST':
        form = RegisterForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect('document_list')
    else:
```

```
        form = RegisterForm()
    return render(request, 'register.html', {'form': form})
```

Listing 3.4: editor/views.py

3.5.2 Document List

Displays the list of documents the user has access to, filtering by the user's collaborations.

```
@login_required
def document_list(request):
    documents = Document.objects.filter(collaborators=request.user)
    return render(request, 'document_list.html', {'documents': documents})
```

3.5.3 Document Editing

Handles document editing, including updating the title via AJAX, updating collaborators, and rendering the editor template.

```
@csrf_exempt
@login_required
def document_edit(request, pk):
    document = get_object_or_404(Document, pk=pk, collaborators=request.user)
    if request.method == 'POST':
        if request.content_type == 'application/json':
            data = json.loads(request.body)
            new_title = data.get('title')
            if new_title:
                document.title = new_title
                document.save()
                return JsonResponse({'status': 'success'})
            else:
                return HttpResponseBadRequest('Invalid title')
        elif 'form_type' in request.POST and request.POST.get('form_type') == 'collaborators_form
    ':
            collaborators_form = CollaboratorsForm(request.POST, instance=document)
            if collaborators_form.is_valid():
                collaborators_form.save()
                return redirect('document_edit', pk=document.pk)
            else:
                print(collaborators_form.errors)
        else:
            return HttpResponseBadRequest('Invalid form submission')
    else:
        collaborators_form = CollaboratorsForm(instance=document)
    return render(request, 'editor.html', {'document': document, 'collaborators_form':
    collaborators_form})
```

3.5.4 Version Control Views

Includes saving, listing, restoring, and deleting document versions.

```
@csrf_exempt
@login_required
def save_version(request, pk):
    if request.method == 'POST':
        document = get_object_or_404(Document, pk=pk, collaborators=request.user)
        data = json.loads(request.body)
        content = data.get('content')
        description = data.get('description', '')
        if content:
            DocumentVersion.objects.create(
                document=document,
```

```
                content=json.dumps(content),
                description=description,
                user=request.user
            )
            return JsonResponse({'status': 'success'})
        else:
            return HttpResponseBadRequest('No content provided.')
    else:
        return HttpResponseBadRequest('Invalid request method.')
```

3.6 WebSocket Consumers

3.6.1 DocumentConsumer

Manages real-time document editing, handling WebSocket connections, broadcasting changes, and managing active users.

```
class DocumentConsumer(AsyncWebsocketConsumer):
    active_users_dict = {}

    async def connect(self):
        self.document_id = self.scope['url_route']['kwargs']['document_id']
        self.room_group_name = f'document_{self.document_id}'
        self.user = self.scope["user"]
        self.color = await self.get_user_color(self.user.id)

        if self.user.is_anonymous:
            await self.close()
        else:
            has_permission = await self.check_permission()
            if not has_permission:
                await self.close()
            else:
                await self.add_active_user()
                await self.channel_layer.group_add(
                    self.room_group_name,
                    self.channel_name
                )
                await self.accept()
                document = await self.get_document()
                content = json.loads(document.content) if document.content else {'ops': []}
                await self.send(text_data=json.dumps({
                    'type': 'init',
                    'content': content
                }))
                await self.broadcast_active_users()
```

Listing 3.5: editor/consumers.py

3.6.2 Handling Real-time Updates

Updates to the document are broadcast to all connected clients, ensuring synchronization across sessions.

```
async def receive(self, text_data):
    data = json.loads(text_data)
    message_type = data.get('type')

    if message_type == 'delta':
        delta_ops = data.get('delta')
        if delta_ops:
            await self.update_document_content(delta_ops)
            await self.channel_layer.group_send(
                self.room_group_name,
                {
```

```
                    'type': 'broadcast_delta',
                    'delta': delta_ops,
                    'sender_channel_name': self.channel_name
                })
```

3.7 Front-end Integration

3.7.1 QuillJS Editor

The QuillJS editor is initialized with real-time collaboration features, including cursor synchronization and commenting functionality.

```
Quill.register('modules/cursors', window.QuillCursors);

const quill = new Quill('#editor-container', {
    theme: 'snow',
    modules: {
        cursors: true,
        toolbar: {
            container: [
                // Toolbar options
            ],
            handlers: {
                'comment': function() {
                    // Comment handler
                }
            }
        }
    }
});
```

Listing 3.6: Initializing Quill Editor

3.7.2 WebSocket Connection

Establishes a WebSocket connection for real-time collaboration, handling messages for content updates, cursor positions, and comments.

```
const wsScheme = window.location.protocol === "https:" ? "wss" : "ws";
const documentId = "{{ document.pk }}";
const socketUrl = `${wsScheme}://${window.location.host}/ws/document/${documentId}/`;
const socket = new WebSocket(socketUrl);

socket.onmessage = function(e) {
    const data = JSON.parse(e.data);
    if (data.type === 'delta') {
        // Apply delta to the editor
    }
};
```

Listing 3.7: WebSocket Connection

3.8 Templates and Static Files

3.8.1 Base Template

Defines the base HTML structure, including the header, footer, and main content blocks.

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>CollabDocs</title>
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
    {% block extra_head %}{% endblock %}
</head>
<body>
    {% block content %}{% endblock %}
    {% block extra_script %}{% endblock %}
</body>
</html>
```

Listing 3.8: templates/base.html

3.8.2 Editor Template

The template for the document editor page, including the QuillJS editor and modals for version control and sharing.

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

{% block extra_head %}
<link href="https://cdn.quilljs.com/1.3.6/quill.snow.css" rel="stylesheet">
{% endblock %}

{% block content %}
<div id="editor-container"></div>
<!-- Additional modals and components -->
{% endblock %}

{% block extra_script %}
<script src="https://cdn.quilljs.com/1.3.6/quill.js"></script>
<script>
// JavaScript code to initialize Quill and WebSocket
</script>
{% endblock %}
```

Listing 3.9: templates/editor.html

3.9 Routing

3.9.1 URL Configuration

Defines URL patterns for views, including authentication, document operations, and version control.

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

urlpatterns = [
    path('register/', views.register, name='register'),
    path('', views.document_list, name='document_list'),
    path('document/new/', views.document_create, name='document_create'),
    path('document/<int:pk>/edit/', views.document_edit, name='document_edit'),
    # Additional URL patterns
]
```

Listing 3.10: editor/urls.py

3.9.2 WebSocket Routing

Configures WebSocket routes for real-time communication using Django Channels.

```python
from django.urls import re_path
from . import consumers

websocket_urlpatterns = [
    re_path(r'^ws/document/(?P<document_id>\d+)/$', consumers.DocumentConsumer.as_asgi()),
]
```

Listing 3.11: editor/routing.py

3.10 Forms

3.10.1 User Registration Form

Extends Django's `UserCreationForm` to include an email field.

```python
class RegisterForm(UserCreationForm):
    email = forms.EmailField(required=True)

    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']
```

Listing 3.12: editor/forms.py

3.10.2 Document Form

Used for creating new documents, allowing the selection of collaborators.

```python
class DocumentForm(forms.ModelForm):
    collaborators = forms.ModelMultipleChoiceField(
        queryset=User.objects.all(),
        widget=forms.CheckboxSelectMultiple,
        required=False,
        label="Share with"
    )

    class Meta:
        model = Document
        fields = ['title', 'collaborators']
```

3.11 Settings

Key settings in `settings.py`:

3.11.1 Installed Apps

Includes necessary Django apps and third-party apps like Channels.

```python
INSTALLED_APPS = [
    'channels',          # For WebSocket support
    'django.contrib.admin',
    'django.contrib.auth',
    # ...
    'editor',            # Your app
]
```

3.11.2 Middleware

Standard Django middleware components.

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    # ...
]
```

3.11.3 Templates

Configure template directories and context processors.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        # ...
    },
]
```

3.11.4 Static Files

Configure static files directories.

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

3.11.5 ASGI Application

Specify the ASGI application for Channels.

```
ASGI_APPLICATION = 'collaborative_text_editor.asgi.application'
```

3.12 Authentication and Authorization

- Uses Django's built-in authentication system.
- Views are protected using `@login_required`.
- Document access is restricted to collaborators via query filters.
- The `CollaboratorsForm` allows updating collaborators for a document.

3.13 Deployment Considerations

- Ensure Redis is running for channel layers.
- Use an ASGI server like **Daphne** for deployment.
- Configure `ALLOWED_HOSTS` and security settings in `settings.py`.
- Collect static files using `python manage.py collectstatic` for production.
- For running the server, use the following command:

```
    daphne -b 0.0.0.0 -p 8000 collaborative_text_editor.asgi:application

```

Chapter 4

Appendix

4.1 Running the Application

1. Install dependencies using `pip`:

```
    pip install -r requirements.txt

```

2. Start the Redis server:

```
    redis-server

```

3. Apply database migrations:

```
    python manage.py migrate

```

4. Create a superuser (optional):

```
    python manage.py createsuperuser

```

5. Collect static files (for production):

```
    python manage.py collectstatic

```

6. Run the application using Daphne:

```
    daphne -b 0.0.0.0 -p 8000 collaborative_text_editor.asgi:application

```

7. Access the application at `http://localhost:8000`

4.2 Requirements File

An example `requirements.txt`:

```
Django>=3.2
channels>=3.0
channels_redis>=3.0
redis>=3.5
python-docx>=0.8
```

4.3 Future Enhancements

- Implement more granular permissions for collaborators (e.g., read-only access).
- Enhance the commenting system with email notifications.
- Optimize performance for large documents by implementing pagination or virtual scrolling.
- Add support for real-time editing of images and other media types.
- Integrate user avatars and profiles for a more personalized experience.

4.4 Additional Notes

4.4.1 Security Considerations

Ensure that:

- `DEBUG` is set to `False` in production.
- Proper `ALLOWED_HOSTS` are configured.
- Sensitive information is kept out of version control (e.g., secret keys).

4.4.2 Scaling the Application

For scaling the application:

- Use a production-ready web server like **Gunicorn** for HTTP requests.
- Use **Daphne** or **Uvicorn** for handling ASGI applications.
- Implement load balancing for handling increased traffic.
- Use a cloud-based Redis service for better performance and reliability.

4.4.3 Testing

Consider writing unit tests and integration tests to ensure the reliability of the application. Django's testing framework can be used for this purpose.

4.4.4 Documentation

Maintain up-to-date documentation, both for users and developers, to facilitate ease of use and contribution to the project.