**FRONTEND DEVELOPMENT**

To create a responsive webpage with the specified features, we can use HTML, CSS, and JavaScript. Below is a complete example that includes a fixed navbar, collapsible left menu, main content area, right-side panel, and a footer. Additionally, a JavaScript function is provided to adjust the page width based on the screen size.

**1.HTML STRUCTURE:**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>Responsive Webpage</title>**

**<link rel="stylesheet" href="styles.css">**

**</head>**

**<body>**

**<div class="navbar">**

**<h1>My Website</h1>**

**<button id="toggleMenu">Toggle Menu</button>**

**</div>**

**<div class="container">**

**<div class="left-menu" id="leftMenu">**

**<h2>Left Menu</h2>**

**<ul>**

**<li>Item 1</li>**

**<li>Item 2</li>**

**<li>Item 3</li>**

**</ul>**

**</div>**

**<div class="main-content">**

**<h2>Main Content Area</h2>**

**<p>This is where the main content goes.</p>**

**</div>**

**<div class="right-panel">**

**<h2>Right Panel</h2>**

**<p>This is the right-side panel.</p>**

**</div>**

**</div>**

**<footer>**

**<p>Footer Content</p>**

**</footer>**

**<script src="script.js"></script>**

**</body>**

**</html>**

**CSS(Style.css):**

**\* {**

**box-sizing: border-box;**

**}**

**body {**

**margin: 0;**

**font-family: Arial, sans-serif;**

**}**

**.navbar {**

**position: fixed;**

**top: 0;**

**left: 0;**

**right: 0;**

**background-color: #333;**

**color: white;**

**padding: 10px;**

**display: flex;**

**justify-content: space-between;**

**align-items: center;**

**}**

**.container {**

**display: flex;**

**margin-top: 60px; /\* Space for the fixed navbar \*/**

**}**

**.left-menu {**

**width: 200px;**

**background-color: #f4f4f4;**

**padding: 15px;**

**transition: width 0.3s;**

**}**

**.left-menu.collapsed {**

**width: 0;**

**overflow: hidden;**

**}**

**.main-content {**

**flex: 1;**

**padding: 15px;**

**}**

**.right-panel {**

**width: 200px;**

**background-color: #f4f4f4;**

**padding: 15px;**

**}**

**footer {**

**background-color: #333;**

**color: white;**

**text-align: center;**

**padding: 10px;**

**position: relative;**

**bottom: 0;**

**width: 100%;**

**}**

**2.JAVA SCRIPT(Script.js):**

**document.getElementById('toggleMenu').addEventListener('click', function() {**

**const leftMenu = document.getElementById('leftMenu');**

**leftMenu.classList.toggle('collapsed');**

**});**

**function adjustPageWidth() {**

**const width = window.innerWidth;**

**const body = document.body;**

**if (width >= 992 && width <= 1600) {**

**body.style.width = '90%';**

**} else if (width >= 700 && width < 992) {**

**body.style.width = '80%';**

**} else if (width >= 600 && width < 700) {**

**body.style.width = '75%';**

**} else if (width <= 600) {**

**body.style.width = '50%';**

**} else {**

**body.style.width = '100%'; // Default for larger screens**

**}**

**}**

**// Adjust page width on load and on resize**

**window.addEventListener('load', adjustPageWidth);**

**window.addEventListener('resize', adjustPageWidth);**

**DJANGO**

Creating a chat application using Django with the specified features involves several steps, including setting up the Django project, creating models for users and messages, implementing user authentication, and using WebSockets for real-time chat functionality. Below is a step-by-step guide to help you build this application.

**STEP 1:SET UP YOUR DJANGO PROJECT**

1. **Install Django and channels:** Make sure you have Django and Django channels. You can do this using pip:

**pip install django channels channels-redis**

1. **Create a new Django project:**Create a Django project and an app for the chat functionality.

**django-admin startproject chatapp**

**cd chatapp**

**django-admin startapp chat**

1. **Update ‘setting.py’:** Add the new app and channels to your **‘INSTALLED\_APPS’** in ‘settings.py’.

**INSTALLED\_APPS = [**

**...**

**'chat',**

**'channels',**

**]**

**ASGI\_APPLICATION = 'chatapp.asgi.application'**

**4.Set up Redis:**If you want to use Redis as the channel layer , make sure you have Redis installed and running.Update your ’setting py’ to include the channel layer configuration:

**CHANNEL\_LAYERS = {**

**'default': {**

**'BACKEND': 'channels\_redis.core.RedisChannelLayer',**

**'CONFIG': {**

**"hosts": [('127.0.0.1', 6379)],**

**},**

**},**

**}**

**STEP 2: CREATE MODELS**

In **‘chat/models.py’**,createcmodels for user and message.

**from django.db import models**

**from django.contrib.auth.models import User**

**class Message(models.Model):**

**sender = models.ForeignKey(User, related\_name='sent\_messages', on\_delete=models.CASCADE)**

**receiver = models.ForeignKey(User, related\_name='received\_messages', on\_delete=models.CASCADE)**

**content = models.TextField()**

**timestamp = models.DateTimeField(auto\_now\_add=True)**

**def \_\_str\_\_(self):**

**return f"{self.sender} to {self.receiver}: {self.content[:20]}"**

**STEP 3 : CREATE VIEWS AND TEMPLATES**

1. **User Authentication:**Use Django ‘s built-in authentication views for sign-up and login .Create a simple sign-up view in **‘ chat/view.py’**.

**from django.shortcuts import render, redirect**

**from django.contrib.auth import login, authenticate**

**from django.contrib.auth.forms import UserCreationForm**

**def signup(request):**

**if request.method == 'POST':**

**form = UserCreationForm(request.POST)**

**if form.is\_valid():**

**user = form.save()**

**login(request, user)**

**return redirect('chat:chatroom')**

**else:**

**form = UserCreationForm()**

**return render(request, 'chat/signup.html', {'form': form})**

1. **Chat Room View:**Create a view to display the chat room.

**from django.contrib.auth.decorators import login\_required**

**from .models import Message**

**@login\_required**

**def chatroom(request):**

**users = User.objects.exclude(id=request.user.id)**

**return render(request, 'chat/chatroom.html', {'users': users})**

1. **URLs:** Update **‘chat/urls.py’.**

**from django.urls import path**

**from .views import signup, chatroom**

**app\_name = 'chat'**

**urlpatterns = [**

**path('signup/', signup, name='signup'),**

**path('chatroom/', chatroom, name='chatroom'),**

**]**

1. **Main URLs:** Include the chat app URLs in your main **‘urls.py**’.

**from django.contrib import admin**

**from django.urls import path, include**

**urlpatterns = [**

**path('admin/', admin.site.urls),**

**path('chat/', include('chat.urls')),**

**]**

**STEP 4: CREATE WEBSOCKET CONSUMER**

Create a Websocket consumer in **‘chat/consumer.py’**.

**import json**

**from channels.generic.websocket import AsyncWebsocketConsumer**

**class ChatConsumer(AsyncWebsocketConsumer):**

**async def connect(self):**

**self.room\_group\_name = f"chat\_{self.scope['user'].username}"**

**await self.channel\_layer.group\_add(self.room\_group\_name, self.channel\_name)**

**await self.accept()**

**async def disconnect(self, close\_code):**

**await self.channel\_layer.group\_discard(self.room\_group\_name, self.channel\_name)**

**async def receive(self, text\_data):**

**data = json.loads(text\_data)**

**message = data['message']**

**sender = self.scope['user']**

**receiver\_username = data['receiver']**

**# Save message to the database**

**from chat.models import Message**

**receiver = User.objects.get(username=receiver\_username)**

**Message.objects.create(sender=sender, receiver=receiver, message=message)**

**# Broadcast message**

**await self.channel\_layer.group\_send(**

**f"chat\_{receiver\_username}",**

**{**

**'type': 'chat\_message',**

**'message': message,**

**'sender': sender.username**

**}**

**)**

**async def chat\_message(self, event):**

**message = event['message']**

**sender = event['sender']**

**await self.send(text\_data=json.dumps({**

**'message': message,**

**'sender': sender**

**}))**

**AWS**

**STEP1: Lambda Function to Add Two Numbers**

1. Go to the AWS management console.
2. Navigate to Lambda.
3. Click on “create function”
4. Choose “Author from scratch”
5. Enter a function name(e.g:”AddTwoNumbers”)
6. Choose a runtime (e.g Python 3.x)
7. Click “create function”.

**STEP2:Add the Code**

**import json**

**def lambda\_handler(event, context):**

**# Extract numbers from the event**

**num1 = event.get('num1', 0)**

**num2 = event.get('num2', 0)**

**# Add the numbers**

**result = num1 + num2**

**# Return the result**

**return {**

**'statusCode': 200,**

**'body': json.dumps({'result': result})**

**}**

**STEP3:Test the Function**

1. Click on”Test”
2. Create a new event with the following JSON:

**{**

**"num1": 5,**

**"num2": 10**

**}**

1. Click “Test” again to execute the function.You should see the result in the output.

**2.AWS Lambda function to store a document or PDF**

**In S3**

**STEP1:**

1.Go to the AWS management console.

2.Navigate to Lambda.

3.Click on “create function”

4.Choose “Author from scratch”

5.Enter a function name(e.g:”UploadTos3”)

6.Choose a runtime (e.g Python 3.x)

7.Click “create function”.

**STEP2:Add the Code**

In the function code editor, replace the default code with the following:

**import json**

**import boto3**

**import base64**

**import os**

**s3 = boto3.client('s3')**

**def lambda\_handler(event, context):**

**# Extract the file name and content from the event**

**file\_name = event.get('file\_name')**

**file\_content = event.get('file\_content') # Base64 encoded content**

**bucket\_name = os.environ['S3\_BUCKET'] # Set this environment variable in Lambda**

**# Decode the base64 content**

**file\_data = base64.b64decode(file\_content)**

**# Upload the file to S3**

**s3.put\_object(Bucket=bucket\_name, Key=file\_name, Body=file\_data)**

**return {**

**'statusCode': 200,**

**'body': json.dumps({'message': 'File uploaded successfully!'})**

**}**

**STEP3: Set up Environment Variable:**

1. In the Lambda function configuration, go on the “Configuration” tab
2. Click on “Environment variables”
3. Add a new Environment variable:

Key: **‘s3\_BUCKET’**

Value:**’<your-s3-bucket-name>’**

**STEP4: Set up IAM Permissions**

Make sure your Lambda function has the necessary permissions to write to the S3 bucket. You can do this by attaching a policy to the Lambda execution role:

1. Go to the IAM service in the AWS Management Console.
2. Find the role associated with the Lambda function.
3. Attach the following policy (or create a custom policy ) that allows **‘s3:putobject**’:

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": "s3:PutObject",**

**"Resource": "arn:aws:s3:::<your-s3-bucket-name>/\*"**

**}**

**]**

**}**

**STEP5: Test the function**

1.Click on”Test”

2.Create a new test event with the following JSON:

**{**

**"file\_name": "example.pdf",**

**"file\_content": "JVBERi0xLjQKJcfs... (base64 encoded content)"**

**}**

Make sure to replace the’**file\_conten**t’ with actual base64 encoded content of a PDF or document.

1. Click “Test “ again to execute the function. You should see a success message in the output.