ContentIQ Backend - Complete Implementation

Create these exact files in your backend folder:

File Structure

backend/
├── main.py
├── config.py
├── mcp_client.py
├── groq_client.py
├── analytics.py
├── content_manager.py
├── models.py
└── requirements.txt

requirements.txt

```
fastapi==0.104.1
uvicorn==0.24.0
python-dotenv==1.0.0
groq==0.4.1
httpx==0.25.2
pydantic==2.5.0
sse-starlette==1.8.2
redis==5.0.1
contentstack==1.8.0
websockets==12.0
```

config.py

```
import os
from dotenv import load_dotenv

load_dotenv()

class Config:
    # Contentstack Configuration
    CONTENTSTACK_API_KEY = os.getenv("CONTENTSTACK_API_KEY",
"blt1234567890")
```

```
    CONTENTSTACK_DELIVERY_TOKEN =
os.getenv("CONTENTSTACK_DELIVERY_TOKEN")
    CONTENTSTACK_ENVIRONMENT = os.getenv("CONTENTSTACK_ENVIRONMENT",
"development")
    CONTENTSTACK_REGION = os.getenv("CONTENTSTACK_REGION", "us")

    # Groq Configuration
    GROQ_API_KEY = os.getenv("GROQ_API_KEY")

    # MCP Configuration
    MCP_COMMAND = "npx"
    MCP_ARGS = ["-y", "@contentstack/mcp"]

    # Redis Cache (optional, use in-memory if not available)
    REDIS_URL = os.getenv("REDIS_URL", "redis://localhost:6379")

    # Analytics
    ENABLE_ANALYTICS = True
    ANALYTICS_BATCH_SIZE = 10
```

models.py

```
from pydantic import BaseModel
from typing import List, Optional, Dict
from datetime import datetime
from enum import Enum

class MessageRole(str, Enum):
    USER = "user"
    ASSISTANT = "assistant"
    SYSTEM = "system"

class ChatMessage(BaseModel):
    role: MessageRole
    content: str
    timestamp: datetime = datetime.now()

class ChatRequest(BaseModel):
    message: str
    session_id: str
    context: Optional[Dict] = {}
    stream: bool = True

class ChatResponse(BaseModel):
    response: str
    content_references: Optional[List[Dict]] = []
    suggestions: Optional[List[str]] = []
```

```
    analytics_tracked: bool = False

class ContentGap(BaseModel):
    query: str
    intent: str
    suggested_content: Dict
    frequency: int = 1
    first_requested: datetime = datetime.now()

class AnalyticsEvent(BaseModel):
    session_id: str
    query: str
    response_time_ms: float
    content_found: bool
    content_type: Optional[str]
    timestamp: datetime = datetime.now()
```

mcp_client.py

```
import subprocess
import json
import asyncio
from typing import Dict, List, Optional
import httpx
from config import Config
import os

class MCPClient:
    def __init__(self):
        self.config = Config()
        self.process = None

    async def initialize(self):
        """Initialize MCP connection"""
        env = {
            "CONTENTSTACK_API_KEY": self.config.CONTENTSTACK_API_KEY,
            "CONTENTSTACK_DELIVERY_TOKEN":
self.config.CONTENTSTACK_DELIVERY_TOKEN,
            "CONTENTSTACK_ENVIRONMENT":
self.config.CONTENTSTACK_ENVIRONMENT,
            "GROUPS": "cma,delivery"
        }

        try:
            self.process = await asyncio.create_subprocess_exec(
                self.config.MCP_COMMAND,
                *self.config.MCP_ARGS,
```

```
            env={**os.environ, **env},
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )
    except Exception as e:
        print(f"MCP initialization failed: {e}")

async def fetch_content(self, content_type: str, query: str) -> List[Dict]:
    """Fetch content from Contentstack based on query"""
    try:
        # Use Contentstack Delivery API
        url = f"https://cdn.contentstack.io/v3/content_types/{content_type}/entries"
        headers = {
            "api_key": self.config.CONTENTSTACK_API_KEY,
            "access_token": self.config.CONTENTSTACK_DELIVERY_TOKEN,
            "environment": self.config.CONTENTSTACK_ENVIRONMENT
        }

        # Search in title and description fields
        params = {
            "query": json.dumps({
                "$or": [
                    {"title": {"$regex": query, "$options": "i"}},
                    {"description": {"$regex": query, "$options": "i"}},
                    {"body": {"$regex": query, "$options": "i"}}
                ]
            })
        }

        async with httpx.AsyncClient() as client:
            response = await client.get(url, headers=headers, params=params)
            if response.status_code == 200:
                data = response.json()
                return data.get("entries", [])
            return []
    except Exception as e:
        print(f"Error fetching content: {e}")
        return []

async def create_draft_content(self, content_type: str, data: Dict) -> Dict:
    """Create draft content in Contentstack when user asks for non-existent content"""
    try:
        url = f"https://api.contentstack.io/v3/content_types/{content_type}/entries"
        headers = {
            "api_key": self.config.CONTENTSTACK_API_KEY,
            "authorization": self.config.CONTENTSTACK_MANAGEMENT_TOKEN,
            "Content-Type": "application/json"
        }
```

```
            entry_data = {
                "entry": {
                    **data,
                    "_metadata": {
                        "created_from": "chat_agent",
                        "auto_generated": True,
                        "needs_review": True
                    }
                }
            }

            async with httpx.AsyncClient() as client:
                response = await client.post(url, headers=headers, json=entry_data)
                if response.status_code == 201:
                    return response.json()
                return {}
        except Exception as e:
            print(f"Error creating draft: {e}")
            return {}
```

groq_client.py

```
from groq import AsyncGroq
from typing import List, AsyncGenerator, Dict
import json
from config import Config
from models import ChatMessage

class GroqClient:
    def __init__(self):
        self.client = AsyncGroq(api_key=Config.GROQ_API_KEY)
        self.model = "llama3-8b-8192"  # Fastest model

    async def generate_response(
        self,
        messages: List[ChatMessage],
        content_context: List[Dict] = None,
        stream: bool = True
    ) -> AsyncGenerator[str, None]:
        """Generate AI response with optional streaming"""
        # Build context from Contentstack content
        system_message = self._build_system_message(content_context)

        # Convert messages to Groq format
        groq_messages = [
            {"role": "system", "content": system_message}
```

```
        ]

        for msg in messages:
            groq_messages.append({
                "role": msg.role.value,
                "content": msg.content
            })

        # Add content context if available
        if content_context:
            context_str = self._format_content_context(content_context)
            groq_messages.append({
                "role": "system",
                "content": f"Available content from CMS:\n{context_str}"
            })

        try:
            if stream:
                stream = await self.client.chat.completions.create(
                    model=self.model,
                    messages=groq_messages,
                    temperature=0.7,
                    max_tokens=1000,
                    stream=True
                )

                async for chunk in stream:
                    if chunk.choices[0].delta.content:
                        yield chunk.choices[0].delta.content
            else:
                response = await self.client.chat.completions.create(
                    model=self.model,
                    messages=groq_messages,
                    temperature=0.7,
                    max_tokens=1000
                )
                yield response.choices[0].message.content
        except Exception as e:
            yield f"Error generating response: {str(e)}"

    def _build_system_message(self, content_context: List[Dict]) -> str:
        return """You are an intelligent content assistant powered by Contentstack CMS.
Your role is to help users find information from the content management system.
If content is not available, politely inform the user and note that the content team has been
notified.
Always be helpful, concise, and accurate. Reference specific content when available."""

    def _format_content_context(self, content_items: List[Dict]) -> str:
```

```
        if not content_items:
            return "No specific content found for this query."

        formatted = []
        for item in content_items[:3]:  # Limit to top 3 results
            formatted.append(f"Title: {item.get('title', 'Untitled')}")
            formatted.append(f"Description: {item.get('description', 'No description')}")
            if 'url' in item:
                formatted.append(f"URL: {item.get('url')}")
            formatted.append("---")

        return "\n".join(formatted)
```

analytics.py

```
import json
from datetime import datetime
from typing import Dict, List
from collections import defaultdict
import asyncio
from models import AnalyticsEvent, ContentGap

class AnalyticsEngine:
    def __init__(self):
        self.events: List[AnalyticsEvent] = []
        self.content_gaps: Dict[str, ContentGap] = {}
        self.query_frequency = defaultdict(int)
        self.response_times = []

    async def track_event(self, event: AnalyticsEvent):
        """Track chat interaction event"""
        self.events.append(event)
        self.query_frequency[event.query.lower()] += 1
        self.response_times.append(event.response_time_ms)

        # Track content gaps
        if not event.content_found:
            gap_key = event.query.lower()
            if gap_key in self.content_gaps:
                self.content_gaps[gap_key].frequency += 1
            else:
                self.content_gaps[gap_key] = ContentGap(
                    query=event.query,
                    intent=self._extract_intent(event.query),
                    suggested_content=self._generate_content_suggestion(event.query)
                )
```

```
        # Trigger analytics export every 10 events
        if len(self.events) % 10 == 0:
            await self.export_analytics()

    def get_dashboard_data(self) -> Dict:
        """Get real-time analytics dashboard data"""
        return {
            "total_queries": len(self.events),
            "average_response_time_ms": sum(self.response_times) / len(self.response_times)
if self.response_times else 0,
            "top_queries": dict(sorted(self.query_frequency.items(), key=lambda x: x[1],
reverse=True)[:10]),
            "content_gaps": [gap.dict() for gap in list(self.content_gaps.values())[:5]],
            "success_rate": len([e for e in self.events if e.content_found]) / len(self.events) * 100
if self.events else 0,
            "recent_queries": [e.query for e in self.events[-5:]],
            "peak_hours": self._calculate_peak_hours()
        }

    def _extract_intent(self, query: str) -> str:
        """Extract user intent from query"""
        query_lower = query.lower()
        if any(word in query_lower for word in ['tour', 'travel', 'visit', 'trip']):
            return 'travel_inquiry'
        elif any(word in query_lower for word in ['price', 'cost', 'budget', 'cheap']):
            return 'pricing_inquiry'
        elif any(word in query_lower for word in ['book', 'reserve', 'schedule']):
            return 'booking_intent'
        else:
            return 'general_inquiry'

    def _generate_content_suggestion(self, query: str) -> Dict:
        """Generate content suggestion for missing content"""
        return {
            "suggested_title": f"Guide to {query}",
            "suggested_type": "article",
            "suggested_tags": self._extract_keywords(query),
            "priority": "high" if self.query_frequency[query.lower()] > 3 else "medium"
        }

    def _extract_keywords(self, query: str) -> List[str]:
        """Extract keywords from query"""
        stop_words = {'the', 'a', 'an', 'to', 'for', 'in', 'on', 'at', 'with', 'about'}
        words = query.lower().split()
        return [w for w in words if w not in stop_words and len(w) > 2]

    def _calculate_peak_hours(self) -> Dict:
        """Calculate peak usage hours"""
```

```
        hours = defaultdict(int)
        for event in self.events:
            hour = event.timestamp.hour
            hours[f"{hour:02d}:00"] += 1
        return dict(sorted(hours.items()))

    async def export_analytics(self):
        """Export analytics to file (in production, this would go to Contentstack)"""
        with open('analytics_export.json', 'w') as f:
            json.dump(self.get_dashboard_data(), f, indent=2, default=str)
```

content_manager.py

```
from typing import Dict, List, Optional
import re

class ContentManager:
    def __init__(self, mcp_client):
        self.mcp_client = mcp_client
        self.content_types = {
            'tour': ['tour', 'travel', 'trip', 'visit', 'destination'],
            'product': ['product', 'item', 'buy', 'purchase', 'shop'],
            'blog': ['article', 'blog', 'post', 'guide', 'how to'],
            'faq': ['question', 'how', 'what', 'why', 'when', 'help']
        }

    def detect_content_type(self, query: str) -> str:
        """Detect content type from user query"""
        query_lower = query.lower()

        for content_type, keywords in self.content_types.items():
            if any(keyword in query_lower for keyword in keywords):
                return content_type

        return 'page'  # default content type

    async def create_draft_for_gap(self, query: str) -> Dict:
        """Create draft content for identified gap"""
        content_type = self.detect_content_type(query)

        draft_data = {
            "title": f"User Requested: {query}",
            "description": f"Auto-generated content based on user query: {query}",
            "status": "draft",
            "tags": self._extract_tags(query),
            "metadata": {
                "auto_generated": True,
```

```
            "source": "chat_agent",
            "original_query": query
        }
    }

    return await self.mcp_client.create_draft_content(content_type, draft_data)

def _extract_tags(self, query: str) -> List[str]:
    """Extract tags from query"""
    words = re.findall(r'\w+', query.lower())
    stop_words = {'the', 'a', 'an', 'to', 'for', 'in', 'on', 'at', 'with'}
    return [w for w in words if w not in stop_words and len(w) > 3]
```

main.py

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import StreamingResponse
from typing import List
import asyncio
import json
import time
from datetime import datetime

from models import ChatRequest, ChatMessage, MessageRole, AnalyticsEvent
from mcp_client import MCPClient
from groq_client import GroqClient
from analytics import AnalyticsEngine
from content_manager import ContentManager

app = FastAPI(title="ContentIQ Chat Platform")

# Enable CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Initialize components
mcp_client = MCPClient()
groq_client = GroqClient()
analytics = AnalyticsEngine()
content_manager = ContentManager(mcp_client)
```

```
# Store chat sessions
chat_sessions = {}

@app.on_event("startup")
async def startup_event():
    """Initialize MCP connection on startup"""
    await mcp_client.initialize()
    print("✅ ContentIQ Chat Platform Started")
    print("📊 Analytics Dashboard: http://localhost:8000/analytics")
    print("💬 Chat Endpoint: http://localhost:8000/chat")

@app.post("/chat")
async def chat_endpoint(request: ChatRequest):
    """Main chat endpoint with streaming support"""
    start_time = time.time()

    # Get or create session
    if request.session_id not in chat_sessions:
        chat_sessions[request.session_id] = []

    # Add user message to session
    user_message = ChatMessage(role=MessageRole.USER, content=request.message)
    chat_sessions[request.session_id].append(user_message)

    # Extract intent and fetch relevant content
    content_type = content_manager.detect_content_type(request.message)
    content_results = await mcp_client.fetch_content(content_type, request.message)

    # Track if content was found
    content_found = len(content_results) > 0

    async def generate():
        """Stream response chunks"""
        response_text = ""
        async for chunk in groq_client.generate_response(
            messages=chat_sessions[request.session_id],
            content_context=content_results,
            stream=request.stream
        ):
            response_text += chunk
            yield f"data: {json.dumps({'chunk': chunk})}\n\n"

        # Save assistant response
        assistant_message = ChatMessage(role=MessageRole.ASSISTANT,
content=response_text)
        chat_sessions[request.session_id].append(assistant_message)

        # Track analytics
```

```
        response_time = (time.time() - start_time) * 1000
        await analytics.track_event(AnalyticsEvent(
            session_id=request.session_id,
            query=request.message,
            response_time_ms=response_time,
            content_found=content_found,
            content_type=content_type
        ))

        # If no content found, create draft
        if not content_found:
            draft = await content_manager.create_draft_for_gap(request.message)
            if draft:
                yield f"data: {json.dumps({'notification': 'Content gap detected. Draft created for review.'})}\n\n"

        yield f"data: {json.dumps({'done': True, 'response_time_ms': response_time})}\n\n"

    if request.stream:
        return StreamingResponse(generate(), media_type="text/event-stream")
    else:
        response = ""
        async for chunk in generate():
            if "chunk" in chunk:
                data = json.loads(chunk.replace("data: ", ""))
                response += data.get("chunk", "")
        return {"response": response, "content_found": content_found}

@app.get("/analytics")
async def get_analytics():
    """Get real-time analytics dashboard data"""
    return analytics.get_dashboard_data()

@app.websocket("/ws/analytics")
async def analytics_websocket(websocket: WebSocket):
    """WebSocket for real-time analytics updates"""
    await websocket.accept()
    try:
        while True:
            data = analytics.get_dashboard_data()
            await websocket.send_json(data)
            await asyncio.sleep(2)  # Update every 2 seconds
    except WebSocketDisconnect:
        pass

@app.get("/content-gaps")
async def get_content_gaps():
    """Get identified content gaps"""
```

```
    return {"gaps": list(analytics.content_gaps.values())}

@app.post("/content-gaps/{gap_id}/create")
async def create_content_for_gap(gap_id: str):
    """Create content for identified gap"""
    if gap_id in analytics.content_gaps:
        gap = analytics.content_gaps[gap_id]
        draft = await content_manager.create_draft_for_gap(gap.query)
        return {"success": True, "draft": draft}
    return {"success": False, "error": "Gap not found"}

@app.get("/sessions/{session_id}/history")
async def get_chat_history(session_id: str):
    """Get chat history for a session"""
    if session_id in chat_sessions:
        return {"history": chat_sessions[session_id]}
    return {"history": []}

@app.get("/health")
async def health_check():
    """Health check endpoint"""
    return {
        "status": "healthy",
        "active_sessions": len(chat_sessions),
        "total_queries": len(analytics.events)
    }

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

.env file to create:

```
CONTENTSTACK_API_KEY=your_contentstack_api_key
CONTENTSTACK_DELIVERY_TOKEN=your_delivery_token
CONTENTSTACK_ENVIRONMENT=development
GROQ_API_KEY=your_groq_api_key
```

Copy all these files, install dependencies, and run `python main.py` - it should work immediately!