AI Chat RAG Agent - Teknisk Dokumentation

Version: 2.3.0
Dato: Januar 2026
System: RAG-baseret AI Chat med Sikker Authentication

Del 1: Overordnet Beskrivelse

Hvad er denne applikation?

AI Chat RAG Agent er en intelligent chatløsning designet til gymnasier og uddannelsesinstitutioner. Applikationen kombinerer kunstig intelligens (AI) med Retrieval Augmented Generation (RAG) teknologi, hvilket gør det muligt for brugere at føre samtaler med en AI-assistent, der kan trække relevant information fra uploadede dokumenter.

For slutbrugeren betyder dette:

- **Intelligent chat**: Du kan stille spørgsmål og få svar baseret på både AI'ens generelle viden og specifikke dokumenter, der er uploadet til systemet
- **Sikker login**: Log ind med din organisations Microsoft-konto (Azure AD) eller et lokalt brugernavn/password
- **Dokumentbaserede svar**: AI'en kan referere til og citere fra uploadede materialer som lærebøger, noter eller andre dokumenter
- **Samtalehistorik**: Dine samtaler gemmes i sessionen, så du kan fortsætte hvor du slap

For teknikeren betyder dette:

- **Full-stack løsning**: Python/FastAPI backend med HTML/CSS/JavaScript frontend
- **RAG-arkitektur**: ChromaDB vektordatabase med OpenAI embeddings
- **Container-baseret**: Docker Compose deployment med NGINX reverse proxy
- **Sikkerhedslagdelt**: JWT authentication, HttpOnly cookies, CSRF protection, rate limiting

---

Del 2: Frontend Dokumentation

2.1 Teknisk Overblik

Frontend er bygget som en Single Page Application (SPA) med ren JavaScript, HTML5 og CSS3.

Teknologier og biblioteker:

Komponent	Formål
Marked.js	Markdown rendering af AI-svar
DOMPurify	XSS-beskyttelse ved HTML-sanitering
Highlight.js	Syntaks-fremhævning af kodeblokke
jsPDF + html2canvas	PDF-eksport af samtaler

Filstruktur:

```
frontend/
├── index.html          # Hovedside med chat interface
├── static/
│   ├── css/
│   │   └── styles.css  # Styling og layout
│   └── js/
│       └── app.js      # Applikationslogik
```

2.2 Brugerinterface

Login-modal

Brugeren mødes af en login-dialog med to muligheder: 1. **Microsoft Login** (Azure AD SSO) - Single Sign-On via organisationens Microsoft-konto 2. **Lokal login** - Brugernavn og password

Sidebar (venstre panel)

- **Konfiguration**: API endpoint indstillinger
- **Prompts**: Valg af system prompt (kontekst for AI'en)
- **Status**: Session- og forbindelsesstatus, token-forbrug og omkostninger
- **Handlinger**: Nulstil session, ryd chat, gem som PDF

Chat-område (højre panel)

- **Beskedhistorik**: Viser alle beskeder med tydelig adskillelse mellem bruger og AI
- **Input-felt**: Tekstområde med Enter-til-send og Shift+Enter til linjeskift
- **Send-knap**: Sender beskeden til API'et

2.3 Sikkerhed i Frontend

XSS-beskyttelse

Al output fra AI'en saniteres med DOMPurify før visning:
DOMPurify er et sikkerhedsbibliotek der fjerner farlig kode fra HTML
uden

```
const sanitizedHtml = DOMPurify.sanitize(marked.parse(response));
```

CSRF Token Håndtering

Frontend læser CSRF-token fra cookie og sender den i header ved alle state-ændrende requests:

```
const csrfToken = getCookie('csrf_token');
headers['X-CSRF-Token'] = csrfToken;
```

Token-lagring (bemærk sikkerhedsopdatering)

Authentication tokens gemmes i HttpOnly cookies, som JavaScript IKKE kan tilgå. Dette beskytter mod token-tyveri via XSS-angreb.

Del 3: Backend Dokumentation

3.1 Teknisk Overblik

Backend er bygget med Python og FastAPI frameworket.

Hovedkomponenter:

Komponent	Teknologi	Formål
Web Framework	FastAPI	Async API med automatisk dokumentation
AI/LLM	OpenAI GPT-4o-mini	Tekstgenerering og samtale
Vektordatabase	ChromaDB	Semantisk søgning i dokumenter
Embeddings	OpenAI Embeddings	Konvertering af tekst til vektorer
Authentication	JWT + Azure AD	Bruger-authentication
Rate Limiting	SlowAPI	Beskyttelse mod misbrug

Filstruktur:

```
backend/
├── main.py                    # FastAPI applikation
├── config/
│   ├── __init__.py
│   └── settings.py            # Konfiguration og miljøvariabler
├── services/
│   ├── __init__.py
│   ├── auth_service.py        # JWT og Azure AD authentication
│   ├── audit_service.py       # Request logging
│   ├── chat_rag_service.py    # Chat og RAG logik
│   └── csrf_service.py        # CSRF beskyttelse
└── routers/
    ├── __init__.py
    ├── auth_router.py         # Authentication endpoints
    └── chat_router.py         # Chat endpoints
```

3.2 API Endpoints

Authentication (`/api/auth/`)

Endpoint	Metode	Beskrivelse
`/login`	POST	Login med brugernavn/password
`/logout`	POST	Log ud og slet cookies
`/refresh`	POST	Forny access token
`/status`	GET	Check om JWT er aktiveret
`/me`	GET	Hent aktuel brugers info
`/azure/login`	GET	Start Azure AD login flow
`/azure/callback`	GET	Håndter Azure AD callback

Chat (`/api/`)

Endpoint	Metode	Beskrivelse
`/prompts`	GET	Hent tilgængelige system prompts
`/chat`	POST	Send besked og få AI-svar
`/history`	GET	Hent samtalehistorik
`/reset`	POST	Nulstil samtale
`/collections`	GET	List tilgængelige dokument-collections
`/health`	GET	Health check endpoint

3.3 RAG (Retrieval Augmented Generation)

RAG-processen fungerer således:

1. **Bruger sender spørgsmål** → API modtager beskeden
2. **Semantisk søgning** → Spørgsmålet konverteres til en vektor og søges i ChromaDB
3. **Kontekst-hentning** → De mest relevante dokumentchunks (TOP_K=3) hentes
4. **Prompt-konstruktion** → System prompt + kontekst + brugerbesked samles
5. **LLM-kald** → OpenAI API genererer svar baseret på al information
6. **Svar med kilder** → Bruger modtager svar inkl. metadata om anvendte kilder

Eksempel på RAG-flow:

```
# 1. Hent relevant kontekst fra dokumenter
context, sources_count, metadata = service.get_rag_context(
    user_query="Hvad er fotosyntese?",
    collection_name="biologi_noter"
)

# 2. Byg messages med kontekst
messages = [
```

```
    {"role": "system", "content": system_prompt},
    {"role": "system", "content": f"Kontekst fra dokumenter:\n{context}"},
    {"role": "user", "content": user_query}
]

# 3. Send til OpenAI
response = await openai.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages
)
```

3.4 Session Management

Sessioner håndteres in-memory med automatisk cleanup:

- **Max sessions**: 1000 samtidige sessioner
- **Timeout**: 2 timer inaktivitet
- **Cleanup**: Automatisk ved nye sessioner eller periodisk
- **Session data**: Beskedhistorik, valgt collection, aktivitetstidspunkt

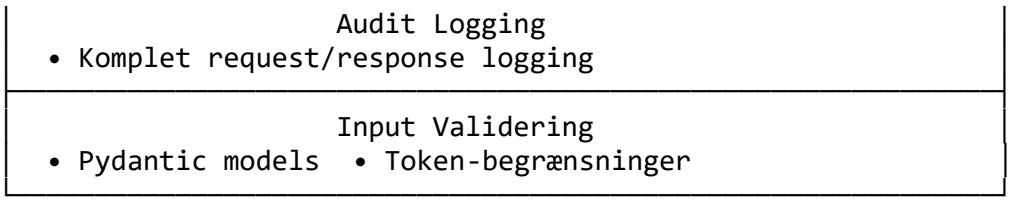
---

Del 4: Sikkerhedselementer

4.1 Oversigt over sikkerhedslag

Applikationen implementerer et "defense in depth" princip med multiple sikkerhedslag:

```
┌─────────────────────────────────────────────────────────────┐
│                    NGINX Reverse Proxy                      │
│  • SSL/TLS terminering  • Rate limiting  • IP filtering     │
├─────────────────────────────────────────────────────────────┤
│                    Internal Secret Header                   │
│  • Verificerer requests kommer fra NGINX                    │
├─────────────────────────────────────────────────────────────┤
│                    JWT Authentication                       │
│  • Access tokens (60 min)  • Refresh tokens (7 dage)        │
├─────────────────────────────────────────────────────────────┤
│                    HttpOnly Cookies                         │
│  • Tokens utilgængelige for JavaScript                      │
├─────────────────────────────────────────────────────────────┤
│                    CSRF Protection                          │
│  • Double Submit Cookie pattern                             │
├─────────────────────────────────────────────────────────────┤
│                    Rate Limiting                            │
│  • Per-endpoint begrænsninger                               │
└─────────────────────────────────────────────────────────────┘
```



4.2 HttpOnly Cookies (XSS-beskyttelse)

Problem som løses:

Cross-Site Scripting (XSS) angreb kan stjæle authentication tokens hvis de er tilgængelige for JavaScript (f.eks. i localStorage).

Løsning implementeret:

JWT tokens gemmes i HttpOnly cookies, som browseren automatisk sender med requests, men som JavaScript IKKE kan læse.

Konfiguration:

```
# Cookie Settings
COOKIE_DOMAIN = settings.cookie_domain      # Domæne-begrænsning
COOKIE_SECURE = True                        # Kun HTTPS
COOKIE_SAMESITE = "lax"                     # Beskytter mod CSRF

# Ved login sættes cookie:
response.set_cookie(
    key="access_token",
    value=token,
    httponly=True,      # KRITISK: JavaScript kan ikke læse
    secure=True,        # Kun over HTTPS
    samesite="lax",     # CSRF-beskyttelse
    max_age=3600        # 1 time
)
```

Effekt:

Selv hvis en angriber injicerer ondsindet JavaScript, kan de IKKE stjæle brugerens token.

4.3 CSRF Protection (Cross-Site Request Forgery)

Problem som løses:

En angriber kan narre brugerens browser til at sende requests til vores API fra et ondsindet website.

Løsning: Double Submit Cookie Pattern

1. Ved login genereres en CSRF token og gemmes i en cookie (IKKE HttpOnly - JS skal kunne læse den)
2. Frontend læser token fra cookie og sender den i `X-CSRF-Token` header
3. Backend verificerer at header-værdi matcher cookie-værdi

Hvorfor dette virker:

- Angriberens website kan få browseren til at SENDE cookies automatisk
- Men angriberens website kan IKKE LÆSE cookies fra vores domæne (Same-Origin Policy)
- Derfor kan angriberen ikke sætte den korrekte header

Implementation:

```
class CSRFProtection:
    def generate_token(self) -> str:
        random_part = secrets.token_urlsafe(32)
        timestamp = str(int(datetime.utcnow().timestamp()))
        signature = self._sign(f"{random_part}.{timestamp}")
        return f"{random_part}.{timestamp}.{signature}"

    def verify_token(self, token: str) -> Tuple[bool, str]:
        # Verificer signatur og timestamp
        ...
```

4.4 JWT Authentication

Token-typer:

Type	Levetid	Formål
Access Token	60 minutter	API-adgang
Refresh Token	7 dage	Forny access token

Sikkerhedselementer:

- **HS256 signering**: Tokens signeres med hemmeligt nøgle (min. 32 tegn)
- **Expiration**: Automatisk udløb forhindrer evige sessioner
- **User binding**: Tokens indeholder bruger-ID, roller og auth-provider

Azure AD Integration:

Understøtter Single Sign-On via Microsoft Azure AD/Entra ID: - OAuth 2.0 Authorization Code flow - Automatisk brugeroprettelse ved første login - Roller kan mappes fra Azure AD grupper

4.5 Rate Limiting

Beskytter mod misbrug og DDoS:

Endpoint	Grænse	Formål

Endpoint	Grænse	Formål
`/auth/*`	5/minut	Forhindrer brute force login
`/chat`	10/minut	Begrænser API-forbrug
`/prompts`	20/minut	Normal brug
`/health`	60/minut	Monitoring tilladt
`/` (root)	30/minut	Generel beskyttelse

Implementation:

```
from slowapi import Limiter
limiter = Limiter(key_func=get_remote_address)

@router.post("/chat")
@limiter.limit("10/minute")
async def chat(request: Request, ...):
    ...
```

4.6 Internal Secret Verification

Problem som løses:

Direkte adgang til FastAPI containeren udenom NGINX.

Løsning:

NGINX tilføjer en hemmelig header til alle requests:

```
proxy_set_header X-Internal-Secret "din-hemmelige-nøgle";
```

Backend verificerer denne header:

```
async def verify_internal_secret(request: Request, call_next):
    secret = request.headers.get("x-internal-secret")
    if secret != API_SECRET_KEY:
        raise HTTPException(status_code=403, detail="Adgang nægtet")
    return await call_next(request)
```

4.7 Audit Logging

Komplet logging af alle API-requests til fil:

Hvad logges:

- Timestamp
- Bruger ID (fra JWT eller 'anonymous')
- IP-adresse
- HTTP metode og endpoint
- Request body (saniteret - passwords maskeres)

- Response status og tid
- Token-forbrug (for chat requests)

Konfiguration:

```
AUDIT_LOGGING_ENABLED = True
AUDIT_LOG_FILE = "/app/logs/audit.log"
AUDIT_LOG_MAX_SIZE_MB = 100        # Rotating logs
AUDIT_LOG_BACKUP_COUNT = 5          # Gem 5 gamle filer
```

Sensitiv data beskyttelse:

```
def _sanitize_body(self, body: Dict) -> Dict:
    sensitive_fields = [
        'password', 'token', 'secret', 'api_key',
        'authorization', 'credential', 'private_key'
    ]
    for field in sensitive_fields:
        if field in body:
            body[field] = '[REDACTED]'
    return body
```

4.8 Input Validering

Token-begrænsninger:

- **Max input tokens**: 2500 tokens (~10.000 tegn)
- **Max output tokens**: 3000 tokens

Pydantic Validering:

```
class ChatRequest(BaseModel):
    session_id: str = Field(..., min_length=1, max_length=100)
    message: str = Field(..., min_length=1, max_length=50000)
    max_output_tokens: int = Field(default=300, ge=50, le=3000)

    @field_validator('message')
    def validate_message_length(cls, v):
        estimated_tokens = len(v) // 4
        if estimated_tokens > MAX_INPUT_TOKENS:
            raise ValueError(f"Besked for lang: {estimated_tokens} tokens")
        return v
```

4.9 CORS Konfiguration

Strict Cross-Origin Resource Sharing:

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://chat-assistant.itcfyn.ai"],  # Kun specifik origin
    allow_credentials=True,   # Tillader cookies
    allow_methods=["GET", "POST", "OPTIONS"],
```

```
    allow_headers=["Authorization", "X-CSRF-Token", "Content-Type"],
    expose_headers=["X-CSRF-Token"]
)
```

Del 5: Deployment

5.1 Docker Compose Arkitektur

```yaml
services:
  nginx-proxy:
    image: jc21/nginx-proxy-manager:latest
    ports:
      - "80:80"    # HTTP
      - "443:443"  # HTTPS
      - "81:81"    # Admin UI

  backend:
    build: ./backend
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - JWT_SECRET_KEY=${JWT_SECRET_KEY}
    volumes:
      - ./chroma_collections:/app/chroma_collections
      - ./prompts:/app/prompts
```

5.2 Miljøvariabler

Variabel	Beskrivelse	Påkrævet
`OPENAI_API_KEY`	OpenAI API nøgle	Ja
`API_SECRET_KEY`	Intern NGINX-FastAPI secret	Ja
`JWT_SECRET_KEY`	JWT signerings-nøgle (min. 32 tegn)	Hvis JWT aktiveret
`JWT_ENABLED`	Aktiver JWT authentication	Nej (default: false)
`AZURE_AD_ENABLED`	Aktiver Azure AD SSO	Nej (default: false)
`AUDIT_LOGGING_ENABLED`	Aktiver audit logging	Nej (default: false)

Del 6: Anbefalinger

6.1 Før produktion

1. **Skift alle standard-passwords og secrets**
2. **Aktiver HTTPS** via NGINX Proxy Manager

3. **Sæt `COOKIE_SECURE=true`** (kun HTTPS)
4. **Konfigurer Azure AD** for enterprise SSO
5. **Aktiver audit logging** for compliance
6. **Begræns `CORS_ORIGINS`** til faktiske domæner

6.2 Monitoring

- Overvåg `/health` endpoint
- Gennemgå audit logs regelmæssigt
- Sæt alerts på rate limit overskridelser
- Monitor token-forbrug og omkostninger

Dokument slut

Genereret: Januar 2026