Search Service - Integration Guide

Overview

This guide explains how to integrate the Search Service with other Mission Engadi services and external applications.

Table of Contents

Service-to-Service Integration

Content Service Integration

The Search Service automatically indexes content from the Content Service.

Setup:

1. Configure Content Service URL in `.env`:

```
CONTENT_SERVICE_URL=http://localhost:8007
```

1. Trigger initial indexing:

```
curl -X POST http://localhost:8011/api/v1/indexing/reindex/content \
  -H "Authorization: Bearer YOUR_TOKEN"
```

Webhook for Auto-Indexing:

Configure Content Service to send webhooks on content changes:

```python
# In Content Service
import requests

def on_content_created(content):
    requests.post(
        "http://localhost:8011/api/v1/indexing/index",
        headers={"Authorization": f"Bearer {SERVICE_TOKEN}"},
        json={
            "document_id": content.id,
            "document_type": "article",
            "title": content.title,
            "content": content.body,
            "language": content.language,
            "author": content.author,
            "status": content.status
        }
    )
```

Partners Service Integration

Configuration:

```
PARTNERS_SERVICE_URL=http://localhost:8009
```

Indexing Partners:

```
curl -X POST http://localhost:8011/api/v1/indexing/reindex/partners \
  -H "Authorization: Bearer YOUR_TOKEN"
```

Projects Service Integration

Configuration:

```
PROJECTS_SERVICE_URL=http://localhost:8010
```

Indexing Projects:

```
curl -X POST http://localhost:8011/api/v1/indexing/reindex/projects \
  -H "Authorization: Bearer YOUR_TOKEN"
```

Social Media Service Integration

Configuration:

```
SOCIAL_MEDIA_SERVICE_URL=http://localhost:8012
```

Authentication

JWT Authentication

The Search Service uses JWT tokens for authentication.

Required Headers:

```
Authorization: Bearer <JWT_TOKEN>
X-User-ID: <USER_ID>
```

Example:

```python
import requests

headers = {
    "Authorization": "Bearer eyJhbGciOiJIUzI1NiIs...",
    "X-User-ID": "123"
}

response = requests.post(
    "http://localhost:8011/api/v1/indexing/index",
    headers=headers,
    json={...}
)
```

Service-to-Service Authentication

For service-to-service calls, use service tokens:

```
SERVICE_TOKEN=your_service_token_here
```

Webhook Integration

Automatic Indexing via Webhooks

Configure your services to send webhooks to Search Service on content changes.

Webhook Endpoint: `POST /api/v1/indexing/index`

Events to Track:
- Content Created
- Content Updated
- Content Deleted
- Content Published

Example Webhook Implementation

In Your Service:

```python
from typing import Dict
import requests

SEARCH_SERVICE_URL = "http://localhost:8011"
SERVICE_TOKEN = "your_service_token"

class SearchWebhook:
    @staticmethod
    def index_document(doc_data: Dict):
        """Index a document in search service."""
        try:
            response = requests.post(
                f"{SEARCH_SERVICE_URL}/api/v1/indexing/index",
                headers={
                    "Authorization": f"Bearer {SERVICE_TOKEN}",
                    "Content-Type": "application/json"
                },
                json=doc_data,
                timeout=5
            )
            response.raise_for_status()
            print(f"Document indexed: {doc_data['document_id']}")
        except Exception as e:
            print(f"Failed to index document: {e}")

    @staticmethod
    def update_document(doc_id: int, updates: Dict):
        """Update an indexed document."""
        try:
            response = requests.put(
                f"{SEARCH_SERVICE_URL}/api/v1/indexing/update/{doc_id}",
                headers={
                    "Authorization": f"Bearer {SERVICE_TOKEN}",
                    "Content-Type": "application/json"
                },
                json=updates,
                timeout=5
            )
            response.raise_for_status()
            print(f"Document updated: {doc_id}")
        except Exception as e:
            print(f"Failed to update document: {e}")

    @staticmethod
    def delete_document(doc_id: int):
        """Remove document from search index."""
        try:
            response = requests.delete(
                f"{SEARCH_SERVICE_URL}/api/v1/indexing/delete/{doc_id}",
                headers={"Authorization": f"Bearer {SERVICE_TOKEN}"},
                timeout=5
            )
            response.raise_for_status()
            print(f"Document deleted: {doc_id}")
        except Exception as e:
            print(f"Failed to delete document: {e}")

# Usage in your models/services
class Article:
    def save(self):
        # Save to database
        super().save()
```

```
        # Index in search service
        SearchWebhook.index_document({
            "document_id": self.id,
            "document_type": "article",
            "title": self.title,
            "content": self.content,
            "language": self.language,
            "author": self.author,
            "status": self.status
        })

    def delete(self):
        # Delete from search index
        SearchWebhook.delete_document(self.id)

        # Delete from database
        super().delete()
```

Client Libraries

Python Client

```python
from typing import Dict, List, Optional
import requests

class SearchClient:
    def __init__(self, base_url: str = "http://localhost:8011", token: str = None):
        self.base_url = base_url
        self.token = token

    def _headers(self) -> Dict:
        headers = {"Content-Type": "application/json"}
        if self.token:
            headers["Authorization"] = f"Bearer {self.token}"
        return headers

    def search(self, query: str, filters: Dict = None, page: int = 1, page_size: int =
10) -> Dict:
        """Universal search."""
        response = requests.post(
            f"{self.base_url}/api/v1/search",
            headers=self._headers(),
            json={
                "query": query,
                "filters": filters or {},
                "page": page,
                "page_size": page_size
            }
        )
        response.raise_for_status()
        return response.json()

    def search_articles(self, query: str, page: int = 1) -> Dict:
        """Search articles."""
        response = requests.post(
            f"{self.base_url}/api/v1/search/articles",
            headers=self._headers(),
            json={"query": query, "page": page, "page_size": 10}
        )
        response.raise_for_status()
        return response.json()

    def get_suggestions(self, query: str, limit: int = 10) -> List[Dict]:
        """Get autocomplete suggestions."""
        response = requests.get(
            f"{self.base_url}/api/v1/autocomplete/suggestions",
            headers=self._headers(),
            params={"query": query, "limit": limit}
        )
        response.raise_for_status()
        return response.json()

    def get_facets(self, query: str = None) -> Dict:
        """Get search facets."""
        params = {"query": query} if query else {}
        response = requests.get(
            f"{self.base_url}/api/v1/facets",
            headers=self._headers(),
            params=params
        )
        response.raise_for_status()
        return response.json()

    def index_document(self, document: Dict) -> Dict:
```

```
        """Index a document."""
        response = requests.post(
            f"{self.base_url}/api/v1/indexing/index",
            headers=self._headers(),
            json=document
        )
        response.raise_for_status()
        return response.json()

# Usage
client = SearchClient(token="your_token")
results = client.search("missions", filters={"language": "en"})
suggestions = client.get_suggestions("miss")
```

JavaScript/TypeScript Client

```typescript
interface SearchRequest {
  query: string;
  filters?: Record<string, any>;
  page?: number;
  page_size?: number;
}

interface SearchResponse {
  results: any[];
  total: number;
  page: number;
  page_size: number;
  total_pages: number;
}

class SearchClient {
  private baseUrl: string;
  private token?: string;

  constructor(baseUrl: string = 'http://localhost:8011', token?: string) {
    this.baseUrl = baseUrl;
    this.token = token;
  }

  private headers(): Record<string, string> {
    const headers: Record<string, string> = {
      'Content-Type': 'application/json',
    };
    if (this.token) {
      headers['Authorization'] = `Bearer ${this.token}`;
    }
    return headers;
  }

  async search(request: SearchRequest): Promise<SearchResponse> {
    const response = await fetch(`${this.baseUrl}/api/v1/search`, {
      method: 'POST',
      headers: this.headers(),
      body: JSON.stringify(request),
    });
    if (!response.ok) throw new Error('Search failed');
    return response.json();
  }

  async getSuggestions(query: string, limit: number = 10): Promise<any[]> {
    const response = await fetch(
      `${this.baseUrl}/api/v1/autocomplete/suggestions?query=${query}&limit=${limit}`,
      { headers: this.headers() }
    );
    if (!response.ok) throw new Error('Failed to get suggestions');
    return response.json();
  }
}

// Usage
const client = new SearchClient('http://localhost:8011', 'your_token');
const results = await client.search({ query: 'missions', page: 1 });
```

Best Practices

1. Indexing Strategy

Use Bulk Indexing:

```
# Instead of multiple single requests
for doc in documents:
    client.index_document(doc)  # BAD

# Use bulk indexing
client.bulk_index(documents)  # GOOD
```

Schedule Reindexing:

```
# Cron job for daily reindexing
0 2 * * * /path/to/reindex_all.sh
```

2. Search Query Optimization

Use Specific Document Types:

```
# More efficient
client.search_articles("query")

# Less efficient
client.search("query", filters={"document_type": "article"})
```

Implement Caching:

```
from functools import lru_cache

@lru_cache(maxsize=100)
def cached_search(query: str, filters_json: str):
    filters = json.loads(filters_json)
    return client.search(query, filters)
```

3. Error Handling

```
from requests.exceptions import RequestException, Timeout

try:
    results = client.search("query")
except Timeout:
    # Handle timeout
    results = get_cached_results("query")
except RequestException as e:
    # Handle other errors
    log_error(e)
    results = {"error": "Search unavailable"}
```

4. Monitoring

Track Search Performance:

```
import time

start = time.time()
results = client.search("query")
latency = time.time() - start

log_metric("search_latency", latency)
```

5. Rate Limiting

Implement backoff:

```
import time
from requests.exceptions import HTTPError

def search_with_retry(query, max_retries=3):
    for i in range(max_retries):
        try:
            return client.search(query)
        except HTTPError as e:
            if e.response.status_code == 429:  # Rate limit
                time.sleep(2 ** i)  # Exponential backoff
            else:
                raise
    raise Exception("Max retries exceeded")
```

Support

For integration support:
- API Documentation: API_DOCUMENTATION.md (./API_DOCUMENTATION.md)
- Deployment Guide: DEPLOYMENT_GUIDE.md (./DEPLOYMENT_GUIDE.md)