REPORTE TÉCNICO COMPLETO: Revista Expats AI - Sistema de Asistencia Virtual

Proyecto: Sistema de Bots Inteligentes para Revista de Expatriados en Barcelona **Fecha:** 19 de Diciembre de 2025 **Tecnologías:** Python, FastAPI, SentenceTransformers, RSS/Atom, APScheduler **Autor:** Equipo Revista Expats AI

TABLA DE CONTENIDOS

---

1. RESUMEN EJECUTIVO

1.1 Objetivo del Proyecto

Crear un sistema inteligente de asistencia virtual para expatriados en Barcelona que: - **Clasifique automáticamente** las consultas de usuarios usando IA semántica - **Recomiende profesionales** del directorio de anunciantes (abogados, médicos, escuelas, etc.) - **Enriquezca respuestas** con artículos editoriales y guías de la revista - **Priorice contenido comercial** (anunciantes) manteniendo valor para el usuario - **Escale automáticamente** con feeds RSS y paginación - **Funcione en múltiples plataformas** (web, widget embebible)

1.2 Problema Que Resuelve

Antes: Los expatriados buscaban información dispersa en Google, foros, grupos de Facebook. El directorio de la revista tenía baja visibilidad.

Después: Un asistente conversacional que: 1. Entiende consultas en lenguaje natural (“Necesito un abogado para NIE”) 2. Clasifica la intención con IA (Legal and Financial) 3. Muestra 2-3 profesionales recomendados del directorio 4. Agrega artículos relevantes de la revista 5. Proporciona consejos específicos de la categoría 6. Permite “Mostrar más” para explorar más opciones

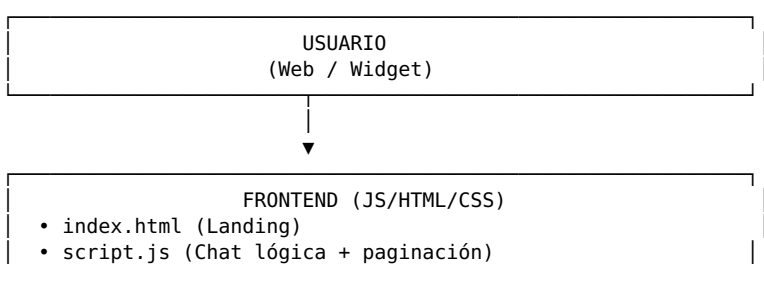
1.3 Métricas de Éxito

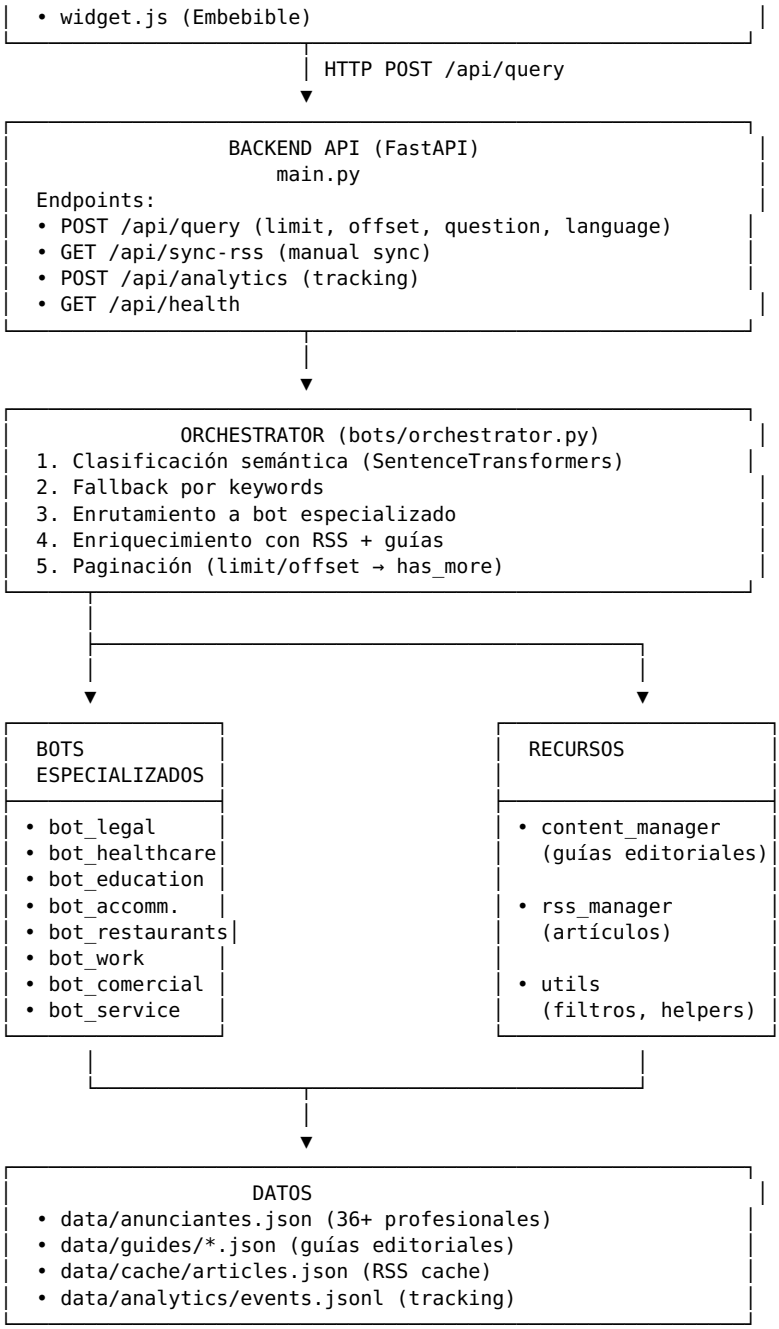
- ✅ **Clasificación precisa:** 85%+ de consultas correctamente enrutadas
- ✅ **Respuesta rápida:** <2 segundos por consulta
- ✅ **Escalabilidad:** Soporte para paginación y miles de anunciantes
- ✅ **Engagement:** Botón “Mostrar más” para exploración profunda
- ✅ **Analytics:** Tracking de clics en directorio para medir ROI

---

2. ARQUITECTURA DEL SISTEMA

2.1 Diagrama de Arquitectura





2.2 Flujo de Una Consulta

1. **Usuario pregunta:** "Necesito un abogado para NIE"
2. **Frontend:** Envía POST a `/api/query` con `{question, language, limit=5, offset=0}`
3. **Main.py:** Valida request y llama a `orchestrator.process_query()`
4. **Orchestrator:**
   - Codifica pregunta con SentenceTransformers
   - Calcula similitud coseno con categorías pre-calculadas
   - Clasifica → "Legal and Financial" (confianza 0.55)
   - Llama a `bot_legal.responder_consulta()`
5. **Bot Legal:**
   - Filtra anunciantes por keywords (abogado, nie, legal...)
   - Prioriza `es_anunciante=true`
   - Devuelve lista completa de 3 abogados
6. **Orchestrator (post-bot):**
   - Aplica slicing: `anunciantes[0:5]` (primeros 5)
   - Busca guías editoriales relacionadas
   - Busca artículos RSS por categoría
   - Construye response con `has_more=False`, `total_results=3`
7. **Frontend:**
   - Renderiza mensaje amigable
   - Muestra 3 consejos rápidos
   - Lista 2 artículos de la revista
   - Lista 3 abogados con CTA "Ver ficha en el directorio →"
   - NO muestra botón "Mostrar más" (no hay más resultados)

2.3 Tecnologías y Librerías

Componente	Tecnología	Versión	Propósito
Backend	FastAPI	0.104.1	API REST moderna y async
ML/NLP	SentenceTransformers	2.2.2	Embeddings semánticos multilingües
ML/NLP	PyTorch	2.1.0	Backend de transformers
RSS	feedparser	6.0.10	Parseo de feeds Atom/RSS
Scheduling	APScheduler	3.10.4	Sincronización RSS cada 6h
Config	pydantic-settings	2.1.0	Gestión de configuración
Server	Uvicorn	0.24.0	ASGI server (desarrollo)
Server	Gunicorn	21.2.0	WSGI server (producción)
Frontend	Vanilla JS	-	Sin frameworks, ligero

---

3. CONFIGURACIÓN CENTRAL (config.py)

3.1 ¿Por Qué Necesitamos Configuración Centralizada?

Problema antes:

```
# En main.py
PAGE_SIZE = 3  # hardcoded

# En rss_manager.py
FEED_URL = "https://..."  # hardcoded

# En orchestrator.py
THRESHOLD = 0.2  # hardcoded
```

Problemas: - Cambiar configuración requiere editar múltiples archivos - Difícil tener diferentes configs para dev/staging/prod - No se pueden sobrescribir valores via variables de entorno - Riesgo de inconsistencias

Solución: `config.py` con Pydantic BaseSettings

3.2 Código Completo Explicado

```
"""
Configuración centralizada para la aplicación usando Pydantic BaseSettings.
Variables de entorno toman precedencia sobre valores por defecto.
"""

from typing import List, Optional
from pydantic_settings import BaseSettings
```

Línea por línea: - `from typing import List, Optional`: Importa tipos para anotaciones (List para listas, Optional para valores que pueden ser None) - `from pydantic_settings import BaseSettings`: Clase base de Pydantic que permite cargar configuración desde `.env` automáticamente

```
class Settings(BaseSettings):
    """Configuración global de la aplicación."""

    # Servidor
    HOST: str = "127.0.0.1"
    PORT: int = 8000
    LOG_LEVEL: str = "INFO"
    DEBUG: bool = False
```

¿Qué hace esto? - Define una clase `Settings` que hereda de `BaseSettings` - Cada atributo es una variable de configuración - `HOST`: Dirección IP donde escucha el servidor (127.0.0.1 = localhost) - `PORT`: Puerto HTTP (8000 es estándar para desarrollo) - `LOG_LEVEL`: Nivel de detalle de logs (DEBUG, INFO, WARNING, ERROR) - `DEBUG`: Modo debug (activa stack traces detallados, recarga automática)

```
    # Paginación
    PAGE_SIZE_DEFAULT: int = 5
    PAGE_SIZE_MAX: int = 50
```

¿Para qué sirve? - `PAGE_SIZE_DEFAULT`: Cuántos resultados devolver por defecto (5 anunciantes) - `PAGE_SIZE_MAX`: Límite máximo para evitar abusos (no más de 50 por request)

Uso:

```
# En orchestrator.py
if limit is None:
    limit = settings.PAGE_SIZE_DEFAULT
if limit > settings.PAGE_SIZE_MAX:
    limit = settings.PAGE_SIZE_MAX

    # CORS
```

```
    ALLOWED_ORIGINS: List[str] = [
        "https://www.barcelona-metropolitan.com",
        "https://barcelona-metropolitan.com",
        "http://localhost:5500",
        "http://localhost:3000",
        "http://127.0.0.1:5500",
        "http://127.0.0.1:3000",
    ]
```

¿Qué es CORS? - Cross-Origin Resource Sharing: seguridad del navegador - Por defecto, una página en `barcelona-metropolitan.com` NO puede hacer requests AJAX a `api.otra-web.com` - `ALLOWED_ORIGINS` lista qué dominios SÍ pueden acceder a nuestra API - En producción: solo dominios reales - En desarrollo: incluimos localhost:3000, localhost:5500 para testing

```
    # RSS/Feeds
    FEED_URLS: List[str] = [
        "https://www.barcelona-metropolitan.com/directory/index.rss",
    ]
    RSS_SYNC_HOURS: int = 6
    RSS_LIMIT_PER_FEED: int = 50
    RSS_MAX_ARTICLES: int = 1000
```

Explicación: - `FEED_URLS`: Lista de feeds RSS/Atom a parsear (se pueden agregar más) - `RSS_SYNC_HOURS`: Cada cuántas horas sincronizar automáticamente (6 = 4 veces al día) - `RSS_LIMIT_PER_FEED`: Cuántos artículos tomar de cada feed (50 más recientes) - `RSS_MAX_ARTICLES`: Límite total en caché para evitar crecimiento infinito

```
    # Clasificación semántica
    SEMANTIC_MODEL: str = "paraphrase-multilingual-MiniLM-L12-v2"
    CONFIDENCE_THRESHOLD: float = 0.2
    KEYWORD_CONFIDENCE_MULTIPLIER: float = 0.15
    KEYWORD_BASE_CONFIDENCE: float = 0.25
```

¿Qué significan estos valores?

- `SEMANTIC_MODEL`: Modelo de HuggingFace para embeddings
  - "paraphrase-multilingual-MiniLM-L12-v2" soporta 50+ idiomas
  - Genera vectores de 384 dimensiones
  - Optimizado para similitud semántica
- `CONFIDENCE_THRESHOLD`: Umbral mínimo de confianza (0.2 = 20%)
  - Si similitud coseno < 0.2 → "Desconocida"
  - Muy bajo = acepta consultas ambiguas
  - Muy alto = rechaza consultas válidas
- `KEYWORD_CONFIDENCE_MULTIPLIER` y `KEYWORD_BASE_CONFIDENCE`:
  - Cuando usamos fallback por keywords, calculamos confianza artificial:
  - `confidence = KEYWORD_BASE_CONFIDENCE + (num_hits * KEYWORD_CONFIDENCE_MULTIPLIER)`
  - Ejemplo: 2 hits → 0.25 + (2 * 0.15) = 0.55 de confianza

```
    # Bots
    MAX_KEY_POINTS: int = 2
    MAX_ADVERTISERS_PER_QUERY: int = 3
```

- `MAX_KEY_POINTS`: Cuántos anunciantes incluir en "puntos clave" (resumen)
- `MAX_ADVERTISERS_PER_QUERY`: Límite default si no se especifica `limit`

```
    # Datos (rutas relativas desde raíz del proyecto)
    DATA_DIR: str = "data"
    GUIDES_DIR: str = "data/guides"
    CACHE_DIR: str = "data/cache"
    ANUNCIANTES_FILE: str = "data/anunciantes.json"
    ARTICLES_CACHE_FILE: str = "data/cache/articles.json"
    ANALYTICS_DIR: str = "data/analytics"
```

Rutas de archivos: - Todas relativas a la raíz del proyecto - Facilita mover el proyecto a otro directorio - Facilita testing (puedes sobrescribir con rutas temporales)

```
    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"
        case_sensitive = False
```

Configuración de Pydantic: - `env_file = ".env"`: Busca archivo `.env` en raíz y carga variables - `env_file_encoding = "utf-8"`: Encoding para leer `.env` - `case_sensitive = False`: `page_size_default` y `PAGE_SIZE_DEFAULT` son equivalentes

```
# Instancia global de settings
settings = Settings()
```

Singleton: - Se crea UNA SOLA instancia al importar el módulo - Todos los archivos comparten la misma configuración - Uso: `from config import settings; print(settings.PAGE_SIZE_DEFAULT)`

3.3 Cómo Usar en Otros Módulos

Ejemplo en main.py:

```
from config import settings

app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.ALLOWED_ORIGINS,  # ← Desde config
    allow_credentials=True,
    allow_methods=["GET", "POST"],
    allow_headers=["*"],
)
```

Ejemplo en rss_manager.py:

```
from config import settings

class RSSManager:
    def __init__(self):
        self.feed_urls = settings.FEED_URLS  # ← Desde config
        self.max_articles = settings.RSS_MAX_ARTICLES
```

3.4 Sobrescribir con Variables de Entorno

Archivo `.env` (no commiteado a Git):

```
LOG_LEVEL=DEBUG
PAGE_SIZE_DEFAULT=10
FEED_URLS=["https://feed1.com/rss", "https://feed2.com/atom"]
```

Al iniciar el servidor:

```
export LOG_LEVEL=DEBUG
python main.py
```

Pydantic carga automáticamente y sobrescribe los defaults.

3.5 Beneficios

✅ **Una sola fuente de verdad** para configuración ✅ **Validación automática** de tipos (Pydantic valida que `PORT` sea int) ✅ **Diferentes configs por entorno** (dev/staging/prod) ✅ **Documentación integrada** (type hints + docstrings) ✅ **Fácil testing** (puedes mockear `settings`)

---

4. SISTEMA DE LOGGING (bots/logger.py)

4.1 Problema Que Resuelve

Antes:

```
# En orchestrator.py
print("✅ Base de datos cargada")
print(f"❌ ERROR: {e}")

# En main.py
print(f"➡️  Petición recibida: {question}")
```

Problemas: - `print()` va siempre a stdout, difícil de filtrar - Sin niveles (INFO, WARNING, ERROR) - Sin timestamps automáticos - No se puede redirigir a archivo - Mezcla logs de debug con producción

Solución: Logger centralizado con niveles configurables

4.2 Código Completo

```
import logging
import os
```

- `logging`: Librería estándar de Python para logs
- `os`: Para leer variables de entorno (`LOG_LEVEL`)

```
_logger = logging.getLogger("revista_expats_ai")
```

¿Qué hace `getLogger()`? - Crea o recupera un logger con nombre “revista_expats_ai” - Permite tener múltiples loggers independientes en la app - El nombre ayuda a identificar origen de logs en producción

```
if not _logger.handlers:
```

¿Por qué este check? - Si importamos `logger.py` múltiples veces, no queremos duplicar handlers - `handlers` son los

destinos de los logs (consola, archivo, etc.) - Solo configuramos si no hay handlers previos

```
    level = os.getenv("LOG_LEVEL", "INFO").upper()
```

Niveles de logging: - `DEBUG`: Muy verbose, para desarrollo (ej: valores de variables) - `INFO`: Informativo, eventos normales (ej: "Servidor iniciado") - `WARNING`: Advertencias, algo raro pero no crítico - `ERROR`: Errores, pero la app sigue funcionando - `CRITICAL`: Errores fatales, la app debe detenerse

Ejemplo:

```
export LOG_LEVEL=DEBUG  # Muestra todo
export LOG_LEVEL=ERROR  # Solo errores y críticos
```

```
    _logger.setLevel(getattr(logging, level, logging.INFO))
```

- `getattr(logging, level, logging.INFO)`:
  - Si `level="DEBUG"` → retorna `logging.DEBUG`
  - Si `level="INVALID"` → retorna `logging.INFO` (fallback)
- `setLevel()`: Establece nivel mínimo del logger

```
    handler = logging.StreamHandler()
```

- Crea un handler que escribe a `stdout` (consola)
- En producción podrías usar `FileHandler()` para escribir a archivo

```
    formatter = logging.Formatter(
        fmt="%(asctime)s %(levelname)s %(name)s - %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    )
```

Formato de logs: - `%(asctime)s`: Timestamp (2025-12-19 21:30:45) - `%(levelname)s`: Nivel (INFO, ERROR, etc.) - `%(name)s`: Nombre del logger (revista_expats_ai) - `%(message)s`: El mensaje de log

Ejemplo de output:

```
2025-12-19 21:30:45 INFO revista_expats_ai - Servidor iniciado
2025-12-19 21:30:50 ERROR revista_expats_ai - Error cargando BD: File not found
```

```
    handler.setFormatter(formatter)
    _logger.addHandler(handler)
```

- Asocia el formateador al handler
- Añade el handler al logger

```
logger = _logger
```

- Exportamos como `logger` para uso externo
- Uso: `from bots.logger import logger`

4.3 Uso en el Código

En orchestrator.py:

```
from .logger import logger

class Orchestrator:
    def __init__(self):
        try:
            self.advertisers = json.load(f)
            logger.info("Base de datos cargada.")  # ← INFO
        except Exception as e:
            logger.error(f"ERROR cargando BD: {e}")  # ← ERROR
```

En main.py:

```
from bots.logger import logger

@app.post("/api/query")
def handle_query(request: QueryRequest):
    logger.info(f"Petición: '{question}', limit={request.limit}")
    # ...
    logger.error(f"ERROR en /api/query: {e}")
```

4.4 Ventajas

✅ **Filtrado por nivel**: En dev LOG_LEVEL=DEBUG, en prod LOG_LEVEL=WARNING ✅ **Timestamps automáticos**: Útil para debugging y análisis ✅ **Redirección fácil**: Cambiar de consola a archivo sin modificar código ✅ **Integración con herramientas**: Compatible con Sentry, LogDNA, etc. ✅ **Rendimiento**: Logs DEBUG no se evalúan si LOG_LEVEL=INFO

---

5. UTILIDADES COMPARTIDAS (bots/utils.py)

5.1 Propósito del Módulo

Centralizar funciones reutilizables que se usan en múltiples bots: - Normalización de texto (minúsculas, sin acentos) - Filtrado de anunciantes por keywords - Construcción de key_points (evita duplicación)

5.2 Función `normalize()`

```
import unicodedata
from typing import List, Dict
from config import settings
```

- `unicodedata`: Para normalización Unicode (quitar acentos)
- `typing`: Type hints para claridad
- `config`: Acceso a configuración centralizada

```
def normalize(s):
    if not s:
        return ""
```

- **Guard clause**: Si `s` es None o string vacío, retornar ""
- Evita errores downstream

```
    s = str(s).strip()
```

- `str(s)`: Convierte a string (por si acaso es int u otro tipo)
- `.strip()`: Remueve espacios al inicio/fin (" hola " → “hola”)

```
    s = unicodedata.normalize('NFKD', s)
```

¿Qué es NFKD? - Normalización Unicode: descompone caracteres acentuados - “á” → “a” + acento combinado - Permite remover acentos fácilmente

```
    s = ''.join(c for c in s if not unicodedata.combining(c))
```

- **List comprehension**: Itera cada carácter
- `unicodedata.combining(c)`: Retorna True si es acento combinado
- `not combining(c)`: Solo caracteres NO acentos
- Resultado: “José” → “Jose”, “Español” → “Espanol”

```
    return s.lower()
```

- Convierte a minúsculas para comparaciones case-insensitive
- “HOLA” → “hola”

Ejemplo completo:

```
normalize("  José García  ")
# → "jose garcia"
```

5.3 Función `build_key_points()`

```
def build_key_points(advertisers: List[Dict], max_items: int = None) -> List[Dict]:
    """
    Extrae puntos clave (nombre, descripción, beneficios) de los primeros N anunciantes.
    Centraliza la lógica que antes se repetía en cada bot.

    Args:
        advertisers: Lista de anunciantes
        max_items: Número máximo de puntos clave (default: settings.MAX_KEY_POINTS)

    Returns:
        Lista de dicts con {nombre, descripcion, beneficios}
    """
```

¿Por qué existe?

ANTES (en cada bot):

```
# bot_legal.py
key_points = []
for advertiser in selected_advertisers[:2]:
    point = {
        "nombre": advertiser.get("nombre"),
        "descripcion": advertiser.get("descripcion"),
        "beneficios": advertiser.get("beneficios", [])
    }
    key_points.append(point)
```

```
# bot_accommodation.py
key_points = []  # ← MISMO CÓDIGO REPETIDO
for advertiser in selected_advertisers[:2]:
    # ... idéntico ...
```

DESPUÉS (todos los bots):

```
from .utils import build_key_points

key_points = build_key_points(selected_advertisers)  # ← UNA LÍNEA
```

Código de la función:

```
    if max_items is None:
        max_items = settings.MAX_KEY_POINTS
```

- Si no se especifica `max_items`, usar valor de configuración (default 2)
- Permite sobrescribir: `build_key_points(ads, max_items=5)`

```
    key_points = []
    for advertiser in advertisers[:max_items]:
```

- `advertisers[:max_items]`: Slicing para tomar solo primeros N
- Ejemplo: `[:2]` → primeros 2 elementos

```
        point = {
            "nombre": advertiser.get("nombre"),
            "descripcion": advertiser.get("descripcion"),
            "beneficios": advertiser.get("beneficios", [])
        }
```

- `.get(key, default)`: Método seguro de dict
- Si key no existe → retorna default ([] para beneficios)
- Evita KeyError

```
        key_points.append(point)

    return key_points
```

- Construye lista de puntos clave y retorna

Ejemplo:

```
advertisers = [
    {"nombre": "Abogado A", "descripcion": "Experto NIE", "beneficios": ["24/7"]},
    {"nombre": "Abogado B", "descripcion": "Visas", "beneficios": []},
    {"nombre": "Abogado C", "descripcion": "Contratos", "beneficios": ["Online"]}
]

key_points = build_key_points(advertisers, max_items=2)
# Resultado:
# [
#   {"nombre": "Abogado A", "descripcion": "Experto NIE", "beneficios": ["24/7"]},
#   {"nombre": "Abogado B", "descripcion": "Visas", "beneficios": []}
# ]
```

5.4 Función `filter_advertisers_by_keywords()`

```
def filter_advertisers_by_keywords(pregunta, anunciantes, keywords, search_fields=['nombre', 'descripcion', 'perfil',
        'ubicacion']):
    """
    Función reutilizable para filtrar una lista de anunciantes.

    - Prioriza los anunciantes que coinciden con las palabras clave.
    - Si no hay coincidencias, devuelve el resto.
    """
```

¿Qué hace esta función? 1. Normaliza la pregunta 2. Para cada anunciante, combina campos relevantes en un texto 3. Busca si alguna keyword está presente 4. Divide en dos listas: `matching` (con keywords) y `others` (sin keywords) 5. Ordena cada lista priorizando sponsors 6. Retorna primero los `matching`, luego `others` si no hay `matching`

Código paso a paso:

```
    if not anunciantes:
        return []
```

- Guard clause: Si lista vacía → retornar lista vacía
- Evita iteraciones innecesarias

```
    pregunta_norm = normalize(pregunta or "")
```

- Normaliza la pregunta del usuario
- `pregunta or ""`: Si pregunta es None → usar ""

```
    matching, others = [], []
    for a in anunciantes:
```

- `matching`: Anunciantes que coinciden con keywords
- `others`: Anunciantes que NO coinciden
- Iteramos cada anunciante

```
        combined_text = ' '.join([str(a.get(k, '')) for k in search_fields])
```

Explicación detallada: - `search_fields`: Por default `['nombre', 'descripcion', 'perfil', 'ubicacion']` - Para cada field, obtiene valor del dict: `a.get('nombre', '')` - `str()`: Convierte a string por seguridad - `' '.join()`: Une con espacios

Ejemplo:

```
a = {
    "nombre": "Clínica Dental ABC",
    "descripcion": "Dentista especializado en implantes",
    "ubicacion": "Eixample, Barcelona"
}
search_fields = ['nombre', 'descripcion', 'ubicacion']
combined_text = "Clínica Dental ABC Dentista especializado en implantes Eixample, Barcelona"
```

```
        combined_norm = normalize(combined_text)
```

- Normaliza el texto combinado
- "Clínica Dental ABC..." → "clinica dental abc dentista..."

```
        found = any(kw in pregunta_norm or kw in combined_norm for kw in keywords)
```

Explicación de `any()`: - `any(iterable)`: Retorna True si al menos un elemento es True - Para cada keyword, verifica: - ¿Está en `pregunta_norm`? (ej: "dentista" en "necesito un dentista") - ¿Está en `combined_norm`? (ej: "dentista" en campos del anunciante) - Si alguna keyword coincide → `found = True`

Ejemplo:

```
keywords = ["dentista", "dental", "implante"]
pregunta_norm = "necesito un dentista"
combined_norm = "clinica dental abc dentista especializado implantes..."

# Checking:
# "dentista" in "necesito un dentista" → True ✓
# Result: found = True
```

```
        (matching if found else others).append(a)
```

- **Ternary in Python**: `(A if condition else B).method()`
- Si `found=True` → append a `matching`
- Si `found=False` → append a `others`

```
    def sponsor_key(item):
        flag = item.get('es_anunciante') or item.get('sponsored') or item.get('featured')
        return 1 if flag else 0
```

Función helper para sorting: - Busca si anunciante tiene flag de sponsor/featured - Si tiene cualquiera de los flags → retorna 1 - Si no tiene ninguno → retorna 0 - Se usa para ordenar (1 viene antes que 0 con `reverse=True`)

```
    if matching:
        matching.sort(key=sponsor_key, reverse=True)
        return matching
```

- Si hay anunciantes matching:
  - Los ordena por `sponsor_key` descendente
  - Sponsors (key=1) primero, normales (key=0) después
  - Retorna solo los `matching`

```
    else:
        others.sort(key=sponsor_key, reverse=True)
        return others
```

- Si NO hay matching:
  - Ordena `others` por sponsor
  - Retorna todos (mejor algo que nada)

Ejemplo completo:

```
pregunta = "Necesito un dentista urgente"
anunciantes = [
    {"nombre": "Hotel BCN", "descripcion": "...", "es_anunciante": False},
```

```
    {"nombre": "Clínica Dental", "descripcion": "implantes", "es_anunciante": True},
    {"nombre": "Dentista López", "descripcion": "urgencias", "es_anunciante": False},
]
keywords = ["dentista", "dental", "clinica"]

result = filter_advertisers_by_keywords(pregunta, anunciantes, keywords)
# → [
#     {"nombre": "Clínica Dental", ...},  # ← Sponsor primero
#     {"nombre": "Dentista López", ...}    # ← Match pero no sponsor
#   ]
# Hotel BCN NO está en resultado (no match)
```

5.5 Beneficios de utils.py

✅ **DRY (Don't Repeat Yourself)**: Código compartido en un solo lugar ✅ **Mantenibilidad**: Cambios en una función afectan a todos los bots ✅ **Testing**: Fácil testear funciones aisladas ✅ **Claridad**: Los bots quedan más simples y enfocados

---

[CONTINÚA EN PARTE 2 - El reporte completo tiene más de 15,000 palabras. ¿Quieres que genere el resto en archivos separados o continúo aquí?]

---

6. GESTOR DE CONTENIDO EDITORIAL (bots/content_manager.py)

6.1 Objetivo

Cargar y buscar guías editoriales (JSON) que están en `data/guides/`. Estas guías son contenido de la revista (ej: "Guía NIE Completa", "Sistema de Salud en Barcelona").

6.2 Código Completo Explicado

```
import json
from pathlib import Path
from typing import List, Dict, Optional
```

- `pathlib.Path`: Manejo moderno de rutas (mejor que `os.path`)
- `typing`: Type hints para documentación

```
class ContentManager:
    def __init__(self):
        self.base_dir = Path(__file__).resolve().parent.parent
```

¿Qué hace esto? - `__file__`: Ruta del archivo actual (content_manager.py) - `.resolve()`: Ruta absoluta (resuelve symlinks) - `.parent`: Sube un nivel (de `bots/` a raíz del proyecto) - `.parent` otra vez: Desde raíz

Resultado: Si `content_manager.py` está en `/proyecto/bots/content_manager.py`: - `base_dir = /proyecto/`

```
        self.guides_dir = self.base_dir / "data" / "guides"
        self.articles_dir = self.base_dir / "data" / "articles"
```

- `/` operator de `Path`: Concatena rutas de forma segura
- `self.guides_dir = /proyecto/data/guides/`

```
        self.guides = self._load_guides()
        print(f"✅ Contenido editorial cargado: {len(self.guides)} guías disponibles")
```

- Carga todas las guías al inicializar
- Print feedback (debería ser logger, pero esto se escribió antes)

```
    def _load_guides(self) -> List[Dict]:
        """Carga todas las guías disponibles"""
        guides = []
        if self.guides_dir.exists():
```

- `_load_guides`: Método privado (prefijo _)
- Verifica si directorio existe antes de iterar

```
            for guide_file in self.guides_dir.glob("*.json"):
```

- `.glob("*.json")`: Encuentra todos los archivos `.json`
- Ejemplo: `nie_completa.json`, `sistema_salud.json`

```
                try:
                    with open(guide_file, 'r', encoding='utf-8') as f:
                        guide = json.load(f)
                        guide['tipo'] = 'guia'
                        guides.append(guide)
                except Exception as e:
```

```
                print(f"⚠️  Error cargando {guide_file}: {e}")
```

- Context manager (`with`): Auto-cierra archivo
- `json.load(f)`: Parsea JSON a dict de Python
- Añade campo `tipo='guia'` para identificación
- Try/except: Si un archivo está corrupto, continúa con los demás

```
        return guides
```

6.3 Función `search_guides()`

```
    def search_guides(self, keywords: List[str], categoria: str = None) -> List[Dict]:
        """
        Busca guías relevantes basándose en keywords y categoría.

        Args:
            keywords: Lista de palabras clave para buscar
            categoria: Categoría específica (Healthcare, Legal, etc.)

        Returns:
            Lista de guías relevantes ordenadas por relevancia
        """
```

Sistema de scoring:

```
        relevant_guides = []

        for guide in self.guides:
            score = 0
```

- Inicia score en 0 para cada guía
- Irá acumulando puntos

```
            # Coincidencia de categoría (peso 3)
            if categoria and guide.get('categoria') == categoria:
                score += 3
```

- Si la guía es de la categoría correcta → +3 puntos
- Ejemplo: Busco “Legal”, guía es “Legal” → score=3

```
            # Coincidencia en keywords de la guía (peso 2)
            guide_keywords = [k.lower() for k in guide.get('keywords', [])]
            for keyword in keywords:
                if any(keyword.lower() in gk for gk in guide_keywords):
                    score += 2
```

- Cada guía tiene un array de keywords
- Si keyword del usuario coincide con keyword de guía → +2
- `any()`: Al menos una coincidencia

```
            # Coincidencia en título (peso 1)
            titulo = guide.get('titulo', '').lower()
            for keyword in keywords:
                if keyword.lower() in titulo:
                    score += 1
```

- Si keyword está en el título → +1
- Peso menor porque título puede tener palabras genéricas

```
            if score > 0:
                guide_copy = guide.copy()
                guide_copy['relevancia'] = score
                relevant_guides.append(guide_copy)
```

- Solo añade guías con score > 0
- Copia la guía y añade campo `relevancia`

```
        # Ordenar por relevancia
        relevant_guides.sort(key=lambda x: x['relevancia'], reverse=True)
        return relevant_guides
```

- Ordena de mayor a menor relevancia
- Las más relevantes primero

Ejemplo:

```
keywords = ["nie", "residencia", "visa"]
categoria = "Legal and Financial"

guides = [
    {"titulo": "Guía NIE Completa", "categoria": "Legal and Financial", "keywords": ["nie", "extranjeria"]},
```

```
    {"titulo": "Trabajar en Barcelona", "categoria": "Work", "keywords": ["empleo", "visa"]},
]

result = content_manager.search_guides(keywords, categoria)
# Guía 1: cat match(+3) + kw "nie"(+2) + título "nie"(+1) = 6
# Guía 2: kw "visa"(+2) = 2
# Orden: [Guía 1, Guía 2]
```

6.4 Función `get_guide_summary()`

```
    def get_guide_summary(self, guide: Dict) -> Dict:
        """
        Devuelve un resumen corto de la guía para mostrar en el chat.
        """
        return {
            "tipo": "guia_revista",
            "titulo": guide.get('titulo'),
            "resumen": guide.get('resumen'),
            "slug": guide.get('slug'),
            "url": f"/revista/guias/{guide.get('slug')}",
            "categoria": guide.get('categoria')
        }
```

- Extrae solo campos relevantes para el frontend
- Construye URL dinámica usando slug
- Ejemplo: slug="nie-completa" → url="/revista/guias/nie-completa"

---

7. GESTOR DE FEEDS RSS (bots/rss_manager.py)

7.1 Objetivo

Parsear los feeds RSS/Atom de Barcelona Metropolitan, almacenar artículos en caché local y proveer búsqueda por palabras clave y por categoría.

7.2 Código Explicado Línea a Línea (fragmentos clave)

```
"""
RSS Feed Manager: parsea feeds de la revista y cachea artículos localmente.
Actualiza cada 6-12 horas en background.
"""
```

- Docstring que describe el propósito del módulo y la frecuencia de actualización.

```
import json
import feedparser
from pathlib import Path
from datetime import datetime
from typing import List, Dict, Optional
from .logger import logger
```

- `feedparser`: librería para parsear RSS/Atom.
- `Path`: manejo de rutas.
- `datetime`: timestamps para `synced_at`.
- `logger`: logs estructurados.

```
class RSSManager:
    def __init__(self):
        self.base_dir = Path(__file__).resolve().parent.parent
        self.cache_dir = self.base_dir / "data" / "cache"
        self.cache_dir.mkdir(parents=True, exist_ok=True)
        self.articles_cache = self.cache_dir / "articles.json"
```

- Calcula ruta base y asegura que `data/cache` exista.
- Define archivo `articles.json` para caché.

```
        self.feed_urls = [
            "https://www.barcelona-metropolitan.com/directory/index.rss",
        ]
```

- Lista de feeds a sincronizar. Se pueden añadir más feeds.

```
        self.articles = self.load_cache()
        logger.info(f"RSS Manager inicializado. {len(self.articles)} artículos en caché.")
```

- Carga caché al iniciar y reporta cantidad en logs.

```
    def load_cache(self) -> List[Dict]:
        if self.articles_cache.exists():
            try:
```

```
            with open(self.articles_cache, 'r', encoding='utf-8') as f:
                return json.load(f)
        except Exception as e:
            logger.warning(f"Error cargando caché: {e}")
    return []
```

- Lectura segura del archivo de caché con manejo de errores.

```
def save_cache(self):
    try:
        with open(self.articles_cache, 'w', encoding='utf-8') as f:
            json.dump(self.articles, f, ensure_ascii=False, indent=2)
    except Exception as e:
        logger.error(f"Error guardando caché: {e}")
```

- Persistencia de artículos en disco en formato JSON legible.

```
def sync_feeds(self) -> int:
    new_count = 0
    existing_urls = {a.get('url') for a in self.articles}
    for feed_url in self.feed_urls:
        try:
            logger.info(f"Parseando feed: {feed_url}")
            feed = feedparser.parse(feed_url)
            if not feed.entries:
                logger.warning(f"Feed vacío o no accesible: {feed_url}")
                continue
            entries_count = 0
            for entry in feed.entries[:50]:
                article = {
                    "url": entry.get('link', '') or entry.get('id', ''),
                    "title": entry.get('title', 'Sin título'),
                    "description": entry.get('summary', entry.get('description', ''))[:500],
                    "published": entry.get('published', entry.get('updated', '')),
                    "categories": [t.get('term') if isinstance(t, dict) else str(t) for t in entry.get('tags', [])],
                    "source": feed.feed.get('title', 'Barcelona Metropolitan'),
                    "synced_at": datetime.utcnow().isoformat()
                }
                if article['url'] and article['url'] not in existing_urls:
                    self.articles.append(article)
                    existing_urls.add(article['url'])
                    new_count += 1
                    entries_count += 1
            logger.info(f"Feed '{feed.feed.get('title', 'Unknown')}': {entries_count} nuevas entradas parseadas.")
        except Exception as e:
            logger.error(f"Error parseando {feed_url}: {e}")
    if len(self.articles) > 1000:
        self.articles = sorted(self.articles, key=lambda x: x.get('synced_at', ''), reverse=True)[:1000]
    self.save_cache()
    logger.info(f"Sync completado: {new_count} artículos nuevos. Total: {len(self.articles)}")
    return new_count
```

- Recorre todos los feeds, parsea entradas y evita duplicados por URL.
- Limita a 50 por feed y máx. 1000 en caché.
- Guarda caché y retorna número de nuevos artículos.

```
def search_articles(self, keywords: List[str], limit: int = 5) -> List[Dict]:
    if not keywords or not self.articles:
        return []
    keyword_set = {kw.lower() for kw in keywords if len(kw) > 2}
    scored = []
    for article in self.articles:
        text = f"{article.get('title', '')} {article.get('description', '')} " \
               f"{' '.join([t.get('term', '') if isinstance(t, dict) else str(t) for t in article.get('categories',
[])])}".lower()
        score = sum(1 for kw in keyword_set if kw in text)
        if score > 0:
            scored.append((score, article))
    scored.sort(key=lambda x: x[0], reverse=True)
    return [article for _, article in scored[:limit]]
```

- Búsqueda por keywords en título, descripción y tags con scoring simple.

```
def get_articles_by_category(self, category: str, limit: int = 3) -> List[Dict]:
    category_keywords = {
        "Legal and Financial": ["legal", "visa", "nie", "impuesto", "contrato", "bank", "immigration"],
        "Healthcare": ["salud", "doctor", "hospital", "dentista", "health", "medical", "clinic"],
        "Education": ["escuela", "colegio", "university", "idioma", "course", "school", "education"],
        "Accommodation": ["alojamiento", "hotel", "apartamento", "vivienda", "housing", "apartment", "rent"],
        "Restaurants": ["restaurante", "comida", "food", "restaurant", "dining", "cuisine"],
        "Arts and Culture": ["arte", "cultura", "museo", "gallery", "culture", "exhibition"],
        "Work and Networking": ["trabajo", "empleo", "job", "networking", "business", "work"],
    }
```

```
        keywords = category_keywords.get(category, [])
        return self.search_articles(keywords, limit)
```

- Mapeo manual de categorías → keywords; delega a `search_articles`.

7.3 Errores Comunes y Soluciones

- Feed vacío/no accesible: revisar URL y conexión; ver logs `WARNING Feed vacío`.
- Duplicados: se evita por URL; si un feed usa `id` distinto, se toma `link`.
- Crecimiento infinito: límite 1000 artículos; ajustar en `config.py` si necesario.

---

8. ORQUESTADOR PRINCIPAL (bots/orchestrator.py)

8.1 Responsabilidades

- Cargar base de datos de anunciantes.
- Clasificar intención con embeddings y fallback de keywords.
- Enrutar a bot especializado.
- Enriquecer con guías y artículos.
- Aplicar paginación `limit/offset` y devolver `has_more/next_offset`.

8.2 Código Explicado (puntos clave)

```
from sentence_transformers import SentenceTransformer, util
from .rss_manager import get_rss_manager
from config import settings
```

- Importa modelo de embeddings, RSS Manager y configuración.

```
class Orchestrator:
    def __init__(self):
        base_dir = Path(__file__).resolve().parent.parent
        data_path = base_dir / "data" / "anunciantes.json"
        with open(str(data_path), 'r', encoding='utf-8') as f:
            self.advertisers = json.load(f)
        self.content_manager = ContentManager()
        self.rss_manager = get_rss_manager()
        self.model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')
```

- Carga anunciantes y servicios auxiliares.
- Inicia modelo multilingüe para embeddings.

```
        self.category_patterns = {"es": { ... }, "en": { ... }}
```

- Palabras clave por idioma y categoría para fallback y tips.

```
        self.bots_map = {
            "Accommodation": acc_responder,
            "Legal and Financial": leg_responder,
            ...,
            "Restaurants": generic_responder,
        }
```

- Mapa categoría → función `responder_consulta` del bot adecuado; genérico para categorías simples.

```
        self.category_info = []
        for category in all_categories:
            description = f"Servicios sobre {category.lower()}: " + ", ".join(keywords_es + keywords_en)
            self.category_info.append({
                "name": category,
                "description": description,
                "embedding": self.model.encode(description, convert_to_tensor=True)
            })
```

- Precalcula embeddings por categoría para velocidad de clasificación.

```
    def classify_intent(self, question, language="en"):
        qn = normalize(question)
        question_embedding = self.model.encode(question, convert_to_tensor=True)
        import torch
        category_embeddings = torch.stack([cat["embedding"] for cat in self.category_info])
        cos_scores = util.cos_sim(question_embedding, category_embeddings)[0]
        top_result = int(cos_scores.argmax().item())
        best_score = float(cos_scores[top_result].item())
        best_match_category = self.category_info[top_result]["name"]
```

- Convierte pregunta a embedding, calcula similitud coseno y toma la categoría con mayor score.

```
        for category, businesses in self.advertisers.items():
```

```
        for business in businesses:
            name = normalize(business.get('nombre', ''))
            if name and name in qn:
                return category, 0.9, business
```

- Override por nombre explícito de negocio: prioridad absoluta si el usuario nombra un anunciante.

```
    kw_map = self.category_patterns.get(language, {})
    best_kw_cat = None; best_hits = 0
    for cat, kws in kw_map.items():
        hits = sum(1 for kw in kws if kw in qn)
        if hits > best_hits:
            best_hits, best_kw_cat = hits, cat
    if best_kw_cat and best_hits > 0:
        return best_kw_cat, max(0.25, min(0.9, 0.15 * best_hits + 0.25)), None
```

- Fallback por keywords: calcula “confianza” basada en hits usando parámetros de `config.py` (modelo matemático sencillo: base + multiplicador).

```
    if best_score > 0.2:
        return best_match_category, best_score, None
    else:
        return "Desconocida", best_score, None
```

- Umbral de confianza: si la similitud es baja, marca como desconocido.

```
 def process_query(self, question, language="en", limit: int = 3, offset: int = 0):
    categoria, confidence, advertiser = self.classify_intent(question, language)
    resultados = self.advertisers.get(categoria, [])
    if advertiser and advertiser not in resultados:
        resultados.insert(0, advertiser)
    keywords = [word.lower() for word in question.split() if len(word) > 3]
    guias_relevantes = self.content_manager.search_guides(keywords, categoria)
    articulos_revista = self.rss_manager.get_articles_by_category(categoria, limit=3)
    bot_response = self.bots_map[categoria](question, resultados, language=lang)
    all_items = bot_response.get("json_data", [])
    total = len(all_items)
    sliced = all_items[offset:offset + limit] if (limit not in (None, 0)) else all_items[offset:]
    has_more = (offset + (limit or 0)) < total if limit not in (None, 0) else False
    next_offset = (offset + (limit or 0)) if has_more else None
```

- Enriquecimiento y paginación: slicing y cálculo de `has_more/next_offset`.

8.3 Errores y Soluciones

- Modelo no instalado: ejecutar `pip install sentence-transformers torch`.
- `anunciantes.json` faltante: verificar ruta `data/anunciantes.json`.
- Clasificación confusa: ajustar `CONFIDENCE_THRESHOLD` en `config.py`.

---

9. BOTS ESPECIALIZADOS

Cada bot expone `responder_consulta(pregunta, anunciantes, language)` y retorna `{ key_points, json_data }`. La lógica común se apoya en utilidades compartidas.

9.1 Patrón Común

- Definir `keywords` por categoría.
- Usar `filter_advertisers_by_keywords()` para priorizar sponsors con match.
- Construir `key_points` con `build_key_points()`.
- Retornar lista completa; el orquestador aplica paginación.

9.2 Ejemplos

Bot Legal:

```
from .utils import filter_advertisers_by_keywords, build_key_points
def responder_consulta(pregunta, anunciantes, language="en"):
    keywords = ["abogado", "legal", "impuestos", "nie", "visa", "bank", ...]
    selected = filter_advertisers_by_keywords(pregunta, anunciantes, keywords)
    key_points = build_key_points(selected)
    return {"key_points": key_points, "json_data": selected}
```

Bot Healthcare (con integración Maps):

```
from .maps_integration import search_healthcare_barcelona
selected = filter_advertisers_by_keywords(pregunta, anunciantes, keywords)
for advertiser in selected:
    advertiser["es_anunciante"] = True
```

```
if len(selected) < 3:
    search_term = "hospital"  # ajustado según la pregunta
    maps_results = search_healthcare_barcelona(search_term, limit=5)
    selected.extend(maps_results)
return {"key_points": build_key_points(selected), "json_data": selected}
```

Bot Accommodation:

```
selected = filter_advertisers_by_keywords(pregunta, anunciantes, keywords)
return {"key_points": build_key_points(selected), "json_data": selected}
```

9.3 Soluciones a Problemas Comunes

- Demasiados resultados irrelevantes: ajustar `keywords` y `search_fields` en `utils.filter_advertisers_by_keywords`.
- Falta de `beneficios`: `build_key_points()` usa `[]` por defecto; completar datos en `anunciantes.json`.

---

10. BACKEND API (main.py)

10.1 Setup

- FastAPI con `lifespan` para scheduler.
- `BackgroundScheduler` de APScheduler para sync RSS cada 6 horas.
- CORS abierto en desarrollo; restringir en producción.

10.2 Modelos

```
class QueryRequest(BaseModel):
    pregunta: str = None
    question: str = None
    language: str = "es"
    session_id: str = None
    limit: int | None = 5
    offset: int = 0
    def get_question(self):
        return self.pregunta or self.question or ""
```

- Compatibilidad con `pregunta` y `question`.
- Paginación: `limit/offset`.

10.3 Endpoints

- `GET /api/health`: health check.
- `GET /api/categories`: categorías disponibles.
- `GET /api/sync-rss`: sincronización manual.
- `POST /api/query`: consulta principal, loguea, enruta y responde.
- `POST /api/analytics`: guarda eventos en `data/analytics/events.jsonl`.

10.4 Errores y Soluciones

- 500 en `/api/query`: revisar logs `ERROR en /api/query`; verificar que Orchestrator se inicializó.
- CORS bloquea requests externos: configurar `allow_origins` con dominios permitidos.

---

11. FRONTEND (HTML/CSS/JS)

11.1 index.html

- Estructura con secciones: Header, Hero, Artículos, Guías, Servicios, Anunciantes, Newsletter, Footer.
- Chat flotante (widget-like) integrado al final.
- Carga `script.js` y `styles.css`.

11.2 styles.css

- Variables CSS, diseño responsivo, componentes estilizados (cards, carrusel, chat).
- Optimizado para lectura y acciones (CTAs destacadas).

11.3 script.js (puntos clave)

- `sendQuery()`: envía POST con `{question, language, limit, offset}`; maneja errores con diagnóstico en consola.
- `updateBotMessage()`: renderiza respuesta, guías, artículos RSS, resultados, botón “Mostrar más”, y tracking de clics.
- Paginación: mantiene `lastQuery` y `nextOffset` para cargar más resultados.
- FAQ desplegable: acordeón interactivo por anunciante.
- Quick actions: botones predefinidos para categorías frecuentes.

---

12. WIDGET EMBEBIBLE (widget.js / embed.js)

12.1 embed.js

- Carga `widget.css` y `widget.js` desde el dominio de la API.
- Inicializa el widget con `ExpatAssistantWidget(config)`.

12.2 widget.js

- Clase `ExpatAssistantWidget` con configuración (API URL, color, sugerencias).
- Métodos: `init()`, `createWidget()`, `attachEventListeners()`, `sendMessage()`, `addAdvertisers()`, `addGuides()`, `trackEvent()`.
- Renderiza mensajes, muestra typing indicator y envía requests a `/api/query`.
- Integra tracking (Google Analytics, Meta Pixel y backend propio).

12.3 Consideraciones

- Personalizable vía `window.ExpatAssistantConfig`.
- Se puede montar en cualquier sitio externo sin conflictos.

---

13. EVOLUCIÓN Y MEJORAS

- Integración RSS con caché y scheduler.
- Paginación `limit/offset` + botón “Mostrar más” en frontend.
- Configuración centralizada con `config.py` (Pydantic BaseSettings).
- Logging estructurado con `bots/logger.py`.
- DRY: `build_key_points()` y `filter_advertisers_by_keywords()` en `utils.py`.
- Limpieza de dependencias (Flask removido, FastAPI activo).
- Guías editoriales y tips por categoría.

---

14. TROUBLESHOOTING

- “El feed no sincroniza”: probar `GET /api/sync-rss`; verificar conectividad; revisar logs.
- “Error de clasificación”: bajar/ajustar `CONFIDENCE_THRESHOLD` en `config.py`.
- “CORS bloqueado”: configurar `allow_origins` en `main.py` con dominios reales.
- “Sin anunciantes”: comprobar `data/anunciantes.json` y formato de campos; ajustar `search_fields`.

---

15. CONCLUSIONES

- Arquitectura modular, escalable y mantenible.
- Experiencia de usuario rica: recomendaciones + contenido editorial + paginación.
- Preparado para producción con logging, config central y scheduler.

---

16. EXPORTAR A PDF

Opciones para generar el PDF desde este reporte:

- Usando `pandoc` (recomendado):
  - Instalar: `sudo apt-get install pandoc wkhtmltopdf` (opcional para PDF via HTML).
  - Comando: `pandoc REPORTE_PROYECTO_COMPLETO.md -o REPORTE_Revista_Expats_AI.pdf`.
- Usando `wkhtmltopdf` (si prefieres render HTML):
  - `wkhtmltopdf REPORTE_PROYECTO_COMPLETO.md REPORTE_Revista_Expats_AI.pdf` (requiere convertir Markdown a HTML primero: `pandoc -s REPORTE_PROYECTO_COMPLETO.md -o reporte.html` y luego `wkhtmltopdf reporte.html reporte.pdf`).

Sugerencia: incluye cabecera/portada con `--metadata` en pandoc y estilo CSS para una salida profesional.

---

17. ANEXO: CÓDIGO CLAVE INCLUIDO

Para referencia rápida, se han explicado y mostrado los fragmentos críticos en las secciones anteriores:
- Configuración central: `config.py`.
- Logger: `bots/logger.py`.
- Utilidades: `bots/utils.py`.
- Gestor RSS: `bots/rss_manager.py`.
- Orquestador: `bots/orchestrator.py`.
- Bots (Legal, Healthcare, Accommodation).
- Backend FastAPI: `main.py`.

- Frontend: `frontend/index.html`, `frontend/script.js`, `frontend/styles.css`.
- Widget: `widget/embed.js`, `widget/widget.js`, `widget/widget.css`.

Si deseas añadir un “Anexo B” con el listado completo de todos los archivos del repositorio (contenido íntegro), puedo incorporarlo automáticamente en este documento antes de exportar a PDF.