Añadiendo llms.txt a tu Blog de Astro: Una Guía Paso a Paso #### Introducción: ¿Por Qué tu Blog Necesita Hablar con las IAs? Este tutorial te guiará, paso a paso, en el proceso de añadir un archivo llms.txt a tu blog de Astro. Lo haremos desde cero, sin instalar ninguna dependencia externa, para que entiendas cada parte del proceso y mantengas el control total sobre el resultado. ##### ¿Qué es llms.txt? llms.txt es un estándar propuesto para que los sitios web sean más fáciles de leer para los agentes de Inteligencia Artificial (IA). Piensa en él como un robots.txt para los crawlers o un sitemap.xml para los motores de búsqueda, pero diseñado específicamente para los LLMs (Grandes Modelos de Lenguaje). El objetivo es simple: en lugar de que una IA tenga que analizar el complejo HTML de tu sitio, le ofreces el contenido puro y estructurado en un formato de texto limpio que puede entender directamente. ¿La prueba de que funciona? La propia documentación de Astro utiliza este patrón. Cuando le pides a un agente que te ayude con Astro, puede obtener su llms.txt y darte información precisa y actualizada en lugar de depender de datos de entrenamiento obsoletos. ##### El Caso de Uso Práctico Para entender el beneficio, imagina que estás interactuando con un agente de IA desde tu terminal: **Tú:** "Oye, ¿puedes hacer curl a https://kumak.dev/llms-full.txt y decirme si hay algo interesante sobre Astro?" **Agente:** *(Obtiene markdown limpio, escanea el contenido)* "Hay un post sobre colecciones de contenido y otro sobre cómo añadir llms.txt. El de las colecciones de contenido cubre..." O un ejemplo aún más práctico: **Tú:** "Implementa https://kumak.dev/llms/adding-llms-txt-to-astro.txt en mi blog" **Agente:** *(Obtiene el post como markdown limpio)* "Entendido. Veo que necesitas tres endpoints. Déjame crear primero el archivo de utilidades..." Sin llms.txt, el agente tendría que descargar el HTML, eliminar la navegación, los pies de página, analizar componentes de React y esperar obtener el contenido correcto. Con llms.txt, obtiene exactamente lo que necesita en un formato que puede leer de inmediato, ahorrando tiempo y tokens de procesamiento. Esta es la ventaja principal: llms.txt proporciona a las IAs contenido limpio y estructurado en Markdown. Con esta base clara del "porqué", veamos ahora el "cómo" implementaremos la arquitectura necesaria en nuestro proyecto de Astro. #### 1. Entendiendo la Arquitectura: Los Tres Endpoints Clave Nuestra

implementación se basará en tres endpoints que trabajan en conjunto para ofrecer el contenido de diferentes maneras, según las necesidades del agente de IA. * /llms.txt: **El Índice.** Actúa como una tabla de contenidos. Le dice al agente: "Esto es lo que tengo disponible". * /llms-full.txt: **El Contenido Completo.** Ofrece todos los posts del blog en un único archivo. Ideal para sistemas que necesitan todo el contexto de una vez, como un sistema RAG (Retrieval-Augmented Generation). * /llms/slug.txt: **El Post Individual.** Proporciona un único post específico, permitiendo a un agente realizar una consulta enfocada sin la sobrecarga de descargar todo el contenido. Se necesitan tres endpoints porque los diferentes agentes de IA tienen distintas necesidades. Un agente que busca una respuesta rápida puede usar el índice para encontrar un post relevante. Un sistema RAG podría querer descargar todo el contenido de una vez para indexarlo. Y una consulta específica sobre un tema solo necesita el post individual. Ahora que entendemos la arquitectura conceptual, vamos a ver cómo se traduce esto a la estructura de archivos de nuestro proyecto en Astro. #### 2. Preparando tu Proyecto Antes de escribir código, organizaremos la estructura de archivos y nos aseguraremos de tener todos los prerrequisitos listos. ##### Estructura de Archivos Añadiremos cuatro nuevos archivos a nuestro proyecto. La lógica principal vivirá en el directorio utils/, y los endpoints se crearán en pages/. text src/ ├── utils/ │ └── llms.ts # Toda la lógica de generación ├── pages/ │ ├── llms.txt.ts # Endpoint del índice │ ├── llms-full.txt.ts # Endpoint de contenido completo │ └── llms/ │ └── [slug].txt.ts # Endpoint dinámico por post Esta separación mantiene nuestro código organizado. Los archivos en pages/ serán "envoltorios" (wrappers) muy delgados que simplemente llaman a la lógica centralizada en utils/llms.ts. ##### Prerrequisitos Para que esta implementación funcione, tu proyecto de Astro ya debería tener las siguientes piezas: 1. **siteConfig** : Un objeto de configuración global que contenga propiedades como name, description, url y author de tu sitio. 2. **getAllPosts()** : Una función que devuelva un array con todos los posts de tus colecciones de contenido de Astro. 3. **BlogPost** : El tipo de TypeScript que define la estructura de un post, proveniente de las colecciones de contenido de Astro (debe incluir slug, body y data). Con la estructura de archivos definida y los prerrequisitos listos, es hora de empezar a escribir el código.

Empezaremos por el corazón de nuestra implementación: el archivo de utilidades. #### 3. Paso 1: Construyendo el Motor de Lógica en utils/llms.ts Toda la lógica para generar los archivos de texto vivirá en src/utils/llms.ts. Esto mantiene nuestro proyecto limpio y el código reutilizable. ##### 3.1: Definiendo los Tipos de Datos Usar TypeScript nos ayuda a crear código autodocumentado y a evitar errores. Primero, definiremos la forma de nuestros datos. typescript // src/utils/llms.ts // Ítem básico para el índice - lo justo para crear un enlace interface LlmsItem { title: string; description: string; link: string; } // Ítem extendido para el contenido completo - incluye los datos del post interface LlmsFullItem extends LlmsItem { pubDate: Date; category: string; body: string; } ¿Por qué dos tipos? El índice (/llms.txt) solo necesita títulos y enlaces. El volcado de contenido completo (/llms-full.txt) necesita todos los detalles del post. Al extender LlmsItem, mantenemos la consistencia. A continuación, los tipos para la configuración de cada generador: typescript // Configuración para el endpoint del índice interface LlmsTxtConfig { name: string; description: string; site: string; items: LlmsItem[]; optional?: LlmsItem[]; // Enlaces que los agentes pueden omitir } // Configuración para el endpoint de contenido completo interface LlmsFullTxtConfig { name: string; description: string; author: string; site: string; items: LlmsFullItem[]; } // Configuración para los endpoints de post individuales interface LlmsPostConfig { post: BlogPost; // Asume que tienes un tipo BlogPost de tus colecciones site: string; link: string; } El campo optional en LlmsTxtConfig es parte de la especificación llms.txt. Le indica a los agentes: "estos enlaces son secundarios, puedes ignorarlos si tienes un contexto limitado". ##### 3.2: El Constructor Universal de Documentos Todos nuestros endpoints devolverán una Response de texto plano. Para evitar repetir código, creamos una función constructora universal. typescript function doc(...sections: (string | string[])[]): Response { const content = sections .flat() // Aplana arrays anidados .join("\n") // Une todo con saltos de línea .replace(/\n{3,}/g, "\n\n") // Normaliza múltiples líneas en blanco a una sola .trim(); // Limpia los bordes return new Response(content + "\n", { headers: { "Content-Type": "text/plain; charset=utf-8" }, }); } Esta función doc() es muy flexible. ¿Por qué usar parámetros rest (...) con arrays? Porque nos permite componer documentos de forma flexible, ya sea pasando strings sueltos (doc("# Título", "Texto")) o arrays completos

(doc(arrayCabecera, arrayCuerpo)). ¿Y por qué normalizar los saltos de línea? Al componer el documento desde múltiples arrays, podrías terminar accidentalmente con tres o cuatro líneas en blanco seguidas. El regex /\n{3,}/g se asegura de que cualquier secuencia de 3 o más saltos de línea se reemplace por solo 2 (una línea en blanco). El resultado es un output limpio, sin importar lo desordenada que sea la entrada. ##### 3.3: Funciones Auxiliares para Formateo Crearemos pequeñas funciones reutilizables para formatear las diferentes partes de nuestros documentos. typescript function formatDate(date: Date): string { return date.toISOString().split("T")[0]; } function header(name: string, description: string): string[] { return [`# ${name}`, "", `> ${description}`]; } function linkList(title: string, items: LlmsItem[], site: string): string[] { return ["", `## ${title}`, ...items.map((item) => `- [${item.title}](${site}${item.link}): ${item.description}`),]; } function postMeta(site: string, link: string, pubDate: Date, category: string): string[] { return [`URL: ${site}${link}`, `Published: ${formatDate(pubDate)}`, `Category: ${category}`]; } * formatDate: Convierte un objeto Date a un formato YYYY-MM-DD. El truco split("T")0 extrae solo la parte de la fecha de un string ISO como "2025-11-30T00:00:00.000Z". * header: Crea la cabecera del documento. El string vacío "" añade una línea en blanco entre el título y la descripción, siguiendo el formato de la especificación. * linkList: Genera una sección con un H2 y una lista de enlaces. El string vacío al principio asegura que siempre haya una línea en blanco antes de la sección. * postMeta: Crea las tres líneas de metadatos (URL, fecha, categoría) para un post, manteniendo un formato consistente. ##### 3.4: Limpiando Contenido MDX Si tus posts están escritos en MDX, pueden contener sentencias import y componentes JSX que no son útiles para las IAs. Necesitamos una función para eliminarlos. typescript const MDX_PATTERNS = [/^import\s+.+from\s+['"].+['"];?\s*$/gm, // sentencias import /<[A-Z][a-zA-Z]*[^>]*>[\s\S]*?<\/[A-Z][a-zA-Z]*>/g, // bloques JSX como ... /<[A-Z][a-zA-Z]*[^>]*\/>/g, // JSX de autocierre como] as const; function stripMdx(content: string): string { return MDX_PATTERNS.reduce((text, pattern) => text.replace(pattern, ""), content).trim(); } **¿Cómo funcionan estos patrones?** La clave está en la convención de PascalCase para los componentes JSX. Las expresiones regulares solo buscan etiquetas que empiecen con una

letra mayúscula (p. ej.). Esto significa que el HTML estándar (como o) y los bloques de código se mantendrán intactos. Es una forma segura de limpiar el contenido sin dañar el markdown. ##### 3.5: Los Generadores Principales Ahora, combinamos todas nuestras funciones auxiliares para crear las tres funciones que exportaremos. typescript // Generador para el índice /llms.txt export function llmsTxt(config: LlmsTxtConfig): Response { const sections = [header(config.name, config.description), linkList("Posts", config.items, config.site),]; if (config.optional?.length) { sections.push(linkList("Optional", config.optional, config.site)); } return doc(...sections); } typescript // Generador para el contenido completo /llms-full.txt export function llmsFullTxt(config: LlmsFullTxtConfig): Response { const head = [...header(config.name, config.description), "", `Author: ${config.author}`, `Site: ${config.site}`, "", "---",]; const posts = config.items.flatMap((item) => ["", `## ${item.title}`, "", ...postMeta(config.site, item.link, item.pubDate, item.category), "", `> ${item.description}`, "", stripMdx(item.body), "", "---",]); return doc(head, posts); } ¿Por qué flatMap? Cada post genera un array de líneas de texto (título, metadatos, cuerpo, etc.). Si usáramos map, obtendríamos un array de arrays. flatMap nos permite mapear y aplanar el resultado en un solo paso, dándonos un único array plano de líneas listo para la función doc(). typescript // Generador para un post individual /llms/[slug].txt export function llmsPost(config: LlmsPostConfig): Response { const { post, site, link } = config; const { title, description, pubDate, category } = post.data; return doc(`# ${title}`, "", `> ${description}`, "", ...postMeta(site, link, pubDate, category), "", stripMdx(post.body ?? "")); } ##### 3.6: Transformadores de Datos de Astro Finalmente, necesitamos un par de funciones para convertir los objetos de las colecciones de contenido de Astro a nuestros tipos LlmsItem y LlmsFullItem. typescript export function postsToLlmsItems(posts: BlogPost[], formatUrl: (slug: string) => string): LlmsItem[] { return posts.map((post) => ({ title: post.data.title, description: post.data.description, link: formatUrl(post.slug), })); } export function postsToLlmsFullItems(posts: BlogPost[], formatUrl: (slug: string) => string): LlmsFullItem[] { return posts.map((post) => ({ ...postsToLlmsItems([post], formatUrl)[0], pubDate: post.data.pubDate, category: post.data.category, body: post.body ?? "", })); } Aquí hay dos

decisiones de diseño importantes: 1. **Callback formatUrl** : ¿Por qué una función callback? Porque cada endpoint necesita un formato de enlace diferente: el índice enlaza a /llms/slug.txt, mientras que el contenido completo enlaza a la página HTML /slug. Pasar el formateador como callback hace que los transformadores sean flexibles y reutilizables. 2. **Reutilización de código (DRY)** : postsToLlmsFullItems reutiliza postsToLlmsItems para generar la base del objeto. En lugar de duplicar la lógica de mapeo, la reutilizamos y esparcimos el resultado (...), añadiendo solo los campos adicionales. Con toda la lógica en su lugar en utils/llms.ts, el siguiente paso es conectar todo en nuestras páginas de Astro. #### 4. Paso 2: Creando los Endpoints en Astro Ahora crearemos tres archivos en src/pages/ que importarán y utilizarán las funciones que acabamos de construir. Estos archivos serán muy "delgados", ya que la mayor parte del trabajo ya está hecha. ##### 4.1: El Endpoint del Índice (/llms.txt) Crea el siguiente archivo en src/pages/llms.txt.ts. El nombre .txt.ts le indica a Astro que debe generar un archivo de texto plano. typescript // src/pages/llms.txt.ts import type { APIRoute } from "astro"; import { siteConfig } from "@/site-config"; import { llmsTxt, postsToLlmsItems } from "@utils/llms"; import { getAllPosts } from "@utils/posts"; export const GET: APIRoute = async () => { const posts = await getAllPosts(); return llmsTxt({ name: siteConfig.name, description: siteConfig.description, site: siteConfig.url, items: postsToLlmsItems(posts, (slug) => `/llms/${slug}.txt`), optional: [{ title: "About", link: "/about", description: "About the author" }, { title: "Full Content", link: "/llms-full.txt", description: "All posts in one file" },], }); }; La función GET obtiene todos los posts, los transforma usando postsToLlmsItems (especificando que los enlaces deben apuntar a la versión .txt), y finalmente llama al generador llmsTxt con toda la configuración. ##### 4.2: El Endpoint de Contenido Completo (/llms-full.txt) Crea este archivo en src/pages/llms-full.txt.ts. typescript // src/pages/llms-full.txt.ts import type { APIRoute } from "astro"; import { siteConfig } from "@/site-config"; import { llmsFullTxt, postsToLlmsFullItems } from "@utils/llms"; import { getAllPosts } from "@utils/posts"; export const GET: APIRoute = async () => { const posts = await getAllPosts(); return llmsFullTxt({ name: siteConfig.name, description: siteConfig.description, author: siteConfig.author, site: siteConfig.url, items: postsToLlmsFullItems(posts, (slug) => `/${slug}`),

}); }; Este endpoint es muy similar al anterior, pero utiliza llmsFullTxt y postsToLlmsFullItems. Nota cómo el formateador de URL ahora apunta a las páginas HTML (/${slug}), ya que un agente que lea el volcado completo puede querer la referencia a la página original. ##### 4.3: Los Endpoints Dinámicos por Post (/llms/slug.txt) Finalmente, crea el archivo para las rutas dinámicas en src/pages/llms/slug.txt.ts. typescript // src/pages/llms/[slug].txt.ts import type { GetStaticPaths } from "astro"; import { siteConfig } from "@/site-config"; import { llmsPost } from "@utils/llms"; import { getAllPosts, type BlogPost } from "@utils/posts"; export const getStaticPaths: GetStaticPaths = async () => { const posts = await getAllPosts(); return posts.map((post) => ({ params: { slug: post.slug }, props: { post }, })); }; export const GET = ({ props }: { props: { post: BlogPost } }) => { return llmsPost({ post: props.post, site: siteConfig.url, link: `/${props.post.slug}`, }); }; Los corchetes en slug.txt.ts indican a Astro que esta es una ruta dinámica. La función getStaticPaths es necesaria para la generación estática: le dice a Astro qué páginas debe crear durante el proceso de construcción (una por cada post). Luego, la función GET recibe los datos del post correspondiente a través de props y llama al generador llmsPost. ¡Hemos terminado con el código! Ahora solo queda un último paso para asegurar que las IAs puedan encontrar nuestro nuevo llms.txt. #### 5. Paso 3: Asegurando la Visibilidad Aunque llms.txt es un estándar que se espera encontrar en la raíz del sitio, es una buena práctica anunciarlo en el HTML de tu página. Esto sigue las convenciones web y ayuda a la detección. Añade la siguiente etiqueta a tu layout base o a tu componente de cabecera (), junto a otras etiquetas similares como las del feed RSS o el favicon. html Con este último paso, has completado la implementación. Ahora puedes verificar que todo funcione correctamente. #### ¡Lo Lograste! Verificación y Próximos Pasos ¡Felicidades por completar la implementación! Si has seguido todos los pasos, tu sitio desplegado ahora debería tener tres nuevos endpoints funcionando: * /llms.txt: Un índice con la lista de todos tus posts y sus descripciones. * /llms-full.txt: Un único archivo de texto con el contenido completo de todos tus posts, separados por reglas horizontales. * /llms/el-slug-de-tu-post.txt: Un archivo de texto individual para cada uno de tus posts. Te animamos a visitar estas URLs en tu propio sitio para ver el resultado. A partir de ahora, los agentes de IA pueden obtener el

contenido limpio de tu blog en formato Markdown sin ningún esfuerzo. #### Consideraciones Adicionales ##### ¿Por Qué no Usar una Librería? Existen integraciones de Astro para llms.txt, pero el enfoque manual que hemos seguido ofrece varias ventajas: * **Control sobre el contenido** : Tú decides exactamente qué posts y páginas se incluyen. * **Endpoints por post** : Las librerías a menudo no generan los endpoints individuales, que son cruciales para consultas enfocadas. * **Control del formato** : Tienes control total sobre cómo se formatea la salida. * **Sin dependencias** : Evitas añadir otro paquete a tu proyecto. Esta implementación son ~150 líneas de TypeScript. Controlas exactamente qué se incluye. Entiendes cada línea. Para algo tan simple, el enfoque manual gana. ##### Limitaciones a Tener en Cuenta Esta implementación está diseñada para blogs que usan **colecciones de contenido de Astro con cuerpos en Markdown o MDX** . Funciona leyendo la propiedad post.body, que contiene el texto plano. No funcionará directamente para páginas construidas puramente con componentes (.astro, React, Svelte, etc.), ya que estas no tienen un cuerpo de Markdown para extraer. Para la mayoría de los blogs, las colecciones de contenido son la opción ideal de todos modos. #### Bonus: Un Icono SVG para tu Sitio El estándar llms.txt tiene un logo distintivo compuesto por cuatro cuadrados redondeados. Aquí tienes una versión SVG que puedes usar en tu sitio web. html Notas de diseño: * **viewBox="-4 -4 32 32"** : Añade un pequeño padding para que se alinee visualmente mejor con otros iconos basados en trazos. * **fill="currentColor"** : Permite que el icono herede el color del texto de su contenedor, adaptándose a cualquier tema (claro/oscuro). * **opacity variable** : Los diferentes niveles de opacidad dan una sensación de profundidad sin necesidad de usar múltiples colores. * **En Astro:** puedes envolverlo en un componente para poder pasarle una prop size y reutilizarlo fácilmente en todo tu sitio.