EDICIÓN #008

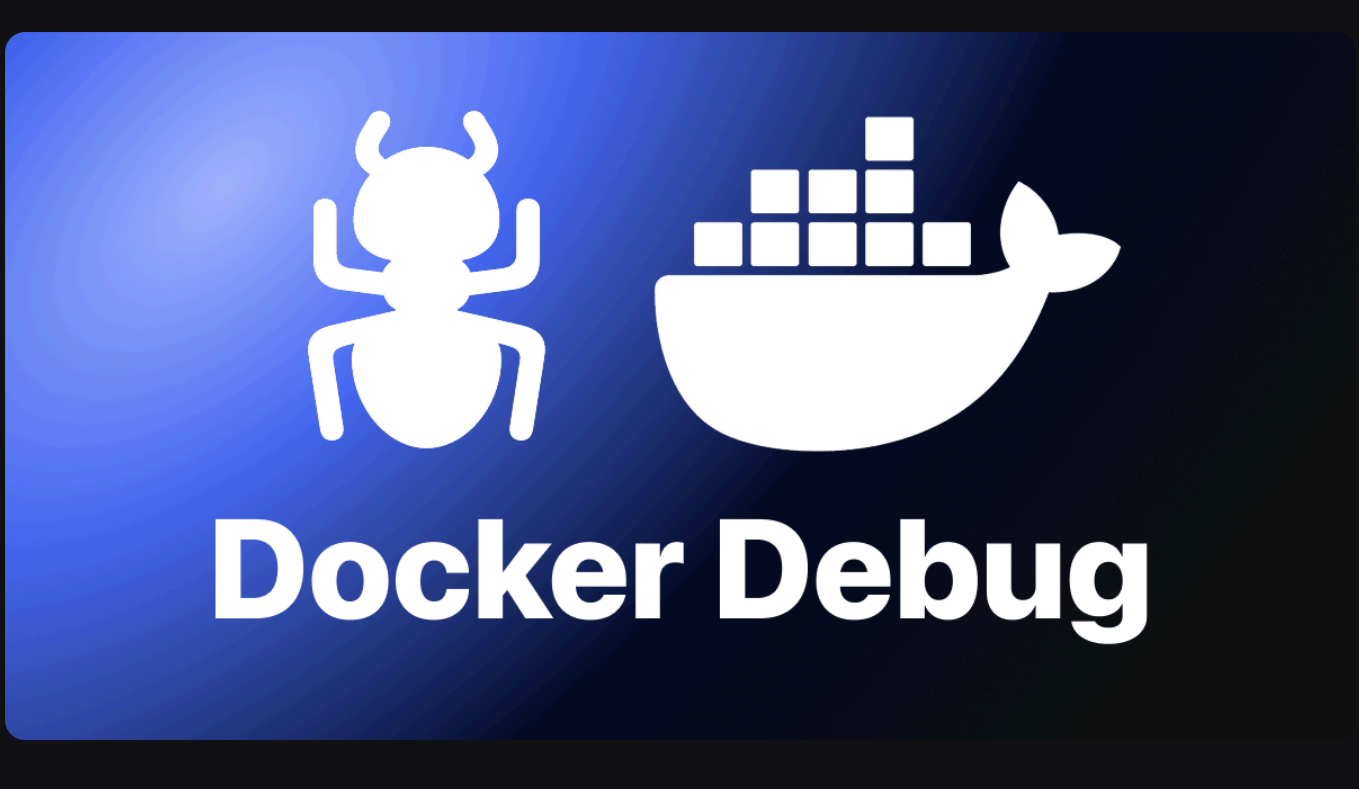
Docker estrena modo debug: Así puedes inspeccionar tus builds desde VS Code

12 de Enero de 2026

Docker estrena modo debug: Así puedes inspeccionar tus builds desde VS Code

Te enseño la nueva función que acaba de lanzar Docker que te deja pausar builds, ver variables y explorar el sistema de ficheros desde VS Code.

19 November 2025



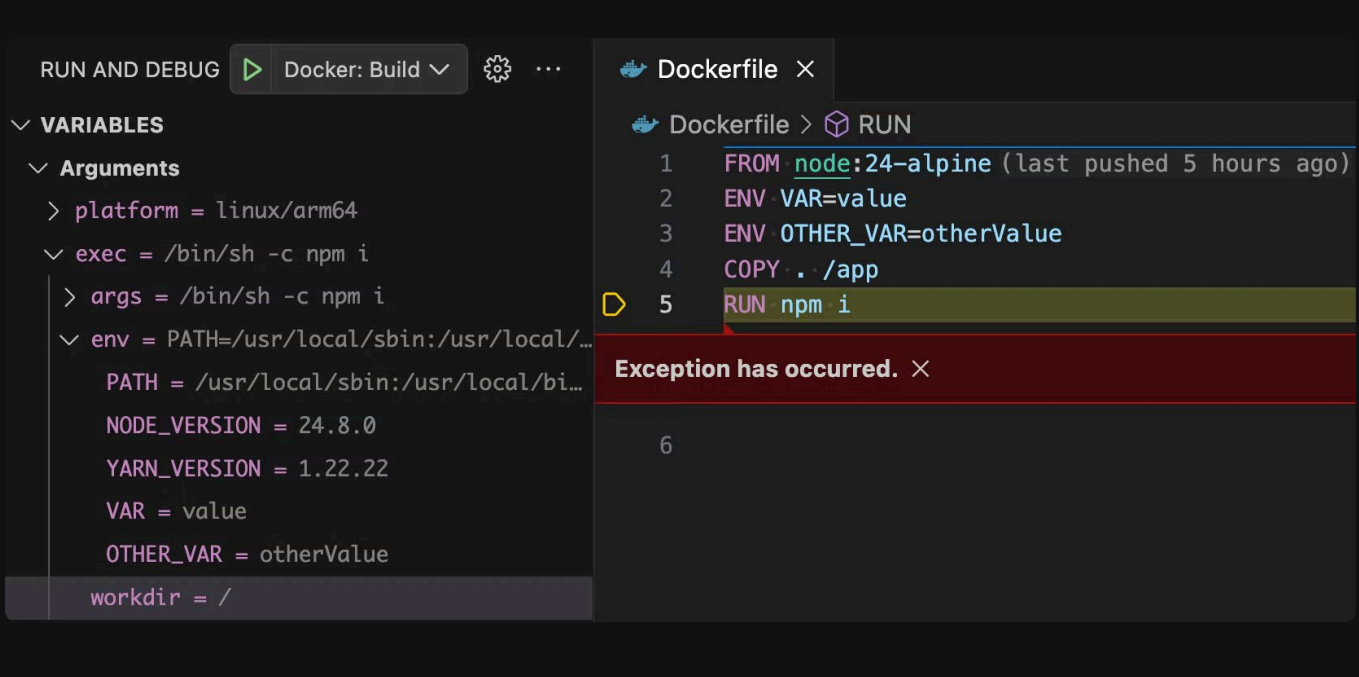
Contenido

- Debugging de imágenes Docker
- Guía rápida: instalar y configurar Build Debugging
- Debug de un Dockerfile en VS Code paso a paso
  - Cómo inspeccionar variables durante un build Docke …
  - Cómo inspeccionar los ficheros de un Dockerfile en construcción
  - Ejecuta comandos en tiempo real dentro de la imagen Docker

Docker es una de esas herramientas que uso casi a diario y me encanta: rápida, predecible y perfecta para aislar entornos. Peeero siempre ha habido algo que era un dolor de cabeza: poder debuggear imágenes.

Antes, si una build fallaba, tocaba meter *echos* en los RUN, añadir sleep o abrir shells en imágenes intermedias… puro ensayo y error.

La semana pasada estuve probando lo nuevo de Docker: Build Debugging. Va integrado en Buildx y en VS Code, y te permite depurar un Dockerfile paso a paso: poner breakpoints, mirar variables, explorar el sistema de ficheros e incluso abrir una shell interactiva dentro de la imagen mientras se construye.



Ejemplo de Docker Debug en VS Code

Debugging de imágenes Docker: cómo se hacía hasta ahora

Hasta ahora, si querías "debuggear" un build de Docker tenías que improvisar.

Podías usar `--progress=plain` para ver logs más detallados, o `--no-cache` para forzar la reconstrucción completa, pero nada de eso te decía en qué estado real estaba la imagen a mitad del proceso.

Otra opción era usar `docker run -it <id_de_capa> /bin/sh` para entrar a una capa intermedia, inspeccionar archivos o probar comandos, pero era muy manual…

Hasta ahora **no había forma de pausar una build y mirar dentro**. El Dockerfile se ejecutaba como una caja negra y, si algo fallaba, te tocaba rehacer el build una y otra vez hasta acertar.

Guía rápida: instalar y configurar Build Debugging en Docker

Para usar este nuevo modo de depuración necesitas tres cosas:

- **Docker Desktop 27.3 o superior**, ya que es la primera versión que incluye soporte nativo para el *debug adapter* de Buildx.
- **Buildx 0.29.x o superior** (puedes comprobarlo con `docker buildx version`).

  ```
  alvaro@Mac-mini-de-Alvaro demoDocker % docker buildx version
  github.com/docker/buildx v0.29.1-desktop.1 28f6246ff24e2c05095e8741e48c48dcb2d3b4bc
  ```

  docker buildx version - v0.29.1

- Y, por supuesto, la extensión Docker DX en Visual Studio Code.

  Esta última es importante: **Docker DX** es la extensión oficial mantenida por Docker.

  La otra que probablemente tengas instalada, "Docker" o "Containers", es una extensión gestionada por Microsoft que sirve para gestionar contenedores y redes, y sigue siendo muy útil, pero no trae el modo de depuración del build.

Con todo esto listo, solo tienes que crear este fichero de config en VS Code. En tu proyecto, crea la carpeta `.vscode` (si no existe) y dentro un fichero llamado `launch.json` con este contenido:

```
{//UseIntelliSensetolearnaboutpossibleattributes.//Hovertoviewdescriptionsofexisti
```

Este bloque le dice a VS Code que el objetivo a depurar es un Dockerfile. A partir de ahí, el propio editor se conecta al proceso de build usando el protocolo de depuración (Debug Adapter Protocol) y puede detener la ejecución en las líneas donde pongas puntos de interrupción.

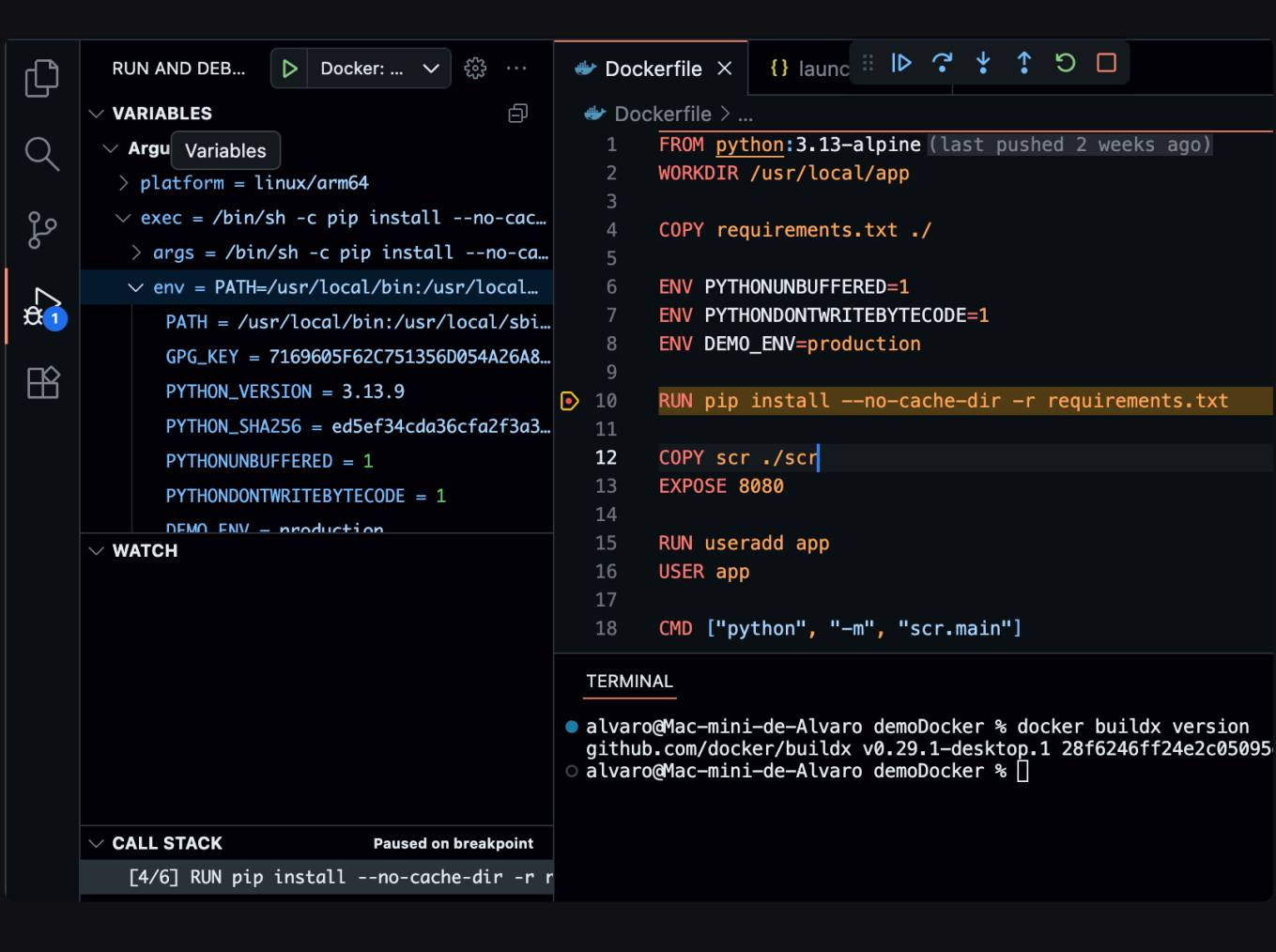
Debug de un Dockerfile en VS Code paso a paso

Voy a usar como ejemplo este Dockerfile simplificado, bastante típico para una app Python con Flask básica:

```
FROM python:3.13-alpine
WORKDIR /usr/local/app

COPY requirements.txt ./

ENV PYTHONUNBUFFERED=1
ENV PYTHONDONTWRITEBYTECODE=1
ENV DEMO_ENV=production

RUN pip install --no-cache-dir -r requirements.txt

COPY scr ./scr
EXPOSE 8080

RUN useradd app
USER app

CMD ["python", "-m", "scr.main"]
```

Lo primero que harás será abrir este Dockerfile en VS Code y colocar un **breakpoint** en alguna instrucción, por ejemplo justo en el `RUN pip install --no-cache-dir -r requirements.txt`. Esto funciona igual de bien si estás trabajando con multi-stage builds, donde cada etapa puede depurarse de forma independiente para verificar que todo se copia e instala correctamente antes de pasar a la siguiente fase.

Luego, abre la vista de "Run and Debug", selecciona "Docker: Build" y pulsa **F5**.



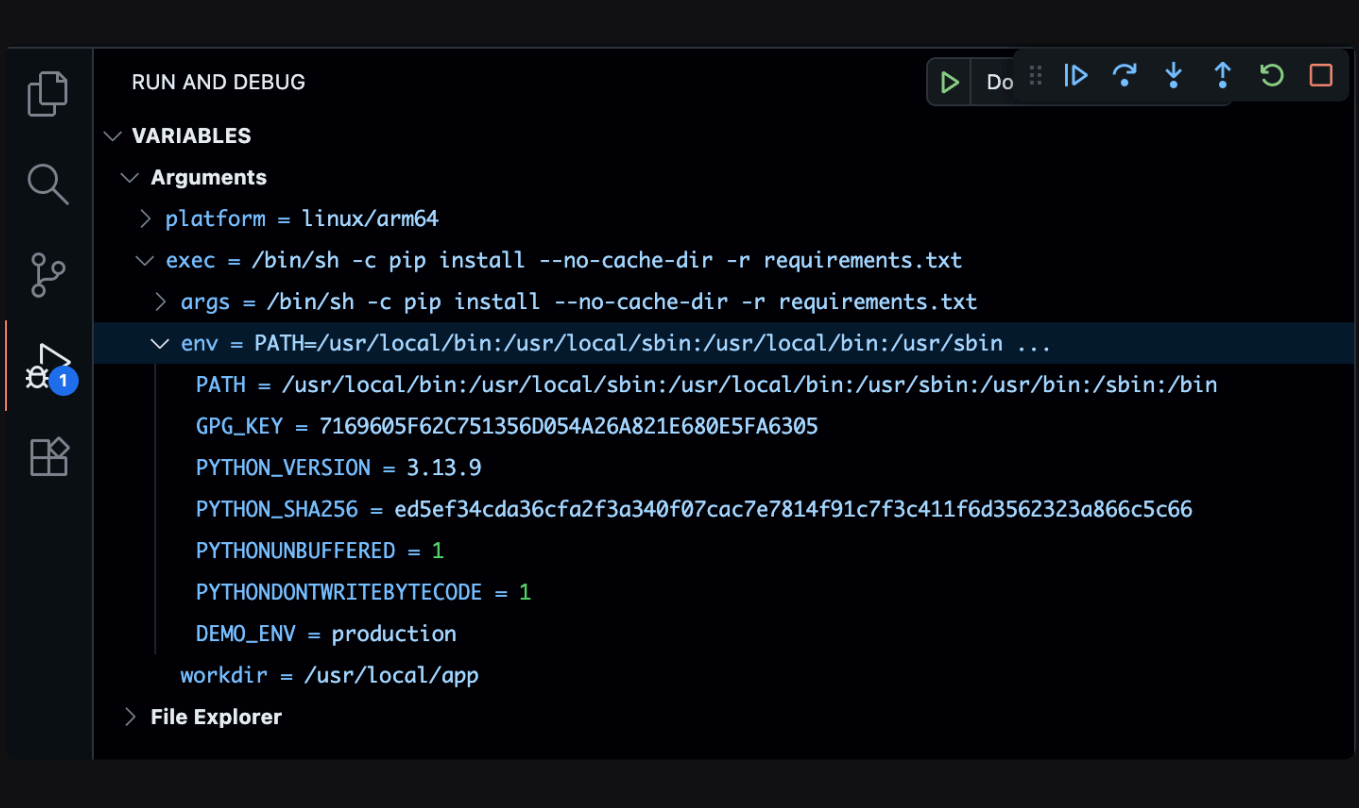
Ejemplo de Docker Debug con la extensión Docker DX en VS Code

El build se ejecutará y se detendrá justo en esa línea. A partir de ahí puedes usar tres funciones clave: inspeccionar variables, explorar archivos y abrir una sesión interactiva.

Cómo inspeccionar variables durante un build Docker

Cuando la build se pausa, verás en el panel lateral de VS Code un **listado con todas las variables disponibles** en ese momento: las que hayas definido tú con ENV o ARG, y las que provengan de la imagen base (PATH, PYTHON_VERSION, etc).

Por ejemplo, en el ejemplo anterior, verás que `DEMO_ENV` tiene el valor production, `WORKDIR` apunta a /usr/local/appy `PYTHONDONTWRITEBYTECODE` está activado.



Ver variables de un Dockerfile durante su construcción

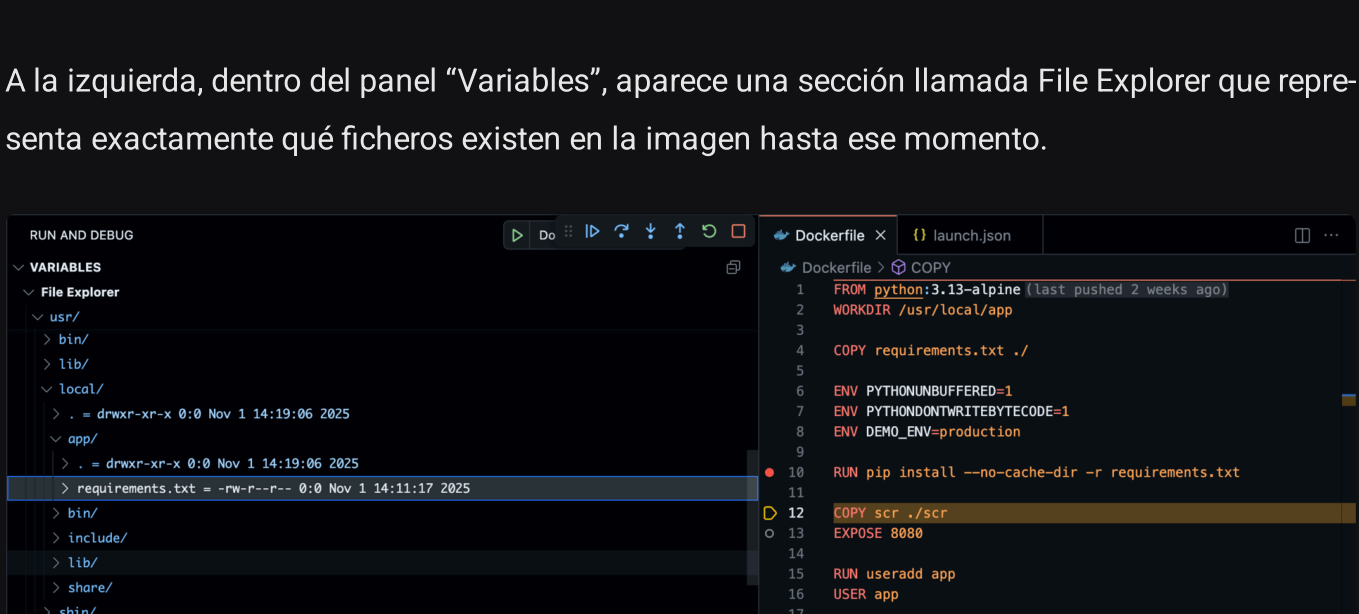
Hasta ahora esto era imposible sin construir toda la imagen y hacer un docker run env.

Ahora lo tienes directamente a un clic, antes de seguir al siguiente paso de la build.

File Explorer: Cómo inspeccionar los ficheros de un Dockerfile en construcción

Una de las cosas que más me ha gustado de este modo es poder **ver el sistema de ficheros intermedio de la imagen**.

A la izquierda, dentro del panel "Variables", aparece una sección llamada File Explorer que representa exactamente qué ficheros existen en la imagen hasta ese momento.



Explorando la imagen Docker en construcción

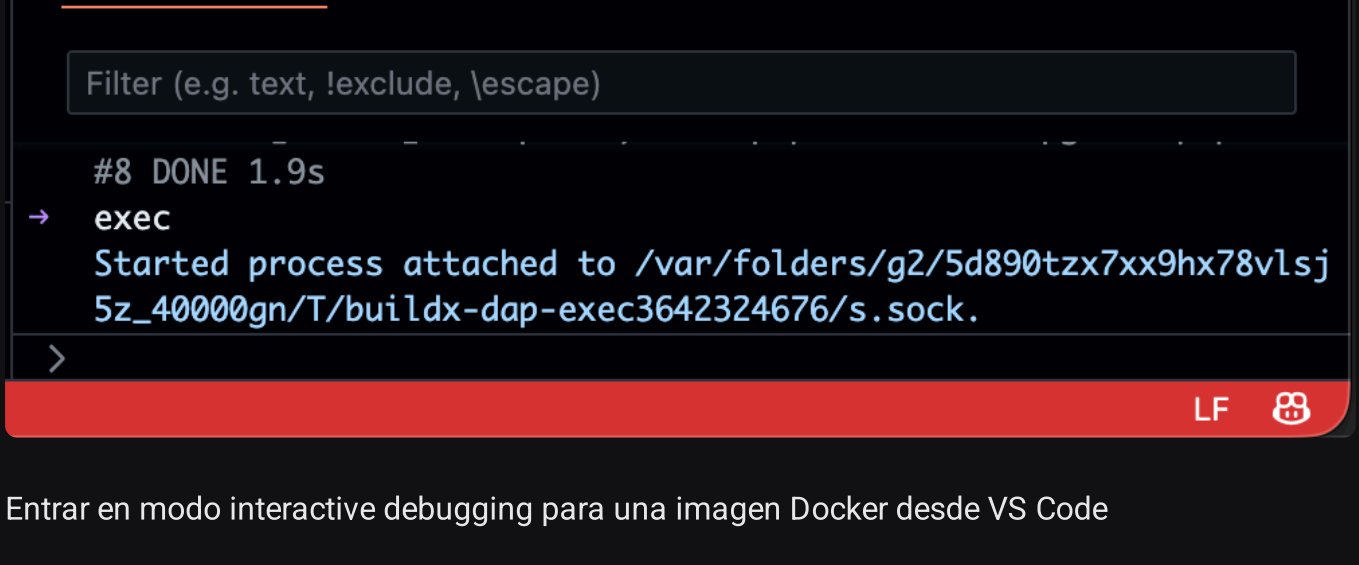
Puedes abrir carpetas y ver los archivos que se han copiado con COPY junto con sus permisos. Esto es increíblemente útil para detectar errores típicos como rutas incorrectas o archivos que no se copiaron por culpa del `.dockerignore`.

Interactive Debugging: Ejecutar comandos en tiempo real dentro de la imagen Docker

Y aquí viene lo mejor.

Mientras el build está pausado, puedes abrir una **shell interactiva dentro de la imagen en construcción**.

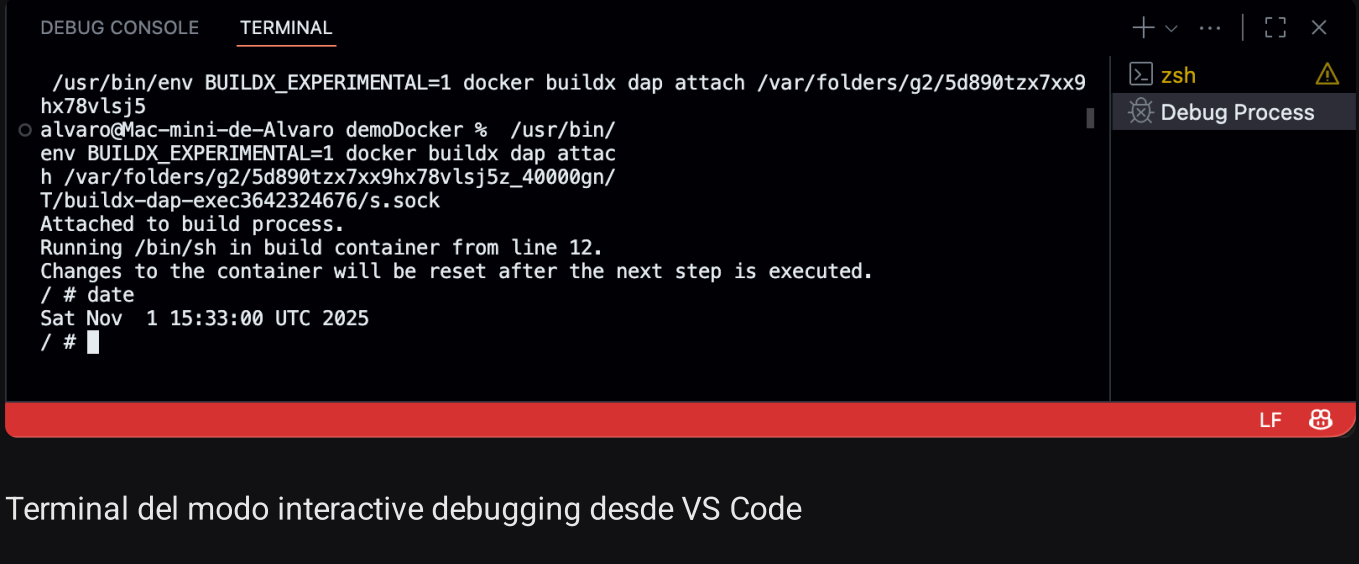
Solo tienes que ir a la consola de depuración (Debug Console), escribir exec y pulsar Enter.



Entrar en modo interactive debugging para una imagen Docker desde VS Code

VS Code abrirá una terminal nueva conectada a ese entorno exactamente en el punto donde se detuvo la build.

Desde ahí puedes ejecutar cualquier comando de Alpine o de Python, comprobar rutas, probar una instalación o ver qué dependencias están disponibles.



Terminal del modo interactive debugging desde VS Code

OJO, **no es persistente** (cuando avanzas al siguiente paso de build, se reinicia el entorno), pero es una forma excelente de experimentar y entender qué está pasando realmente dentro de tu imagen.

Por ejemplo, puedes escribir `ls -la /usr/local/app` y ver qué archivos hay después del COPY requirements.txt, o probar `python --version` para confirmar qué versión exacta se está usando.

Ya no necesitas comentar medio Dockerfile ni construir imágenes intermedias para entender un fallo. Todo ocurre en tu propio editor, con contexto y control total.

Docker ha hecho algo que muchos llevábamos años esperando: **convertir la build de una imagen en un proceso visible y depurable**. Y lo mejor es que no requiere nada raro: solo la extensión oficial Docker DX y tener Docker Desktop actualizado.

Lo llevo unos días usando y me parece una mejora enorme en la experiencia de desarrollo con contenedores. Si trabajas con Dockerfile grandes o builds multi-stage, dale una oportunidad.