Use Compose Watch

- Compose Watch versus bind mounts
- Configuration
  - Prerequisites
  - action
    - Sync
    - Rebuild
    - Sync + Restart
  - path and target
  - ignore
  - initial_sync
- Example 1
- Example 2
- Use watch
- Reference

{{< summary-bar feature_name="Compose file watch" >}}

{{% include "compose/watch.md" %}}

`watch` adheres to the following file path rules:

- All paths are relative to the project directory, apart from ignore file patterns
- Directories are watched recursively
- Glob patterns aren't supported
- Rules from `.dockerignore` apply
  - Use `ignore` option to define additional paths to be ignored (same syntax)
  - Temporary/backup files for common IDEs (Vim, Emacs, JetBrains, & more) are ignored automatically
  - `.git` directories are ignored automatically

You don't need to switch on `watch` for all services in a Compose project. In some instances, only part of the project, for example the Javascript frontend, might be suitable for automatic updates.

Compose Watch is designed to work with services built from local source code using the `build` attribute. It doesn't track changes for services that rely on pre-built images specified by the `image` attribute.

Compose Watch versus bind mounts

Compose supports sharing a host directory inside service containers. Watch mode does not replace this functionality but exists as a companion specifically suited to developing in containers.

More importantly, `watch` allows for greater granularity than is practical with a bind mount. Watch rules let you ignore specific files or entire directories within the watched tree.

For example, in a JavaScript project, ignoring the `node_modules/` directory has two benefits:

- Performance. File trees with many small files can cause a high I/O load in some configurations
- Multi-platform. Compiled artifacts cannot be shared if the host OS or architecture is different from the container

For example, in a Node.js project, it's not recommended to sync the `node_modules/` directory. Even though JavaScript is interpreted, `npm` packages can contain native code that is not portable across platforms.

Configuration

The `watch` attribute defines a list of rules that control automatic service updates based on local file changes.

Each rule requires a `path` pattern and `action` to take when a modification is detected. There are two possible actions for `watch` and depending on the `action`, additional fields might be accepted or required.

Watch mode can be used with many different languages and frameworks. The specific paths and rules will vary from project to project, but the concepts remain the same.

Prerequisites

In order to work properly, `watch` relies on common executables. Make sure your service image contains the following binaries:

- stat
- mkdir
- rmdir

`watch` also requires that the container's `USER` can write to the target path so it can update files. A common pattern is for initial content to be copied into the container using the `COPY` instruction in a Dockerfile. To ensure such files are owned by the configured user, use the `COPY --chown` flag:

```
# Run as a non-privileged user
FROM node:18
RUN useradd -ms /bin/sh -u 1001 app
USER app

# Install dependencies
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install

# Copy source files into application directory
COPY --chown=app:app . /app
```

`action`

Sync

If `action` is set to `sync`, Compose makes sure any changes made to files on your host automatically match with the corresponding files within the service container.

`sync` is ideal for frameworks that support "Hot Reload" or equivalent functionality.

More generally, `sync` rules can be used in place of bind mounts for many development use cases.

Rebuild

If `action` is set to `rebuild`, Compose automatically builds a new image with BuildKit and replaces the running service container.

The behavior is the same as running `docker compose up --build <svc>`.

Rebuild is ideal for compiled languages or as a fallback for modifications to particular files that require a full image rebuild (e.g. `package.json`).

> When a service is rebuilt, the previous image version becomes dangling. By default, Compose removes these superseded dangling images after each rebuild to avoid accumulating unused layers. To keep them, use `docker compose watch --prune=false`.

Sync + Restart

If `action` is set to `sync+restart`, Compose synchronizes your changes with the service containers and restarts them.

`sync+restart` is ideal when the config file changes, and you don't need to rebuild the image but just restart the main process of the service containers. It will work well when you update a database configuration or your `nginx.conf` file, for example.

> Optimize your `Dockerfile` for speedy incremental rebuilds with image layer caching and multi-stage builds.

`path` and `target`

The `target` field controls how the path is mapped into the container.

For `path: ./app/html` and a change to `./app/html/index.html`:

- `target: /app/html` -> `/app/html/index.html`
- `target: /app/static` -> `/app/static/index.html`
- `target: /assets` -> `/assets/index.html`

`ignore`

The `ignore` patterns are relative to the `path` defined in the current `watch` action, not to the project directory. In the following Example 1, the ignore path would be relative to the `./web` directory specified in the `path` attribute.

`initial_sync`

When using a `sync+x` action, the `initial_sync` attribute tells Compose to ensure files that are part of the defined `path` are up to date before starting a new watch session.

Example 1

This minimal example targets a Node.js application with the following structure:

```
myproject/
├── web/
│   ├── App.jsx
│   ├── index.js
│   └── node_modules/
├── Dockerfile
├── compose.yaml
└── package.json
```

```
services:
  web:
    build: .
    command: npm start
    develop:
      watch:
        - action: sync
          path: ./web
          target: /src/web
          initial_sync: true
          ignore:
            - node_modules/
        - action: rebuild
          path: package.json
```

In this example, when running `docker compose up --watch`, a container for the `web` service is launched using an image built from the `Dockerfile` in the project's root. The `web` service runs `npm start` for its command, which then launches a development version of the application with Hot Module Reload enabled in the bundler (Webpack, Vite, Turbopack, etc).

After the service is up, the watch mode starts monitoring the target directories and files. Then, whenever a source file in the `web/` directory is changed, Compose syncs the file to the corresponding location under `/src/web` inside the container. For example, `./web/App.jsx` is copied to `/src/web/App.jsx`.

Once copied, the bundler updates the running application without a restart.

And in this case, the `ignore` rule would apply to `myproject/web/node_modules/`, not `myproject/node_modules/`.

Unlike source code files, adding a new dependency can't be done on-the-fly, so whenever `package.json` is changed, Compose rebuilds the image and recreates the `web` service container.

This pattern can be followed for many languages and frameworks, such as Python with Flask: Python source files can be synced while a change to `requirements.txt` should trigger a rebuild.

Example 2

Adapting the previous example to demonstrate `sync+restart`:

```
services:
  web:
    build: .
    command: npm start
    develop:
      watch:
        - action: sync
          path: ./web
          target: /app/web
          ignore:
            - node_modules/
        - action: sync+restart
          path: ./proxy/nginx.conf
          target: /etc/nginx/conf.d/default.conf

  backend:
    build:
      context: backend
      target: builder
```

This setup demonstrates how to use the `sync+restart` action in Docker Compose to efficiently develop and test a Node.js application with a frontend web server and backend service. The configuration ensures that changes to the application code and configuration files are quickly synchronized and applied, with the `web` service restarting as needed to reflect the changes.

Use `watch`

{{% include "compose/configure-watch.md" %}}

> Watch can also be used with the dedicated `docker compose watch` command if you don't want to get the application logs mixed with the (re)build logs and filesystem sync events.

> Check out `dockersamples/avatars`, or local setup for Docker docs for a demonstration of Compose `watch`.

Reference

- Compose Develop Specification

□□□ #2
□ □□□ □□□ 2026-10-31 19:07:58 UTC
□ □□□ □□□ 2026-10-31 19:08:00 UTC