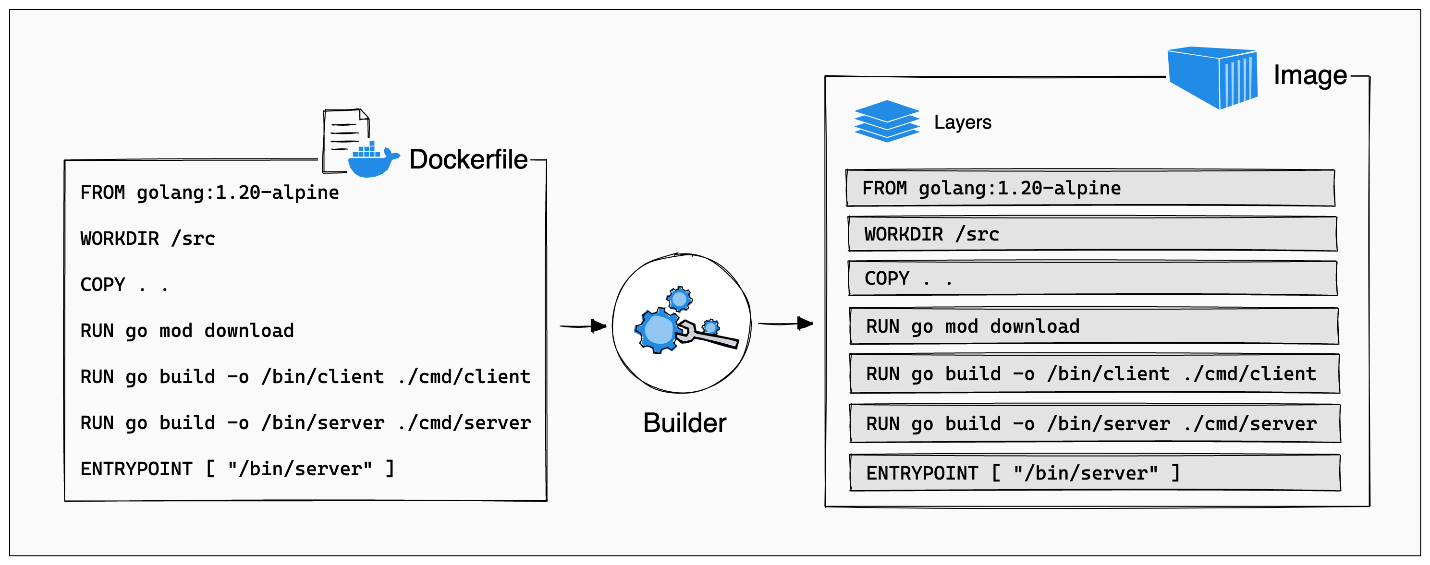
**Layers**

The order of Dockerfile instructions matters. A Docker build consists of a series of ordered build instructions. Each instruction in a Dockerfile roughly translates to an image layer. The following diagram illustrates how a Dockerfile translates into a stack of layers in a container image.

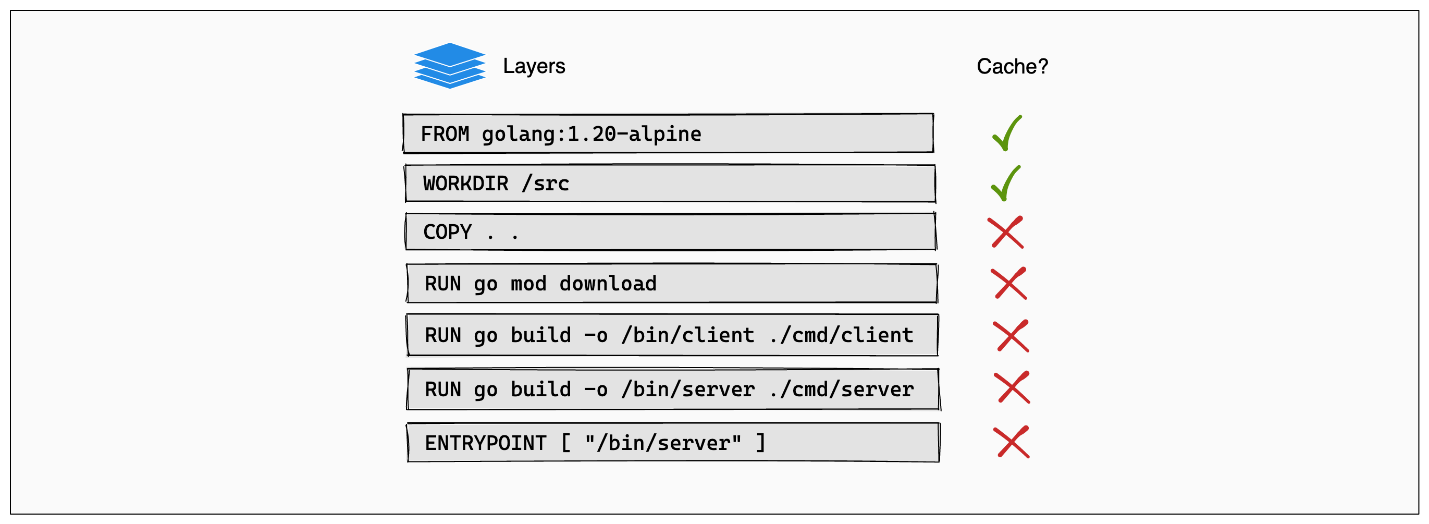


So A/c to above image you can see that how each instruction in a Docker file is an individual layer in Docker Image.

[**Cached layers**](https://docs.docker.com/build/guide/layers/#cached-layers)

When you run a build, the builder attempts to reuse layers from earlier builds. If a layer of an image is unchanged, then the builder picks it up from the build cache. If a layer has changed since the last build, that layer, and all layers that follow, must be rebuilt.

The Dockerfile from the previous section copies all project files to the container (COPY . .) and then downloads application dependencies in the following step (RUN go mod download). If you were to change any of the project files, then that would invalidate the cache for the COPY layer. It also invalidates the cache for all of the layers that follow.



**First understanding**

Now first toh yeh remember krein kay Docker caching layers ka use tab hota hai jab aap **second time** ya uske baad image rebuild karte hain. Pehli baar jab image build hoti hai, to Docker sabhi layers ko naya create karta hai, kyunki uss waqt koi cache available nahi hota.

Jab aap dubara ya further image rebuild karte hain, tab Docker pehle se existing cache ko check karta hai. Agar koi layer unchanged hai, to Docker us layer ka cache use karta hai, aur uss layer ko dubara build nahi karta. Agar koi layer change hui hai, to uss layer ke baad wali sari layers ko Docker dubara build karega.

Iska fayda ye hota hai ki agar aapke project me chhoti si change hui hai, to poori image ko dubara build karne ki zaroorat nahi padti, jo time aur resources dono bachaata hai.

**Second understanding**

Docker Image may jo layers create hoti hai wo on the top of previous layer create hoti hai like a Stack. Which means kay like A/c to above picture agar **COPY** layer invalidate/rebuild hogi toh uskay nichay jitni be layers hain wo dubara say rebuild hongi like in above picture there are some **RUN** **command** and **ENTRYPOINT** command.

**Now understanding of above picture**

Ab above image mabi yehi ho rha hai kay ab pehlay **COPY** ki command hai then jo modules wagera hain wo install kr rhay hain toh usmay yeh hua kay now imagine kay ek dafa build create hogyi sb project ki file copy hhogyi and using RUN saray modules be install hogye , but after that you have made a little change to your code so again jab Image rebuild hui toh Docker jab current files ko previous image file say compare kia toh usme srf file ma change hui thi toh iska mtlb kay jo COPY wali layer hai wo invalidate hogyi or ab wo new create hogi toh ab jo baki nichay wali layers hain after COPY wo be new create hongi but ab jo humaray modules hain unme toh kuch change nhi hua but phr be wo new create hongay which is not good therefore we have to write instruction in Dokcer file in a proper order.

[**Update the instruction order**](https://docs.docker.com/build/guide/layers/#update-the-instruction-order)

You can avoid this redundancy by reordering the instructions in the Dockerfile. Change the order of the instructions so that downloading and installing dependencies occur before the source code is copied over to the container. In that way, the builder can reuse the "dependencies" layer from the cache, even when you make changes to your source code.

Go uses two files, called go.mod and go.sum, to track dependencies for a project. These files are to Go, what package.json and package-lock.json are to JavaScript. For Go to know which dependencies to download, you need to copy the go.mod and go.sum files to the container. Add another COPY instruction before RUN go mod download, this time copying only the go.mod and go.sum files.

# syntax=docker/dockerfile:1

FROM golang:1.21-alpine

WORKDIR /src

COPY go.mod go.sum .

RUN go mod download

COPY . .

RUN go build -o /bin/client ./cmd/client

RUN go build -o /bin/server ./cmd/server

ENTRYPOINT [ "/bin/server" ]

Now here we have updated the Dockerfile mtlb kay ab pehlay hum jo dependencies wali file hoti hain unko COPY kr rhay hain then phr dependencies ko install kr rhay hain or then phr apnay jo project ki files hain unko coy kr rhay hain , Takay ab jab Docker Image rebuild ho toh jo dependencies/modules already hain wo dubara say reinstall na ho blka previous build cache ko use krein or srf jo project file may changes hon wohi new create hon.

in the following code snippet mujah yeh nhi smjh ayi kay jo package\*.json pehlay humnay COPY krli ab jab next again COPY ki command may sari project ki files copy hongi toh usme jo package\*.json hai wo again copy hogi ya jo previous hai wo use hogi ?

# syntax=docker/dockerfile:1

FROM node

WORKDIR /app

COPY package.json yarn.lock . # Copy package management files

RUN npm install # Install dependencies

COPY . . # Copy over project files

RUN npm build # Run build

Toh iska answer yeh haka jab second COPY ki command execute hogi toh **/app** WORKDIR may jo pehlay package\*.json hain wo overwrite hojayegi which means kay jo recent COPY command may package\*.json hongi wo ajayegi previous wali ki jagah.

Q-Now agar hum jo pehlay package\*.json copy kr rhy hain wo na krein bs direct run npm install krdein pr phr after that jab sb files ko COPY kreingay toh usme package\*.json be ajayegi. is it ok or not ?

* Agar aap package\*.json files ko pehle copy nahi karte aur seedha npm install chalane ki koshish karte hain, toh Docker ko package.json aur yarn.lock files nahi milenge, aur npm install command fail ho jaayegi. Iska reason yeh hai ki npm install ko dependencies install karne ke liye package.json aur yarn.lock files ki zaroorat hoti hai, jo us waqt container ke andar available nahi hongi agar aap pehle COPY command nahi chalatay.

Or agar optimization kay lehaz say dekhain kay package\*.json 2 bar copy hogi toh uska be msla nhi hai kiu kay package.json just blueprint hai. Its light weight so it does not affect optimization.

For more info check: <https://docs.docker.com/build/cache/>