**When server showing no space but still there is space**

Reducing the size of Docker images can significantly improve build times, reduce storage costs, and speed up deployment. Here are several strategies you can employ:

### 1. Choose a Minimal Base Image

- Use a lightweight base image such as `alpine`, `busybox`, or `scratch` instead of larger images like `ubuntu` or `debian`. For example:

```dockerfile

FROM alpine:latest

```

### 2. Multi-Stage Builds

- Use multi-stage builds to compile your application in a larger image, and then copy only the necessary artifacts to a smaller image:

```dockerfile

# Stage 1: Build the application

FROM node:20 AS build

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build

# Stage 2: Create the final image

FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html

```

### 3. Minimize Layers

- Combine commands using `&&` to reduce the number of layers created in your image:

```dockerfile

RUN apt-get update && apt-get install -y \

package1 \

package2 \

&& apt-get clean \

&& rm -rf /var/lib/apt/lists/\*

```

### 4. Clean Up Temporary Files

- Always clean up temporary files and package caches after installing packages to avoid adding unnecessary data to the image:

```dockerfile

RUN apt-get update && apt-get install -y package \

&& rm -rf /var/lib/apt/lists/\*

```

### 5. Use `.dockerignore` File

- Create a `.dockerignore` file to exclude files and directories that are not necessary for your application from being added to the image:

```

node\_modules

npm-debug.log

.git

```

### 6. Optimize Dependencies

- Use production-only dependencies if your application supports it. For Node.js, you can run:

```dockerfile

RUN npm install --only=production

```

### 7. Use Specific Version Tags

- Specify version tags for base images to avoid pulling larger, untagged images unnecessarily:

```dockerfile

FROM node:20.16.0

```

### 8. Remove Unused Packages

- If your image includes development tools that are only needed during build, remove them in the same layer after installing necessary packages:

```dockerfile

RUN apt-get update && apt-get install -y build-essential \

&& make install \

&& apt-get remove --purge -y build-essential \

&& apt-get autoremove -y

```

### 9. Minimize Files and Resources

- Minimize the size of files included in your image. Remove documentation, tests, and other unnecessary files.

### 10. Use Docker Squash

- Use the Docker Squash option to combine all layers into a single layer after building your image. This can be done by using the `--squash` option when building your image (note that this is experimental):

```bash

docker build --squash -t myimage:latest .

```

### Summary

By applying these strategies, you can significantly reduce the size of your Docker images, leading to faster builds and deployments. Start by identifying the areas in your Dockerfile where these practices can be applied most effectively.