前言

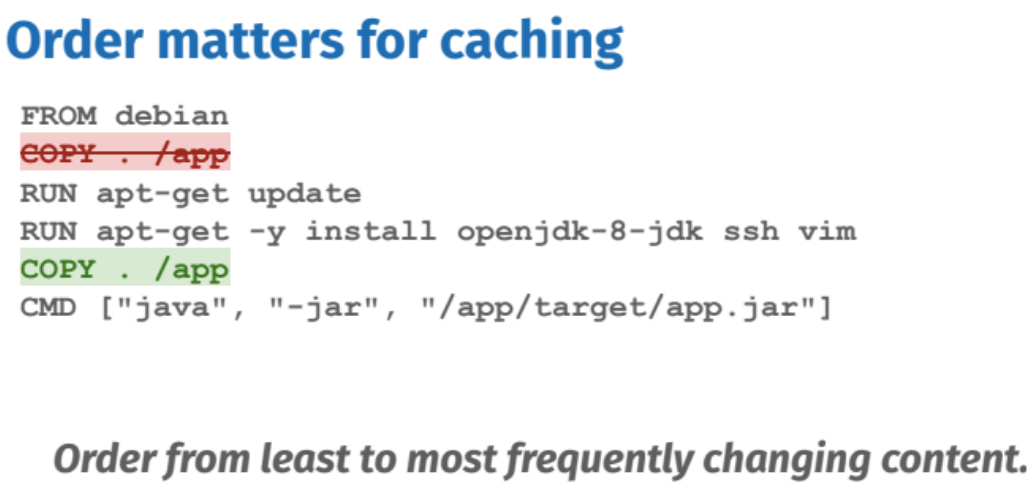
如今 GitHub 仓库中已经包含了成千上万的 Dockerfile，但并不是所有的 Dockerfile 都是高效的。本文将从五个方面来介绍 Dockerfile 的最佳实践，以此来帮助大家编写更优雅的 Dockerfile。如果你是 Docker 的初学者，恭喜你，这篇文章就是为你准备的。后面的系列将会更加深入，敬请期待！

> *本文使用一个基于 Maven 的 Java 项目作为示例，然后不断改进 Dockerfile 的写法，直到最后写出一个最优雅的 Dockerfile。中间的所有步骤都是为了说明某一方面的最佳实践。*

减少构建时间

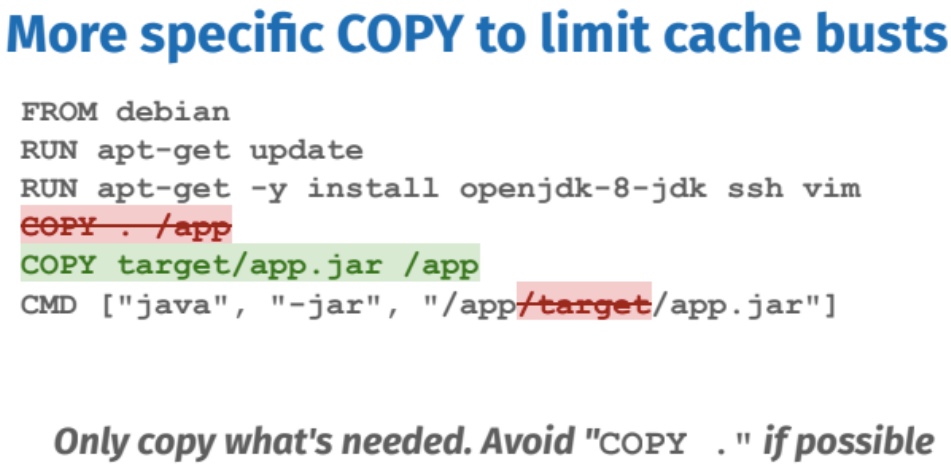
一个开发周期包括构建 Docker 镜像，更改代码，然后重新构建 Docker 镜像。在构建镜像的过程中，如果能够利用缓存，可以减少不必要的重复构建步骤。

构建顺序影响缓存的利用率



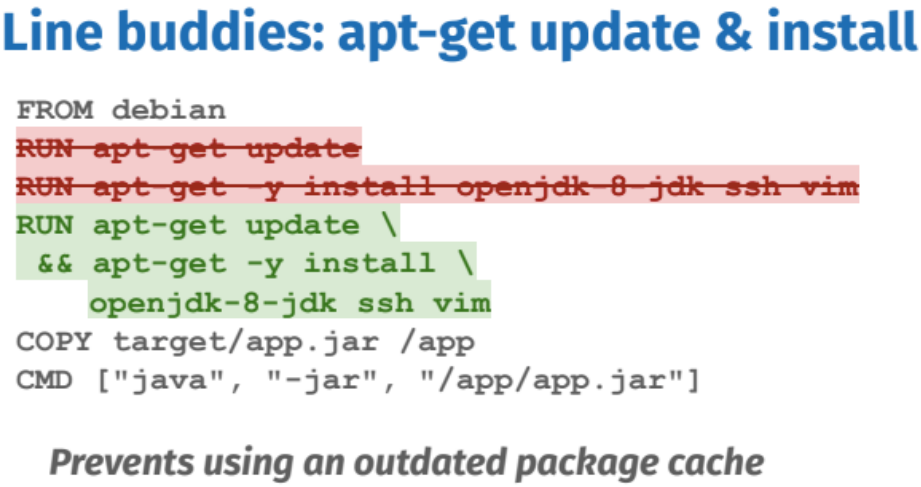
镜像的构建顺序很重要，当你向 Dockerfile 中添加文件，或者修改其中的某一行时，那一部分的缓存就会失效，该缓存的后续步骤都会中断，需要重新构建。**所以优化缓存的最佳方法是把不需要经常更改的行放到最前面，更改最频繁的行放到最后面。**

只拷贝需要的文件，防止缓存溢出



当拷贝文件到镜像中时，尽量只拷贝需要的文件，切忌使用 COPY . 指令拷贝整个目录。如果被拷贝的文件内容发生了更改，缓存就会被破坏。在上面的示例中，镜像中只需要构建好的 jar 包，因此只需要拷贝这个文件就行了，这样即使其他不相关的文件发生了更改也不会影响缓存。

最小化可缓存的执行层

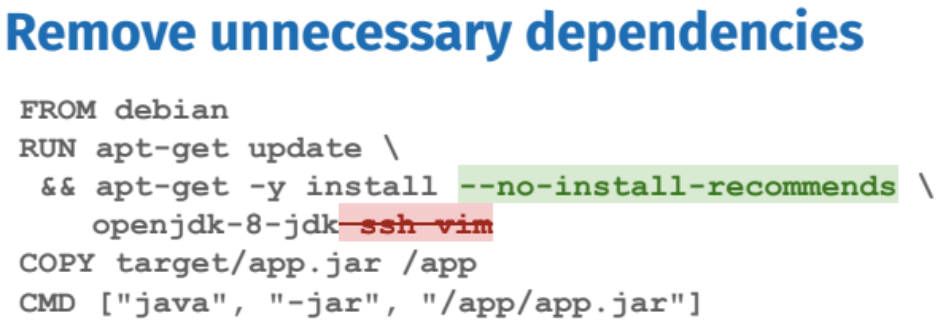


每一个 RUN 指令都会被看作是可缓存的执行单元。太多的 RUN 指令会增加镜像的层数，增大镜像体积，而将所有的命令都放到同一个 RUN 指令中又会破坏缓存，从而延缓开发周期。当使用包管理器安装软件时，一般都会先更新软件索引信息，然后再安装软件。推荐将更新索引和安装软件放在同一个 RUN 指令中，这样可以形成一个可缓存的执行单元，否则你可能会安装旧的软件包。

减小镜像体积

镜像的体积很重要，因为镜像越小，部署的速度更快，攻击范围越小。

删除不必要依赖



删除不必要的依赖，不要安装调试工具。如果实在需要调试工具，可以在容器运行之后再安装。某些包管理工具（如 apt）除了安装用户指定的包之外，还会安装推荐的包，这会无缘无故增加镜像的体积。apt 可以通过添加参数 `--no-install-recommends` 来确保不会安装不需要的依赖项。如果确实需要某些依赖项，请在后面手动添加。

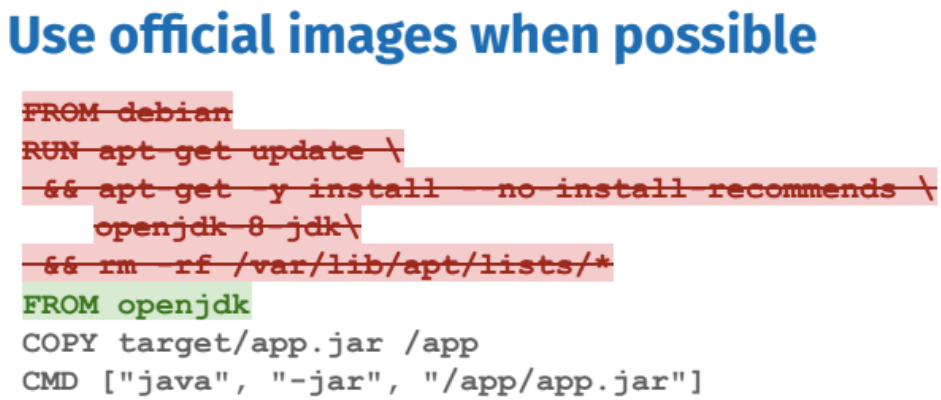
删除包管理工具的缓存

包管理工具会维护自己的缓存，这些缓存会保留在镜像文件中，推荐的处理方法是在每一个 RUN 指令的末尾删除缓存。如果你在下一条指令中删除缓存，不会减小镜像的体积。

当然了，还有其他更高级的方法可以用来减小镜像体积，如下文将会介绍的多阶段构建。接下来我们将探讨如何优化 Dockerfile 的可维护性、安全性和可重复性。

可维护性

尽量使用官方镜像

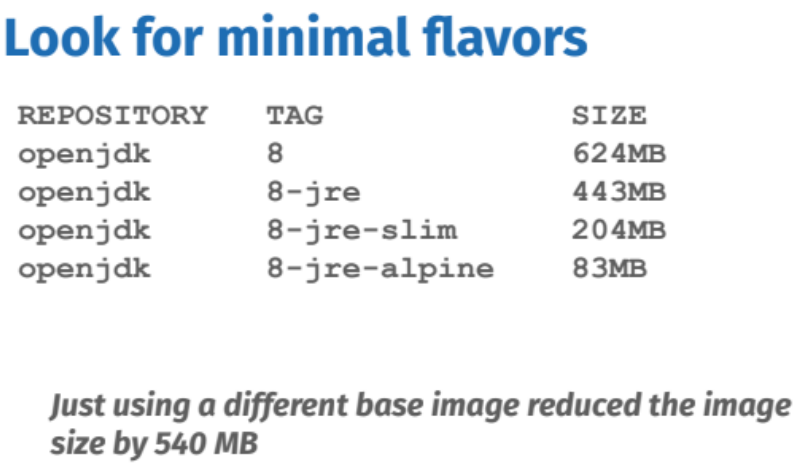


使用官方镜像可以节省大量的维护时间，因为官方镜像的所有安装步骤都使用了最佳实践。如果你有多个项目，可以共享这些镜像层，因为他们都可以使用相同的基础镜像。

使用更具体的标签

基础镜像尽量不要使用 `latest` 标签。虽然这很方便，但随着时间的推移，latest 镜像可能会发生重大变化。因此在 Dockerfile 中最好指定基础镜像的具体标签。我们使用 `openjdk` 作为示例，指定标签为 8。其他更多标签请查看官方仓库。

使用体积最小的基础镜像



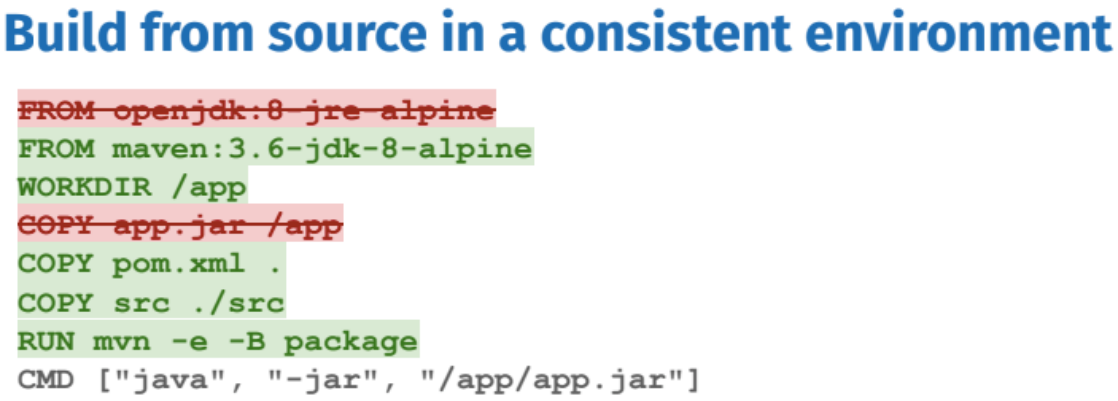
基础镜像的标签风格不同，镜像体积就会不同。`slim` 风格的镜像是基于 Debian 发行版制作的，而 `alpine` 风格的镜像是基于体积更小的 Alpine Linux 发行版制作的。其中一个明显的区别是：Debian 使用的是 GNU 项目所实现的 C 语言标准库，而 Alpine 使用的是 Musl C 标准库，它被设计用来替代 GNU C 标准库（glibc）的替代品，用于嵌入式操作系统和移动设备。因此使用 Alpine 在某些情况下会遇到兼容性问题。 以 openjdk 为例，`jre` 风格的镜像只包含 Java 运行时，不包含 `SDK`，这么做也可以大大减少镜像体积。

重复利用

到目前为止，我们一直都在假设你的 jar 包是在主机上构建的，这还不是理想方案，因为没有充分利用容器提供的一致性环境。例如，如果你的 Java 应用依赖于某一个特定的操作系统的库，就可能会出现问题，因为环境不一致（具体取决于构建 jar 包的机器）。

在一致的环境中从源代码构建

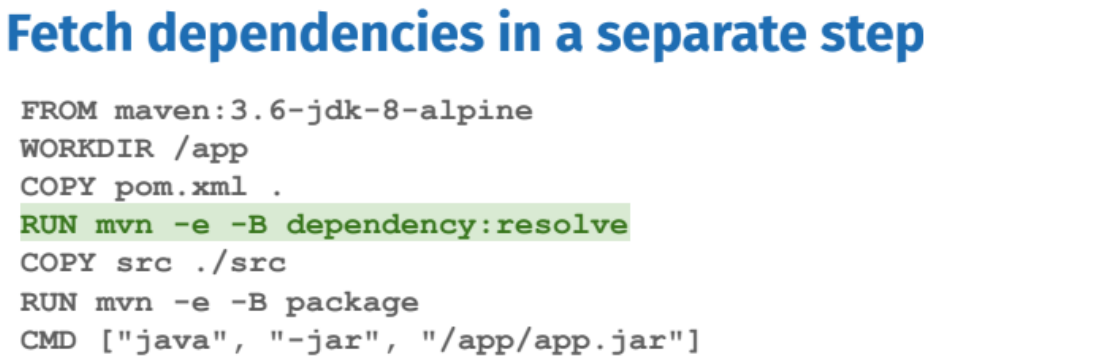
源代码是你构建 Docker 镜像的最终来源，Dockerfile 里面只提供了构建步骤。



首先应该确定构建应用所需的所有依赖，本文的示例 Java 应用很简单，只需要 `Maven` 和 `JDK`，所以基础镜像应该选择官方的体积最小的 maven 镜像，该镜像也包含了 JDK。如果你需要安装更多依赖，可以在 RUN 指令中添加。`pom.xml`文件和 `src` 文件夹需要被复制到镜像中，因为最后执行 `mvn package` 命令（`-e` 参数用来显示错误，`-B` 参数表示以非交互式的“批处理”模式运行）打包的时候会用到这些依赖文件。

虽然现在我们解决了环境不一致的问题，但还有另外一个问题：**每次代码更改之后，都要重新获取一遍 pom.xml 中描述的所有依赖项**。下面我们来解决这个问题。

在单独的步骤中获取依赖项



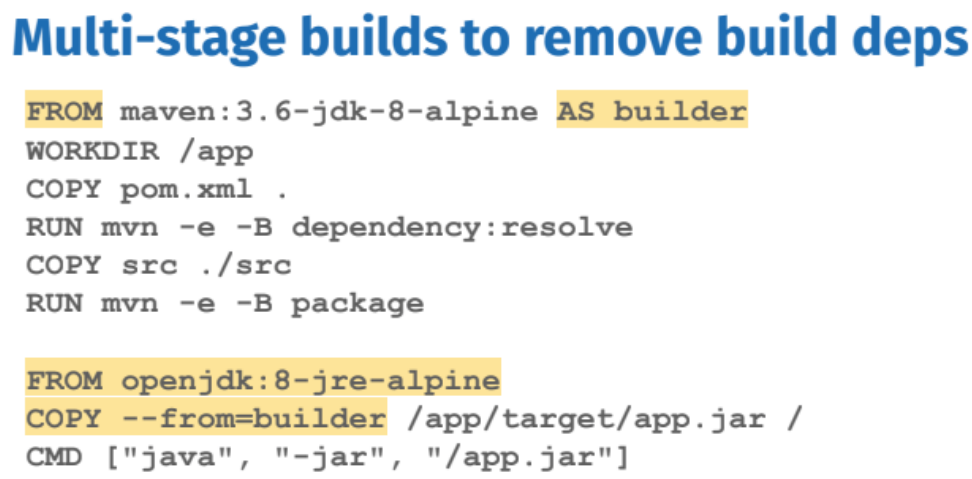
结合前面提到的缓存机制，我们可以让获取依赖项这一步变成可缓存单元，只要 pom.xml 文件的内容没有变化，无论代码如何更改，都不会破坏这一层的缓存。上图中两个 COPY 指令中间的 RUN 指令用来告诉 Maven 只获取依赖项。

现在又遇到了一个新问题：跟之前直接拷贝 jar 包相比，镜像体积变得更大了，因为它包含了很多运行应用时不需要的构建依赖项。

使用多阶段构建来删除构建时的依赖项

多阶段构建可以由多个 FROM 指令识别，每一个 FROM 语句表示一个新的构建阶段，阶段名称可以用 AS 参数指定。本例中指定第一阶段的名称为 `builder`，它可以被第二阶段直接引用。两个阶段环境一致，并且第一阶段包含所有构建依赖项。

第二阶段是构建最终镜像的最后阶段，它将包括应用运行时的所有必要条件，本例是基于 Alpine 的最小 JRE 镜像。上一个构建阶段虽然会有大量的缓存，但不会出现在第二阶段中。为了将构建好的 jar 包添加到最终的镜像中，可以使用 `COPY --from=STAGE_NAME` 指令，其中 STAGE_NAME 是上一构建阶段的名称。



多阶段构建是删除构建依赖的首选方案。

本文从在非一致性环境中构建体积较大的镜像开始优化，一直优化到在一致性环境中构建最小镜像，同时充分利用了缓存机制。下一篇文章将会介绍多阶段构建的更多其他用途。

附录

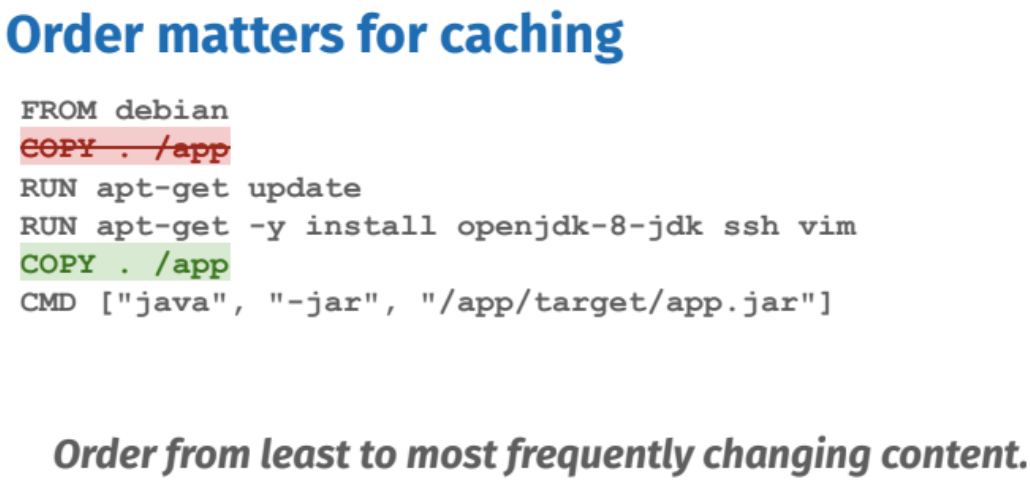
There are over one million Dockerfiles on GitHub today, but not all Dockerfiles are created equally. Efficiency is critical, and this blog series will cover five areas for Dockerfile best practices to help you write better Dockerfiles: incremental build time, image size, maintainability, security and repeatability. If you're just beginning with Docker, this first blog post is for you! The next posts in the series will be more advanced.

Important note*:* *the tips below follow the journey of ever-improving Dockerfiles for an example Java project based on Maven. The* last Dockerfile is thus the recommended Dockerfile, while all intermediate ones are there only to illustrate specific best practices.

Incremental build time

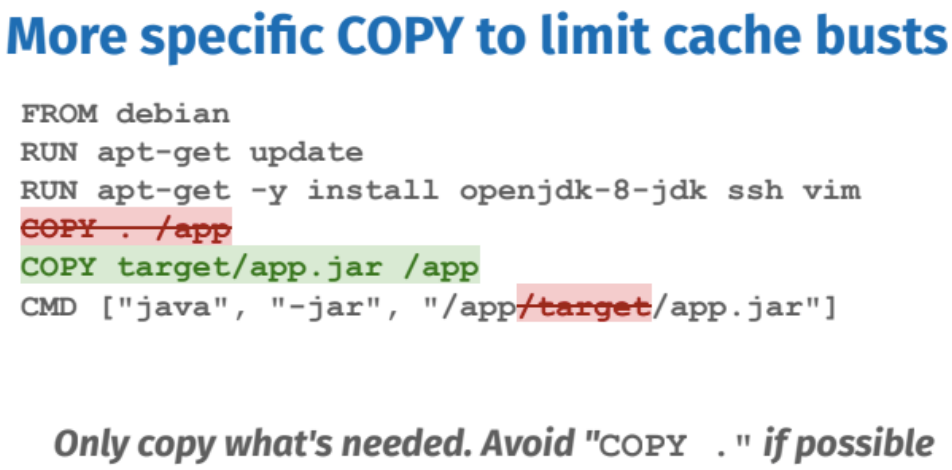
In a development cycle, when building a Docker image, making code changes, then rebuilding, it is important to leverage caching. Caching helps to avoid running build steps again when they don't need to.

Tip #1: Order matters for caching



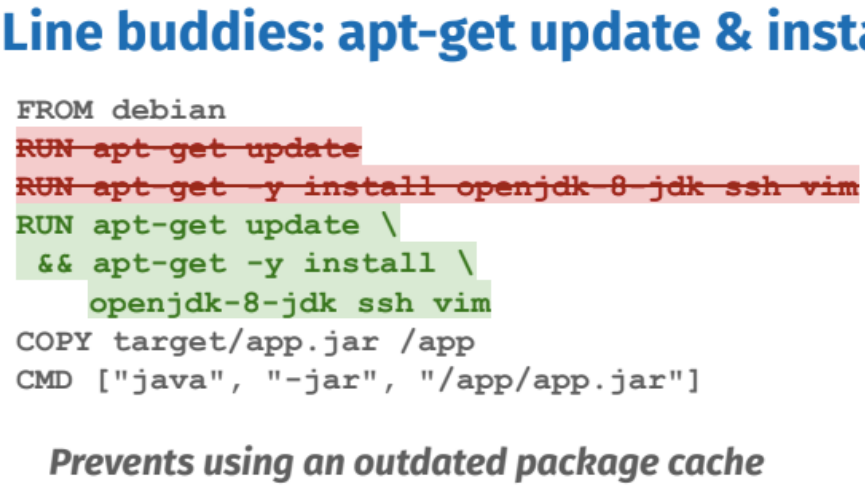
However, the order of the build steps (Dockerfile instructions) matters, because when a step's cache is invalidated by changing files or modifying lines in the Dockerfile, subsequent steps of their cache will break. Order your steps from least to most frequently changing steps to optimize caching.

Tip #2: More specific COPY to limit cache busts



Only copy what's needed. If possible, avoid "COPY ." When copying files into your image, make sure you are very specific about what you want to copy. Any changes to the files being copied will break the cache. In the example above, only the pre-built jar application is needed inside the image, so only copy that. That way unrelated file changes will not affect the cache.

Tip #3: Identify cacheable units such as apt-get update & install



Each RUN instruction can be seen as a cacheable unit of execution. Too many of them can be unnecessary, while chaining all commands into one RUN instruction can bust the cache easily, hurting the development cycle. When installing packages from package managers, you always want to update the index and install packages in the same RUN: they form together one cacheable unit. Otherwise you risk installing outdated packages.

Reduce Image size

Image size can be important because smaller images equal faster deployments and a smaller attack surface.

Tip #4: Remove unnecessary dependencies

Remove unnecessary dependencies and do not install debugging tools. If needed debugging tools can always be installed later. Certain package managers such as apt, automatically install packages that are recommended by the user-specified package, unnecessarily increasing the footprint. Apt has the –no-install-recommends flag which ensures that dependencies that were not actually needed are not installed. If they are needed, add them explicitly.

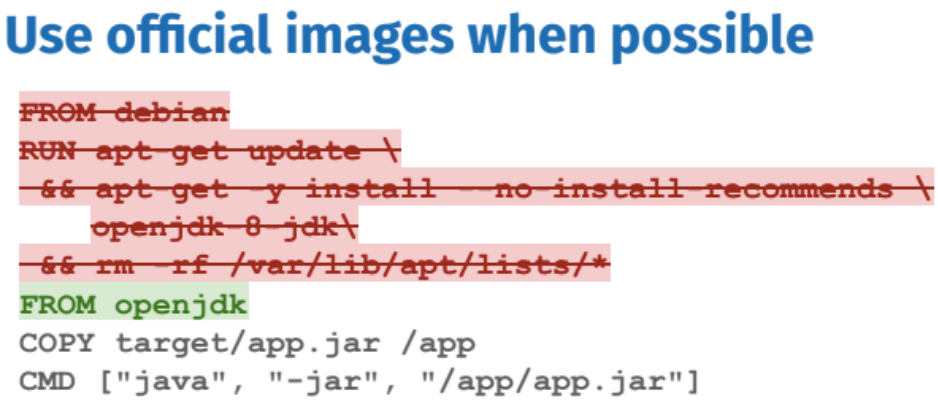
Tip #5: Remove package manager cache

Package managers maintain their own cache which may end up in the image. One way to deal with it is to remove the cache in the same RUN instruction that installed packages. Removing it in another RUN instruction would not reduce the image size.

There are further ways to reduce image size such as multi-stage builds which will be covered at the end of this blog post. The next set of best practices will look at how we can optimize for maintainability, security, and repeatability of the Dockerfile.

Maintainability

Tip #6: Use official images when possible



Official images can save a lot of time spent on maintenance because all the installation steps are done and best practices are applied. If you have multiple projects, they can share those layers because they use exactly the same base image.

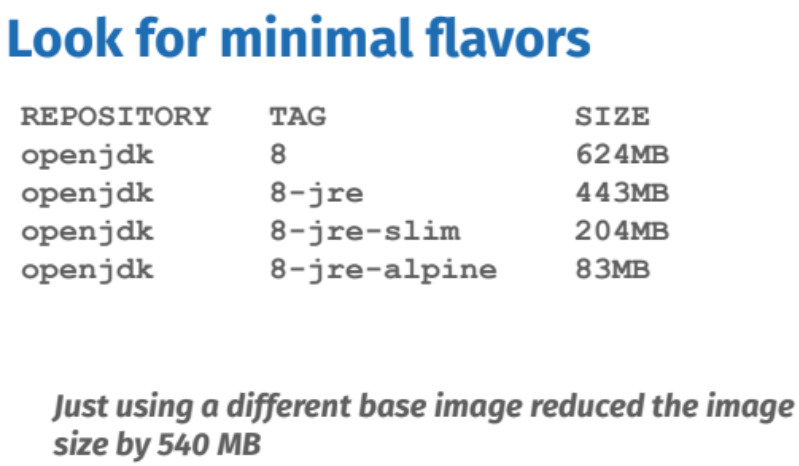
Tip #7: Use more specific tags

> ***The "latest" tag is a rolling tag. Be specific, to prevent unexpected changes in your base image.***

Do not use the latest tag. It has the convenience of always being available for official images on Docker Hub but there can be breaking changes over time. Depending on how far apart in time you rebuild the Dockerfile without cache, you may have failing builds.

Instead, use more specific tags for your base images. In this case, we're using openjdk. There are a lot more tags available so check out the Docker Hub documentation for that image which lists all the existing variants.

Tip #8: Look for minimal flavors



Some of those tags have minimal flavors which means they are even smaller images. The slim variant is based on a stripped down Debian, while the alpine variant is based on the even smaller Alpine Linux distribution image. A notable difference is that debian still uses GNU libc while alpine uses musl libc which, although much smaller, may in some cases cause compatibility issues. In the case of openjdk, the jre flavor only contains the java runtime, not the sdk; this also drastically reduces the image size.

Reproducibility

So far the Dockerfiles above have assumed that your jar artifact was built on the host. This is not ideal because you lose the benefits of the consistent environment provided by containers. For instance if your Java application depends on specific libraries it may introduce unwelcome inconsistencies depending on which computer the application is built.

Tip #9: Build from source in a consistent environment

The source code is the source of truth from which you want to build a Docker image. The Dockerfile is simply the blueprint.

Build from source in a consistent environment

```
FROM openjdk:8-jre-alpine
FROM maven:3.6-jdk-8-alpine
WORKDIR /app
COPY app.jar /app
COPY pom.xml .
COPY src ./src
RUN mvn -e -B package
CMD ["java", "-jar", "/app/app.jar"]
```

You should start by identifying all that's needed to build your application. Our simple Java application requires Maven and the JDK, so let's base our Dockerfile off of a specific minimal official maven image from Docker Hub, that includes the JDK. If you needed to install more dependencies, you could do so in a RUN step.

The pom.xml and src folders are copied in as they are needed for the final RUN step that produces the app.jar application with `mvn package`. (The -e flag is to show errors and -B to run in non-interactive aka "batch" mode).

We solved the inconsistent environment problem, but introduced another one: every time the code is changed, all the dependencies described in pom.xml are fetched. Hence the next tip.

Tip #10: Fetch dependencies in a separate step

Fetch dependencies in a separate step

```
FROM maven:3.6-jdk-8-alpine
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
RUN mvn -e -B package
CMD ["java", "-jar", "/app/app.jar"]
```

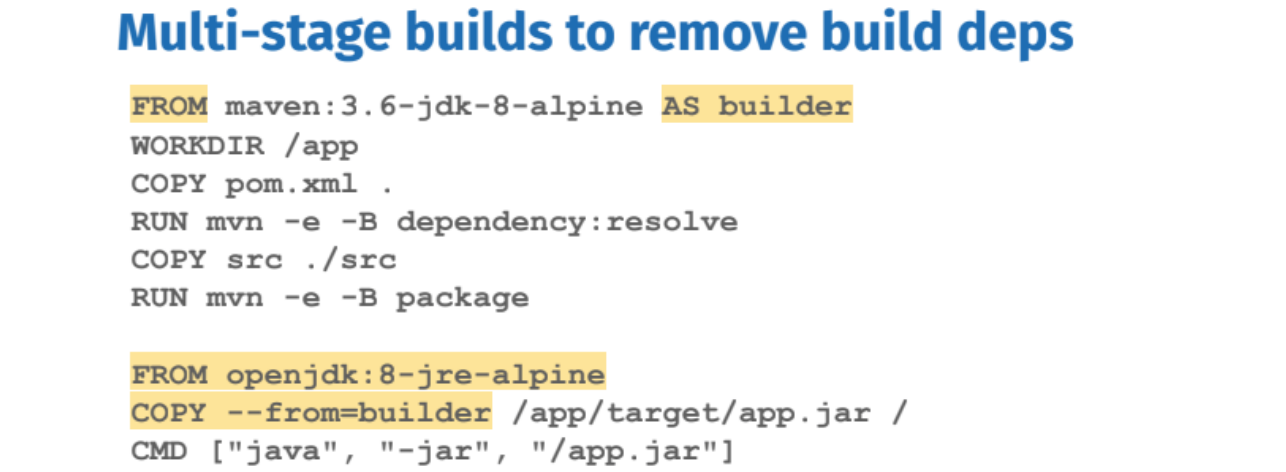
By again thinking in terms of cacheable units of execution, we can decide that fetching dependencies is a separate cacheable unit that only needs to depend on changes to pom.xml and not the source code. The RUN step between the two COPY steps tells Maven to only fetch the dependencies.

There is one more problem that got introduced by building in consistent environments: our image is way bigger than before because it includes all the build-time dependencies that are not needed at runtime.

Tip #11: Use multi-stage builds to remove build dependencies (recommended Dockerfile)

Multi-stage builds are recognizable by the multiple FROM statements. Each FROM starts a new stage. They can be named with the AS keyword which we use to name our first stage "builder" to be referenced later. It will include all our build dependencies in a consistent environment.

The second stage is our final stage which will result in the final image. It will include the strict necessary for the runtime, in this case a minimal JRE (Java Runtime) based on Alpine. The intermediary builder stage will be cached but not present in the final image. In order to get build artifacts into our final image, use `COPY --from=STAGE_NAME`. In this case, STAGE_NAME is builder.



Multi-stage builds is the go-to solution to remove build-time dependencies.

We went from building bloated images inconsistently to building minimal images in a consistent environment while being cache-friendly. In the next blog post, we will dive more into other uses of multi-stage builds.