Monorepo vs Multirepo

In this lesson you will learn about the two approaches to mapping your microservice code bases to source controlled repositories.

Organizing microservices source code

Using source control to safely store and keep track of your code base is critical for any type of application these days. However, when dealing with microservices, you will usually start wondering what's the best way to organize the source code for each of them.

There are basically two approaches you can follow with microservices code repositories:

- Use a single repository for all your microservices, also known as monorepo
- Or, use one repository per microservice, or multirepo

Let's explore each of these approaches in detail.

Single repository (Monorepo)

In the monorepo approach you have the code base for all your microservices living in just one repository. To make any changes to any of the microservices, all you have to do is clone the single remote repository to your box, make the updates and later push those changes back to remote.

Using this approach is good enough and very convenient when you are in the initial stages of your project, since the codebase is small, and the entire codebase is usually undergoing lots of changes.

However, as the code base of each microservice grows, and more people starts joining the coding effort, you will likely start experiencing a few issues.

To start with, since all microservices live in the same repo, it is very tempting to create direct code dependencies between different services. For instance, Trading microservice developers might decide to directly use the Catalog microservice classes to retrieve items from the Catalog database.

This would immediately generate a strong coupling between the two microservices, which would mean Catalog can no longer change freely without having to worry about breaking Trading. Catalog and Trading would have lost their autonomy.

The next issue is related to building and deploying code after a change is merged. For instance, if a change is made to the Identity microservice (even if it's just one line), the continuous integration server will have to rebuild the entire repository, producing new binaries for all the microservices.

This can be very time consuming because the CI server would need to clone, build, and test tons of code that did not change at all, plus it's not easy to tell which of the microservices need to get redeployed, so you might end up deploying all of them unnecessarily.

Consider also what happens when a compilation error is introduced to the mono repo. If the code base of any of the microservices is broken, then the entire repository's build is broken too, and it is no longer possible to produce builds for any of the microservices.

This can become a significant bottleneck that prevents most microservices to get deployed even when only one of them is actually broken.

Lastly, if a bug is detected in one of the microservices in Production, the best option in many cases is to roll it back to the previous version. However, since that bug may involve the codebases of more than one microservice, you may not have other option than roll back all the microservices. And, when the bug is fixed, you would need to redeploy a new version for all of them again. This can be very painful and cause a lot of issues along the way.

There are even more issues with this approach, but this should give you an idea of the challenges you'll find with a monorepo.

Let's now see how things look like when using the multirepo approach.

Repository per microservice (Multirepo)

In the multirepo approach, the codebase of each microservice lives in its own repository, as opposed to have everything in a single repo. This brings in multiple benefits.

To start with, trying to have dependencies from one microservice code base to another is not really possible, since each repository is completely independent, making classes and projects of one repo unable to reference the ones in another repo.

Next, when a code change is made to the code base of one microservice, the CI Server will clone, build and run tests for only that one microservice. This is significantly faster than in the monorepo approach, allowing teams to release changes very quickly regardless of anything going on with other microservices.

What happens if a compilation error is introduced to, say, the Identity microservice? Well, in that case only the Identity build is broken, blocking new Identity releases to go out. However, this has no impact on the builds of any other microservice, which can keep releasing as long as their code bases are healthy.

Then, if a bug actually makes its way to Production, and the best option is to roll back to a previous version, you now have the option of only rolling back the broken microservice. This gives teams great flexibility to fix issues quickly without having to worry about the state of other microservices.

Lastly, the multirepo approach allows for much better ownership of the code bases, since now the Catalog team can have full ownership of the Catalog repository, the Inventory team can fully own the Inventory repository and the same for Identity and Trading.

In the next lesson you will learn how to add your microservice code base to a source control repository using Git.