What's wrong with the monolith?

Before diving into microservices and their benefits, it is probably worth it to understand first what a monolith is, its benefits and the issues it can present.

One possible architecture

To understand what we mean by a monolith let's look at one way we could design our Play Economy system.

We start with our desktop and mobile clients. Like we said, they need some sort of backend piece to enable the game experience. Therefore, we introduce a server component, likely hosted as a web server somewhere in the internet.

As we go through the requirements, we identify a module that needs to take care of the in-game catalog of items. So, we introduce the Catalog module. Right away we also perceive the need for a place to store these catalog items, which means its time to introduce a database along with a table to store the catalog items.

We also identify the need to store the information about our players, and in general any user of the system. So, we bring in a Users module, which we place in the same web server, next to the Catalog module. Also, since we already have a database configured for our web server, we decide to add tables for the Users module in that same database.

Next we identify the need to store the set of items that the player has purchased, so we bring in an Inventory module, and same as before we place it next to our other modules and add relevant tables to the same database.

We also need some place to perform the actual purchase process, and since this involves both debiting gil from the player and granting items to him, we know we will need to use some sort of transaction for this. So, we introduce a Trading module on the same server, and we let it take advantage of the transactional capabilities of the database.

Finally, we will need some way to secure the access to our web server so that only players registered in the system can make use of it. So, we bring in an Authentication and Authorization module, which we will abbreviate as Authz. As with everything else, we will place it on the same server and make it use the same database.

This system that we have just described is what we call a monolith, and more specifically a modular monolith. In this system all of our server side capabilities are hosted together in the same web server and they all use the same database. Likely we also have the source code for all the modules in a single repository, we build all the code base with a single build process and we deploy everything together.

Let's now look at the pros and cons of a monolith.

Monolith pros and cons

Here for the pros and cons of a monolith. First the pros:

PROS

- **Convenient for new projects.** Usually when you are just getting started a monolith is the best option since you can put all the source code in one place, including all common libraries. This lets you iterate quickly for a while.
- **Tools mostly focused on them.** Most IDEs are build around the concept of building a single application, which aligns pretty well with a monolith.
- **Great code reuse.** Since all the code lives in the same repository, it is easy to reuse code across the multiple modules of the application.
- **Easier to run locally.** Being a single process, it is straightforward to have the full application running in your box usually with a single command.
- **Easier to debug and troubleshoot.** When it's time to fix bugs there's usually a single place to look for logs to find out what happened. You can also start a debugging session in your box to reproduce and fix the issue.
- **One thing to build.** A single command is usually enough to build the entire code base locally and if using a continuous integration server, a single build pipeline will do the trick.
- **One thing to deploy.** All the application modules can be deployed as a unit with a single deployment pipeline.
- **One thing to test end to end.** End to end verification of the application can take place as soon as the application has been deployed, with no other moving pieces that could impact testing.
- **One thing to scale.** When the increased load demands more instances of the web application, there's only one app to scale. Same when the increased scale is no longer needed, there is only one thing to scale down.

Now, the cons:

CONS

- **Easily gets too complex to understand.** Since the entire code base is in a single place there will be a point where it starts becoming too hard to navigate it and understand the relationships between the different components.
- **Merging code can be challenging.** When using things like git and with more and more people contributing to multiple modules in a single code base, eventually resolving conflicts gets more troublesome and the chances of a bad merge that could impact unrelated modules increases.
- **Slows down IDEs.** If the devs use an Integrated Development Environment (IDE) like Visual Studio, the massive number of projects and source code will eventually start slowing down the IDE, starting with simple things such as opening the work environment.
- **Long build times.** The bigger the single code base becomes the more time it takes to build it. Things like incremental builds can help here, but this is something that can easily not be configured properly and, regardless, it is undesired to have to build code for all the application modules when usually you only work on one or two of them.
- **Slow and infrequent deployments.** Just like building the code takes longer it also takes more time to complete a deployment, since there are a lot of components to deploy with every new

build. Not only that, teams tend to delay deployments to Friday night, or even weekends due to the lack of confidence on the impact of the changes that go into that deployment. Remember that with each deployment you are deploying all the applications modules all the time.

- **Long testing and stabilization periods.** Once a deployment is complete, regardless of what changed in the deployment, you still need to test all scenarios across all of the application modules. Not only this is inefficient but also teams that don't have testing fully automated may need to run manual tests to compensate and run them, again, for all the application scenarios.
- **Rolling back is all or nothing.** Bugs will eventually make their way to a deployment and some times there's no time to code and deploy the ideal fix. In these cases you just want to roll back to a previous version. However, with a monolith, you will be forced to roll back the entire application, not just the faulty module and, once the fix arrives, you will need to redeploy the entire thing again, with all the associated testing and stabilization.
- **No isolation between modules.** The smallest bug in one of the modules that makes the web application crash will take down all the modules with it.
- **Can be hard to scale.** This really depends on the kind of app you are building but if you have some module that needs, say, lot of memory and therefore needs to be scaled to multiple instances to not take over all the ram in a single server, while all other modules are fine with a small amount of ram most of the time, you still need to scale the entire application with all the modules to satisfy the ram needs of the first module.
- **Hard to adopt new tech.** If you would like to move one or two of the modules to the latest version of, say, the .NET platform, you are forced to update all the modules along with it, which significantly increases the time investment and the associated risk of moving to a new stack. Moreover, if you need to switch a module to a different programming language or web framework, you still need to account for how that change will impact the rest of the modules.

In the next lesson we will start learning about microservices, their pros and cons and when it's a good time to start considering them.