Klaus Kopecz
January 5, 2018 8 minute read

Multi-Target Applications, Microservices and APIs

Follow RSS feed Like

5 Likes 2,075 Views 2 Comments

Multi-Target Applications (MTAs) are the result of a development process that at its end can build an archive containing multiple software or content pieces that is labeled by a unique version number. The purpose of MTAs is to enforce one common lifecycle for all these contained software pieces, although the runtime representation might be distributed.

An often-heard question is how that fits to µServices, as leading architecture principle for cloud software. In this blog, I want to explain how both concepts, MTAs and (µ)services, come together.

There is a recent blog on this topic that puts "MTAs vs. µServices" in the context of how the development-to-deployment process of an application has been designed. I recommend to read this first...

There are good reasons to couple the lifecycle of certain technical components, if they jointly represent a service offering a dedicated (business) capability. MTAs offer the option to leave it as a hidden implementation detail of how many runtime components such a service consists of. Now, one could think that MTAs could lead to monolithic applications, contradicting the µService approach. This is not the intent. To avoid this, the decision, which components shall have this coupled lifecycle, is critical. Also, µService architecture comprises a set of principles that have to be applied correctly also when working with MTAs.

If this is considered, MTAs are not contradicting µServices, but both complement each other. When designing an MTA, you still have to do your componentization homework to avoid the monolith. However, MTAs enable you to choose the "size" of your µService according to your trade-off decisions related to "coarse-grained" vs. "fine-grained" services. Furthermore, MTAs help you to manage your dependencies to others, a task which gets more difficult the more µServices are around you.

Fowler's µServices

Let' start with a brief recap of how Fowler motivated the case for µServices (see the blog *Microservices – a definition of this new architectural term* from James Lewis and Martin Fowler, available at https://martinfowler.com/articles/microservices.html – my last read-through was in January 2018). There are basically two promises: (1) improved scaling behavior and (2) componentization according to business capabilities and not according to technical capabilities and skills.

According to Conway's law, cross-functional teams (each responsible for a business capability) will lead to an architecture that is clustered according to business capability. What has to be considered here as well (and what is missed in Fowler's µService view outlined in his blog referenced above) is the UI and the database. When combining Fowler's view with UI and database, µServices would look something like this:

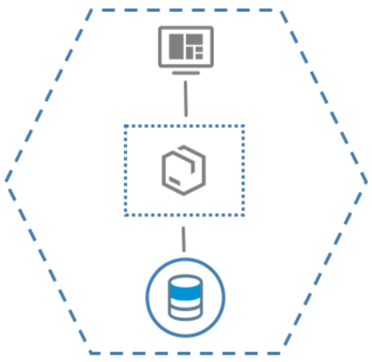

Figure 1: Microservices with database and UI

So, if the team's responsibility is for a complete business capability, in general, their µService will still be composed of multiple technical components. If they want to attach one lifecycle (from dev to deploy) to these components, then the µService is an MTA!

In some scenarios, the database can be removed "out of" the µService by having a central DB "backing" service that is shared by multiple µServices. However, this introduces a **dependency** that needs to be managed now. First, you are relying on someone else to provide this backing service and your µService must contain an element that describes this dependency. This dependency should be **configurable**, because you might want to deploy your µService into different settings (e.g. Test vs. Production). This is the price you have to pay for giving away the database. MTAs help you to manage the remaining dependency. Your µService now looks like this:

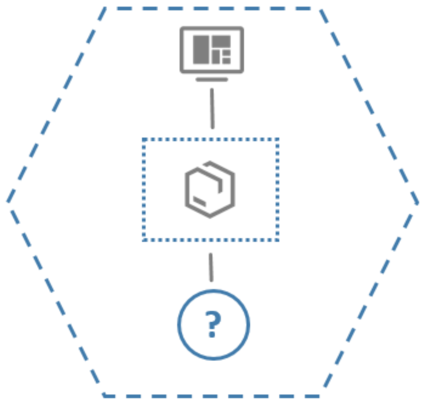

Figure 2: Microservice with dependency to database

The database part is even harder to remove from your µService, if you want or need to deploy schema definitions or want to "push down" application logic to this database (as happening when pushing down code to SAP HANA). You could hold your business logic code module responsible for keeping the DB schema definition + DB code and for pushing it down to the database.

This has limitations, as you might want to treat your DB code (e.g. DB procedures) as a code module, like you do it for Java and others. You might want to have development support for creating, checking and debugging such code. When it comes to deployment, you need to have something that takes your DB code and deploys it to the appropriate schema in the shared database. And you will want to have one joint deployment lifecycle for the DB code and your usual Java code. Suddenly, you have to solve an orchestration problem, because your µService is looking somehow like this:

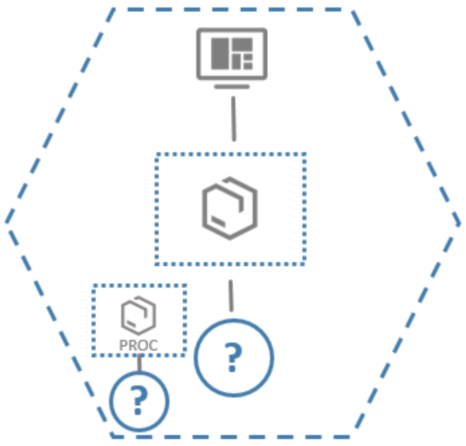

Figure 3: Microservice with dependency to database and DB code

The MTA deploy service implementations for SAP HANA XS Advanced and the Cloud Foundry environment of SAP Cloud Platform helps you to orchestrate DB code deployment with other application code deployments.

Now, let's care about the UI part. If you want to remove UI-related resources "out of" your µService via server-side rendering, you might end-up with one technical component (neglecting the code push-down aspect above) that might be for instance an Java EE server equipped with servlets, beans, etc. However, this does not fit to SAP-Fiori-like UI strategies that imply that there are rich JavaScript clients and server components serving the static content for these clients. Similar to DB procedures, you will want to develop your JavaScript UI code as a separate module that at the end somehow needs to be deployed somewhere. Again, you end up in a deployment orchestration issue (which can be addressed by MTAs). In this case, your µService keeps looking as before.

What comes next is that if you service is really "micro" (in dimensions of business capabilities), it is likely that its UI will be used only in the context of a UI integration service (such as Portal or SAP Fiori Launchpad). A similar thing happens as for the database: because you are "micro", you are depending on a central service that is not provided by you, so you have to manage this dependency. As a result, your µService can end up in looking like this:

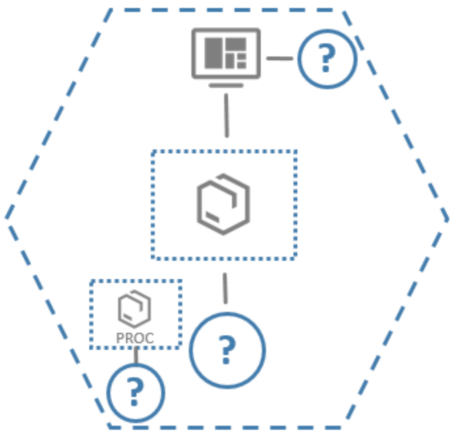

Figure 4: Microservice with dependencies to database, DB code, UI

This could be further continued by saying you might want to manage the UI integration content (required to configure the UI integration service to display your UI appropriately) as a separate content module being part of your µService, or a documentation module that shall be part of your project and the deployed version of your docs should always match the version of your code.

To conclude, it is possible to provide a µService that looks like this:

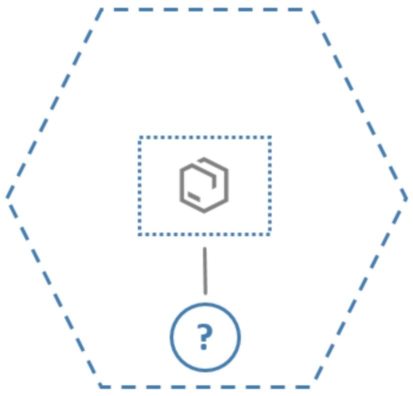

Figure 5: Microservice with generic dependencies

This is reasonable, if you want to provide a reusable, stateless service. In all other cases, this can happen only under very special assumptions on UI technology, on the role of the database and on DB content

management. So, it can make sense to give your µService a structure. MTA concepts have been made to manage such structures from development to deployment.

APIs, (µ)Services and MTAs

Another way to link MTAs with (µ)Service concepts is to consider the role of (public) APIs. I'm using the prefix "(public)" to emphasize that the APIs I'm referring to here have certain qualities that distinguish them from just being provided interfaces. For me, the term "API" always implies these qualities. Below, I'm using "API" always synonymously to "public API":

(Public) API qualities:

- Stable or backward compatible (contract)
- Well-documented
- Discoverable

If you are consuming a provided interface that is not a (public) API, you've implicitly coupled the lifecycle of your code to this interface:

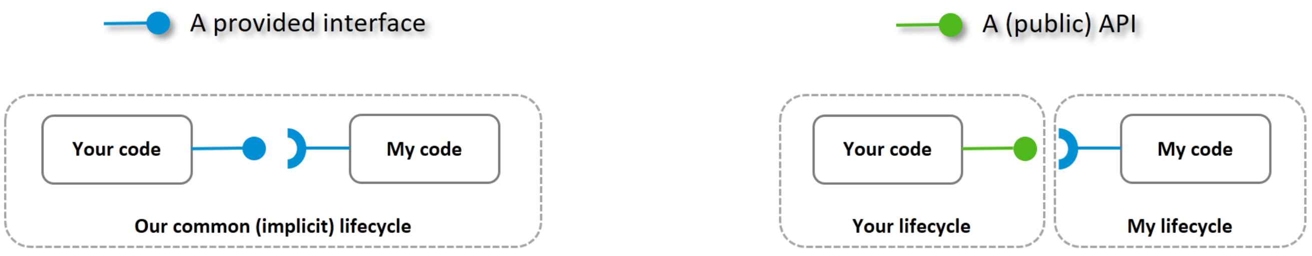

Figure 6: Interface vs. API

The reason is that you always have to watch and react on changes to the provided interface. Any new release from the interface provider might force you to deliver a new release of your code, because there is no contract available. This fact is often forgotten when praising µServices. Technically, you might be able to compose any stuff into nice small components, but do they all offer a (public) API? If not, you are left with the burden of managing all the implicit lifecycle couplings.

What does this mean when looking at this small, agile, cross-functional team that wants to provide a business capability based on a µServices architecture, where this capability is technically represented by multiple runtime components (such UI, application logic, DB)? They have two choices: Either they introduce (public) APIs for these components (OPTION 1 in figure 7 below), or they accept and manage the coupled lifecycle (OPTION 2 in the below figure).

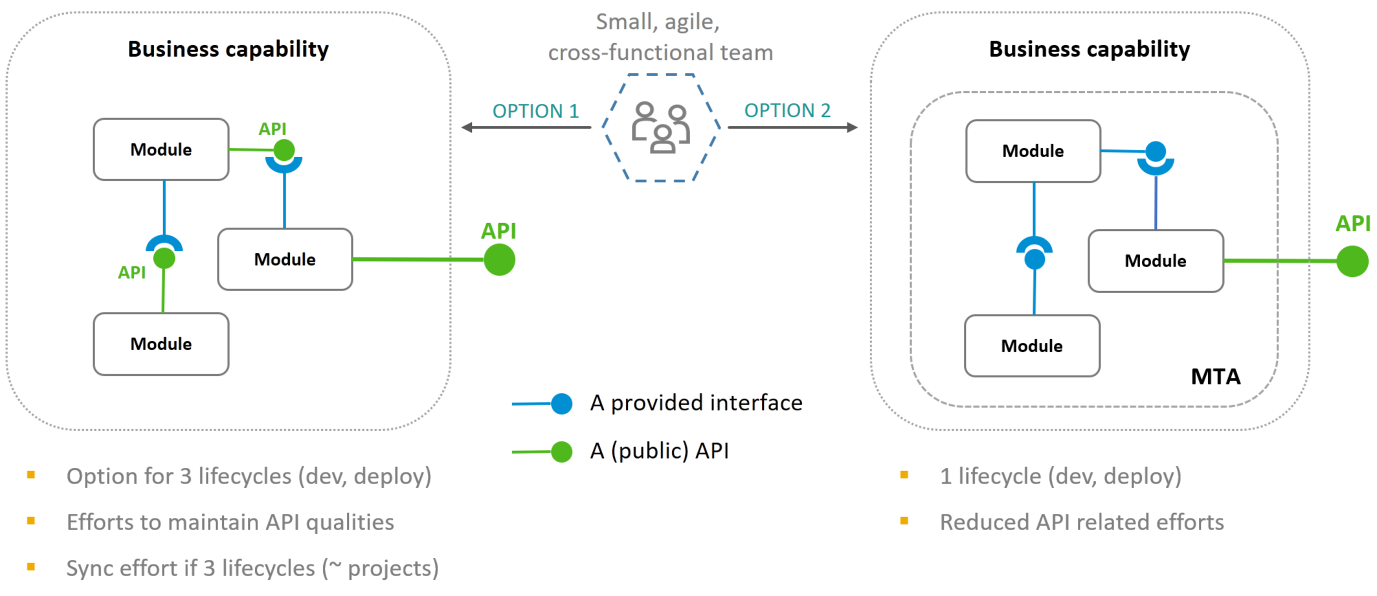
Small, agile, cross-functional team
OPTION 1
OPTION 2
Business capability
Module
API
Module
API
Module
API
Business capability
Module
Module
Module
MTA
API
A provided interface
A (public) API
Option for 3 lifecycles (dev, deploy)
Efforts to maintain API qualities
Sync effort if 3 lifecycles (~ projects)
1 lifecycle (dev, deploy)
Reduced API related efforts

Figure 7: APIs vs. management of coupled lifecycle

OPTION 1 enables to develop, release, deploy the components with independent lifecycles. At the same time, you have API-related effort to assure the required API qualities. The question then is: why should the team invest additional API-related efforts, if they need to deliver the entire business capability anyway always as a whole entity. It will not make sense to deliver "parts" of a business capability. If this would make sense, then the team did the wrong cut anyway, because this "part" is then a business capability of its own (go back to start …). Further, it's a "small" ("pizza size") team. They don't need rigid contracts, because they can communicate and implement necessary changes quickly.

In my opinion, it's better to go for OPTION2 and accept the implications of coupled lifecycles. **MTAs are a way to turn implicit lifecycle coupling into explicit lifecycle coupling**. That's the essence of MTAs. As the lifecycle is now an explicit one (captured by the MTA model), any tool that contributes to development, release, distribution, deployment, operations, monitoring processes can make use of this and help the team to establish the required common lifecycle for its components.

There will be a cost for introducing the MTA. It's probably not a big issue to author some additional metadata (descriptors). What is more costly is that an investment has to be made into development and deployment tools, so that they are able to interpret MTA metadata and perform the appropriate orchestrating actions. However, such investments have to be made only once and within a big software provider, a lot of teams can benefit from these.

Where is the service?

The idea of providing a business capability via a well-defined (public) API gets blurred, if there are multiple (public) APIs hanging around with the service. Although not available today, an MTA model can be the base for deployment and network infrastructures for exposing really only those endpoints, which are meant to provide the business capability while leaving others restricted to MTA-internal communication.

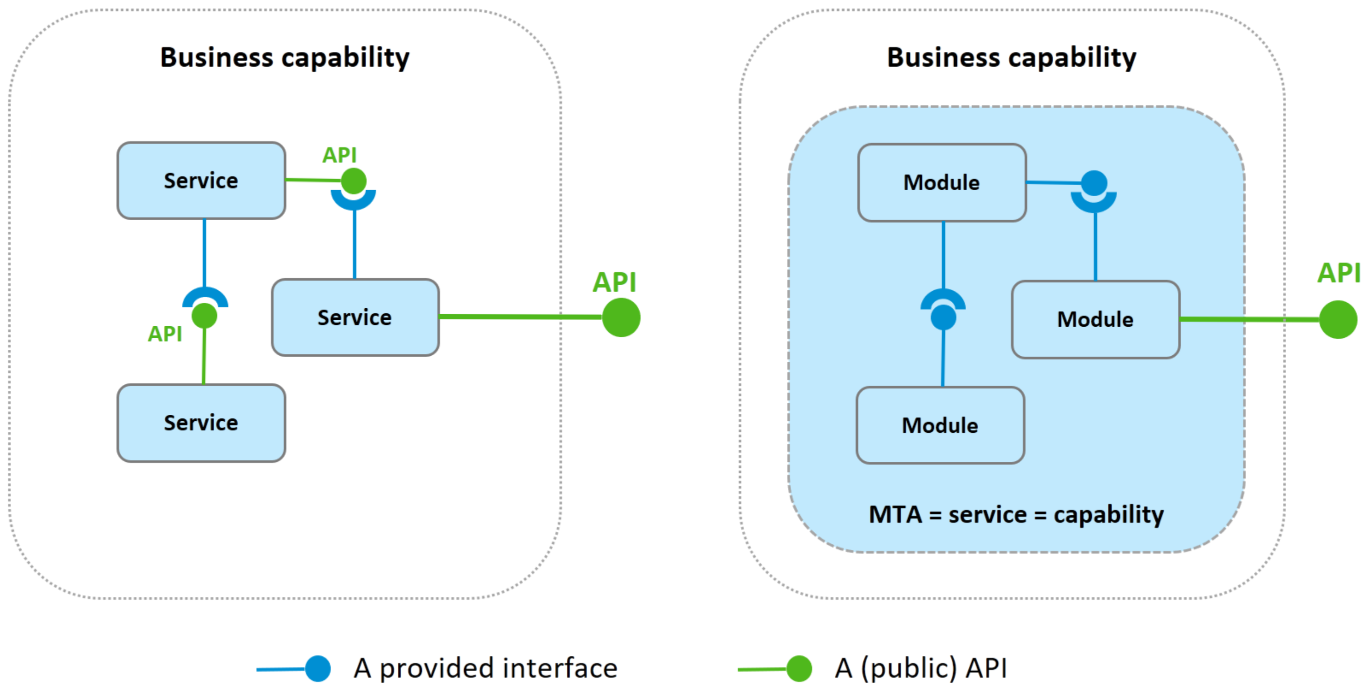

Figure 8: On the right side, the service is provided by one well-defined (public) API, while the number of runtime components and their internal interfaces are hidden as implementation detail "behind" the API

Alert Moderator

Assigned tags

SAP Cloud Platform | development tools for SAP Cloud Platform | micro services | MTA | Multi-Target Application |