Benefits & Variations

In this lesson, we'll look at some benefits of self-contained systems and some variations to the approaches discussed here.

WE'LL COVER THE FOLLOWING

Benefits

SCSs are a logical extension of the idea of bounded contexts.

> A **bounded context** contains the logic on a specific part of the domain.

SCSs make sure that the UI and the persistence are part of one single microservice so that the **split by domain that domain-driven design advocates are further improved by SCS**.

Ideally, one feature should lead to one change. In an SCS system, the chance that this actually happens is quite high: the change will likely be in one domain. It can be contained in one SCS even if the UI and the persistence need to change, and the change can be put into production with one deployment. Therefore, **SCSs provide better changeability**.

Also, **testing is easier** because all the required code for the domain is in one domain. So, it is easier to do meaningful tests of the logic together with the UI.

Due to the focus on the UI, SCSs make it **easier to implement integration in**

the UI. So, SCSs open up additional integration options. This is particularly

useful for heterogeneous UI technology stacks. With the rate of innovation in UI technologies, it is unrealistic to assume a uniform UI technology stack.

Variations

Instead of implementing an SCS with UI, logic, and data there are different variations. These architectures are not SCS architectures; however, they might still be sensible in certain contexts.

- Domain microservices with logic and data but *without UI* can make sense when an API or a backend is to be implemented.
- Microservices without logic or data, which consequently implement *only a UI*, can be a good approach for implementing a portal or a different kind of frontend.

Both of these variations are sensible when the project is a pure frontend or backend and does not allow the full implementation of an SCS.

Typical changes

Good architecture should **limit a change to one microservice**. The basic assumption of the SCS architecture is that a change typically goes through all layers and is limited to a domain implemented in one microservice. This assumption has been confirmed in many projects.

Still, if a change in a project usually affects only the UI or only the logic, it may be better to implement a division by layers. A new look and feel or new colors can be implemented in the UI layer, while logic changes can be implemented in the logic layer – of course, only if they do not affect the UI.

Possibilities for combinations

Self-contained systems are easy to combine with other recipes.

- Frontend integration (see chapter 3) is preferred when integrating SCSs.
- SCSs focus on asynchronous communication (chapter 6). SCSs are web applications so that the use of Atom (chapter 8) for asynchronous REST

communication is especially easy because it also builds on HTTP. Kafka

(chapter 7), on the other hand, has a different technical foundation; even though it can still be combined with SCSs, it represents an additional technical effort.

- Synchronous communication (chapter 9), even though possible, should be avoided.

Thus, UI integration, asynchronous communication, and synchronous communication between SCSs are possible, but a clear prioritization exists. Of course, SCSs can also communicate with microservices or other systems via these mechanisms.

QUIZ

1

Suppose you are implementing a system where one change always affects the UI or the logic. How would this system best be divided?

COMPLETED 0%

1 of 2

In the next lesson, we’ll look at the conclusion to this chapter!