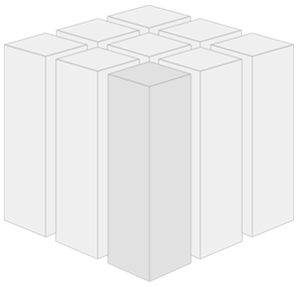

Self-Contained Systems

ASSEMBLING SOFTWARE FROM INDEPENDENT SYSTEMS

Introduction

The Self-contained System (SCS) approach is an architecture that focuses on a separation of the functionality into many independent systems, making the complete logical system a collaboration of many smaller software systems. This avoids the problem of large monoliths that grow constantly and eventually become unmaintainable. Over the past few years, we have seen its benefits in many mid-sized and large-scale projects.

The idea is to break a large system apart into several smaller self-contained systems, or SCSs, that follow certain rules.

[Speaker Deck]

Talk by [Roman Stranghöner]

[Full Screen]

Self-Contained Systems

Infodeck

innoQ

SCS Characteristics

1. Each SCS is an autonomous web application. For the SCS's domain, all data, the logic to process that data and all code to render the web interface is contained within the SCS. An SCS can fulfill its primary use cases on its own, without having to rely on other systems being available.

2. Each SCS is owned by one team. This does not necessarily mean that only one team can change the code, but the owning team has the final say on what goes into the code base, for example by merging pull-requests.

3. Communication with other SCSs or 3rd party systems is asynchronous wherever possible. Specifically, other SCSs or external systems should not be accessed synchronously within the SCS's own request/response cycle. This decouples the systems, reduces the effects of failure, and thus supports autonomy. The goal is decoupling concerning time: An SCS should work even if other SCSs are temporarily offline. This can be achieved even if the communication on the technical level is synchronous, e.g. by replicating data or buffering requests.

4. An SCS can have an optional service API. Because the SCS has its own web UI, it can interact with the user — without going through a UI service. However, an API for mobile clients or for other SCSs might still be useful.

5. Each SCS must include data and logic. To really implement any meaningful features both are needed. An SCS should implement features by itself and must therefore include both.

6. An SCS should make its features usable to end-users via its own UI. Therefore the SCS should have no shared UI with other SCSs. SCSs might still have links to each other. However, asynchronous integration means that the SCS should still work even if the UI of another SCS is not available.

7. To avoid tight coupling an SCS should share no business code with other SCSs. It might be fine to create a pull-request for an SCS or use common libraries, e.g. database drivers or oAuth clients.

8. To make SCSs more robust and improve decoupling shared infrastructure should be minimized. E.g. a shared database make fail safeness and scalability of the SCSs depend on the central database. However, due to e.g. costs a shared database with separate schemas or data models per SCS might still be a valid compromise.