Definition

In this lesson, we'll further discuss and define microservices.

We'll cover the following

- Rules for communication
- SCS databases
- Rules for the organization
- Rule: minimal common basis
  - Avoid shared business logic
  - Avoid common infrastructure

Self-contained systems make different macro architecture decisions.

- Each self-contained system is an **autonomous web application**, meaning SCSs contain web UIs.
- There is **no common UI**. An SCS can have HTML links to other SCSs or can integrate itself by different means into the UI of other SCSs. But every part of the UI belongs to an SCS. Chapter 3 (https://www.educative.io/collection/page/10370001/5441945024331776/6168726367895552) describes different options for the integration of frontends. This means each SCS generates a part of the UI. There is no separate microservice that generates all of the UI and then calls logic in the other microservices.
- The SCSs can have an **optional API**. This API can, for example, be useful if mobile clients or other systems have to use the logic in the SCS.

- The **complete logic** and **data** for the domain are contained in the SCS. This is what SCSs were named for. An SCS is self-contained because it contains UI, logic, and data.

These rules ensure that an SCS completely implements a domain. This means that **a new feature causes changes to only one SCS** even if the logic, data, and the UI is changed. These changes can be rolled out with a single deployment.

If several SCSs shared a UI, many changes would affect not only the SCSs but also the UI. Then two well-coordinated deployments and a closely coordinated development would be necessary.

Rules for communication

The communication between SCSs has to follow a number of rules:

- Integration at the **UI level** is ideal as the coupling is very loose. The other SCS can display its UI as required. Even if the UI is changed, other SCSs are not affected.
    - For example, if HTML links are used, the integrated SCS doesn't even have to be available. The link is displayed even if the system is unavailable; there will be an error if the user clicks on the link. This helps with resilience because the failure of an integrated SCS does not affect other SCSs.
- The next option is **asynchronous communication**. The advantage of this is that if the integrated SCS fails, the requests are delayed only until the failed SCS is available again. However, the calling SCS will not fail because it has to be able to deal with longer latency times.
- Finally, integration with **synchronous communication** is also

possible. In such cases, precautions must be taken to deal with the potential failure and slow responses of the integrated SCSs. In addition, the response times add up if you have to wait for the responses of all synchronous services.

These rules focus on a very loose coupling between the SCSs to avoid error cascades in which one system fails and consequently the dependent systems fail.

SCS databases

These rules mean that an SCS **replicates data**. The SCS has its own database, and since it is primarily intended to communicate asynchronously with other SCSs, it is a challenge if the SCS must use data from another SCS to process a request. If the data is requested during the processing of the request, the communication is synchronous.

For asynchronous communication, the data must be replicated beforehand so that it is available in the SCS when processing the request.

Consequently, SCSs are **not always consistent**. If a change to the dataset has not yet been passed to all SCSs, then the SCSs have different datasets which may not be acceptable in some situations. For particularly high-consistency requirements, the SCSs must use synchronous communication. In this scenario, one SCS receives the current state of the data from the other SCS.

Rules for the organization

SCSs provide macro architecture rules For the organization that they are being used in. An SCS belongs to **one team**.

The team does not necessarily have to make all the changes to the code, but it must at least review, accept, or reject changes. This enables the team to direct and control the development of the SCS.

A team can handle several SCSs. However, it isn't a good idea for an SCS to be changed by more than one team.

Thus, self-contained systems **use strong architectural decoupling to achieve organizational advantages**.

- The teams do not need to coordinate very much and can work in parallel on their tasks.
- Requirements can usually be implemented in one SCS by one team because an SCS implements a domain. Coordination in this respect is therefore hardly necessary.
- Because the technical decisions mostly concern only one SCS, there are hardly any arrangements necessary.

For SCSs, as well as for microservices and other types of modules, the division is aimed at enabling independent development. Having several teams change the same module is obviously not a good idea.

The modules allow independent development, but when several teams are working on one module, close coordination is still necessary. The advantage of modularization is not exploited then. This is why equal joint development of an SCS by several teams is not allowed.

Rule: minimal common basis

Because self-contained systems aim at enabling a high degree of

independence, the common basis should be minimal.

Avoid shared business logic

Business logic must not be implemented in code used by more than one SCS. *Shared business logic* leads to a close coupling of the SCSs which should be avoided. Otherwise, a change to one SCS could require changes to the shared code.

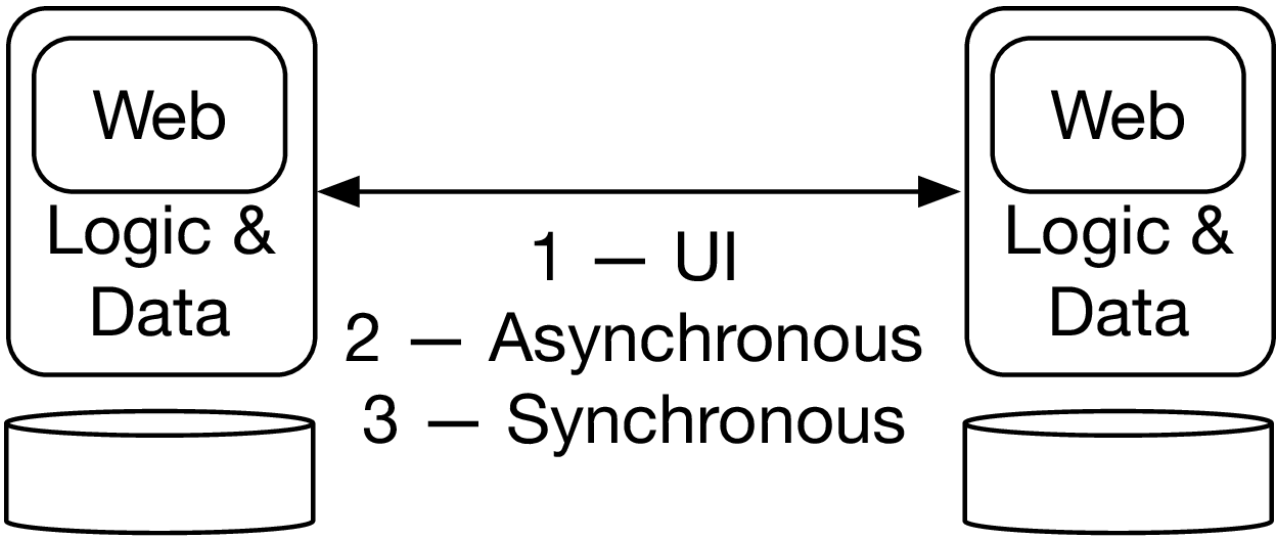
Such changes must be coordinated with other users of the code which creates a tight coupling that SCSs should avoid. In addition, common business code indicates a bad domain macro architecture. Business logic should be implemented in a single SCS. The business logic in an SCS can, of course, be called and used by another SCS via the optional interface of the SCS.

Avoid common infrastructure

Common infrastructure should be avoided. SCSs should not share a database, otherwise, the failure of the database could lead to a failure of all SCSs.

However, a separate database for each SCS requires a considerable effort. For this reason, compromises are conceivable if the robustness of the system is not quite so important. The SCSs could have a separate schema in a common database. A shared schema would violate the rule that each SCS should have its own data.

The drawing below provides an overview of the most important features of the concept of self-contained systems.



SCS: Concept

Here are some shorter rules to keep a minimal common basis:

- Each SCS contains its own *web UI*.
- In addition, the SCS contains the *data* and the *logic*.
- Integration is *prioritized*. UI integration has the highest priority, followed by asynchronous and finally synchronous integration.
- Each SCS ideally has its *own database* to avoid a common infrastructure.

Q U I Z

1 Which of the following is the best for inter-SCS

communication?

A) UI Integration

B) Asynchronous Communication

C) Synchronous Communication

Submit Answer

Question 1 of 6
0 attempted

Reset Quiz

In the next lesson, we'll look at an example of an SCS architecture.

← Back

Introduction

Next →

An Example

Mark as Completed

Report an Issue