Architecture Decision Record 001

TAPAS Group 3

Status:	Accepted
Date:	30/09/2021
Decided by:	Team

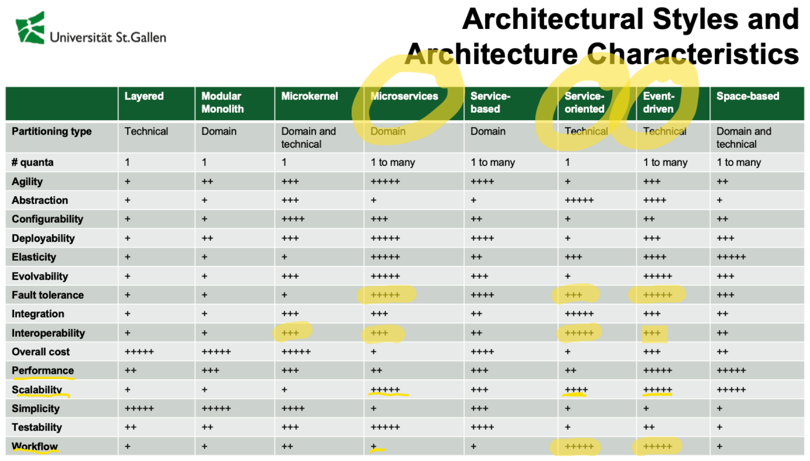
Architectural Style

Which architecture style is best for the implementation of the TAPAS system?

Context

From the architecture characteristics worksheet, we identified **interoperability, fault tolerance and workflow** consistency as key characteristics.

We identified **service-based microservices** architecture styles as the most suitable styles for the identified characteristics. Furthermore **feasibility, maintainability and simplicity** are the most relevant implicit characteristics.



Universität St.Gallen

Architectural Styles and Architecture Characteristics

	Layered	Modular Monolith	Microkernel	Microservices	Service-based	Service-oriented	Event-driven	Space-based
Partitioning type	Technical	Domain	Domain and technical	Domain	Domain	Technical	Technical	Domain and technical
# quanta	1	1	1	1 to many	1 to many	1	1 to many	1 to many
Agility	+	++	+++	+++++	++++	+	+++	++
Abstraction	+	+	+++	+	+	+++++	++++	+
Configurability	+	+	++++	+++	++	+	++	++
Deployability	+	++	+++	+++++	++++	+	+++	+++
Elasticity	+	+	+	+++++	++	+++	++++	+++++
Evolvability	+	+	+++	+++++	+++	+	+++++	+++
Fault tolerance	+	+	+	+++++	++++	+++	+++++	+++
Integration	+	+	+++	+++	++	+++++	+++	++
Interoperability	+	+	+++	+++	++	+++++	+++	++
Overall cost	+++++	+++++	+++++	+	++++	+	+++	++
Performance	++	+++	+++	++	+++	++	+++++	+++++
Scalability	+	+	+	+++++	+++	++++	+++++	+++++
Simplicity	+++++	+++++	++++	+	+++	+	+	+
Testability	++	++	+++	+++++	++++	+	++	+
Workflow	+	+	++	+	+	+++++	+++++	+

Decision

After active discussion within the group, we decided to go with a **microservice** architecture.

Reasoning: A microservice architecture makes sense because it also provides high agility compared to the service-oriented architecture. The domains auction-house, executors, task list and roster also lead to a natural segregation of functionality. Further, they in some cases only need access to certain data irrelevant to other domains, which also implies a segregation.

If it makes sense, we can also apply service-based patterns. (Sharing a database between different microservices) We also expect that due to different execution times some executors need highly asynchronous communication, which needs to be kept in mind for potential later adaptions.

Consequences

- System architecture will be slightly more complex because we have multiple services that interact with each other
- Our system will be very easy to extend
- Partial deployment will be possible
- Scalability and fault tolerance will be very good
- Team organization will be easy because we can assign the responsibility for a microservice to a sub-team

Architecture Decision Record 002

TAPAS Group 3

Status:	Accepted
Date:	30/09/2021
Decided by:	Team

Service Granularity

Context

During event storming we identified the following domains:

- *Task Manager Domain*
- *Roster Domain*
- *Executor Pool Domain*
- *Executors Domain*
- *Auction House Domain*

Furthermore, we identified during discussion the following integrating and disintegrating forces:

Disintegrators:

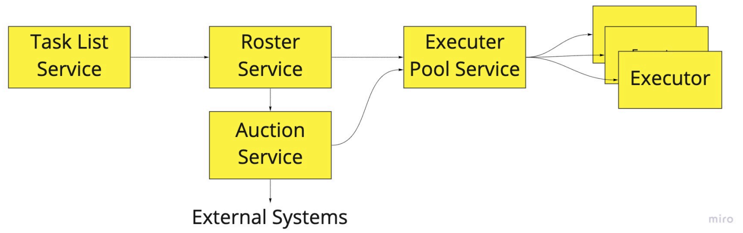
- **Service functionality:** The domains auction-house, executors, task list and roster have different functions. According to Single Responsibility Principle, we separate them into different services.
- **Scalability and throughput:** There would be many users and operations in the task list and roster domain but few users and operations in the auction house and executor's domain.
- **Fault Tolerance**: Fault tolerance is one of the driving characteristics we identified. Therefore, we separate the domains according to the bounded context map, so that the whole system and the other services would not fail in case a service fails.

Integrators:

- **Data relationships:** Both roster and task list need to have consistent data. Since tasks are central for the core operation, a single source of truth is beneficial for indicating the current state of tasks.
- **Workflow consistency:** Responsiveness of the whole system is not crucial as the executors take varying time for execution. Thus, low latency is only necessary for the task list as a microservice, where the user interaction takes place.

Decision

We decided to split up services according to the 5 identified domains. Data relationship is secured by introducing a shared database between the task list and the roster. For a detailed description see next page.



Consequences

Communication: Due to their nature some task take a lot of time to complete. We therefore expect to need asynchronous communication especially between the executor pool and actual executors. In microservice based architecture, asynchronous communication is made to allow decoupled systems, and to increase scalability and performance, which are only necessary for some quanta in the system.
Consistency: With this design we get eventual consistency because the data is spitted over multiple different services. Microservice based architecture is a distributed architecture, which shows high scalability, performance and availability at the expense of data consistency and integrity.
Coordination In our design (choreography), each service calls other services as needed without a mediator. However, it is also possible to build a localized mediator to handle the complexity and coordination required for the business workflow, which will create coupling between the services.

Status:	Accepted
Date:	30/09/2021
Decided by:	Team

Service Granularity

Service Description

Task List Service
The task list service provides all functions that are directly used by our end user. Mainly for logging and managing operations for tasks. The actual execution of the tasks is delegated to the *Roster Service*.

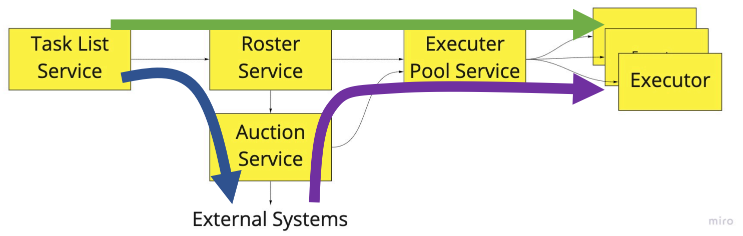
Roster Service
The main job of the roster service is to decide if a task should be executed internally or by an external executor via the action house. If internally executed, it forwards the task to the executer pool service. Therefore, the main responsibility of the roster service is to orchestrate tasks and bookkeeping.

Executor Pool Service
The executor pool acts as a facade for all the different internal executors. It therefore also keeps track which internal executors are available, their capabilities and registers new executors. If it gets a request, it forwards it to the correct executor. The executor pool does not differentiate between internal and external tasks.

Action Service
The responsibility of the action service is to manage all ingoing and outgoing action requests. It therefore places bids if another team outsources a task and it creates an action if the roster requested to execute a task. If the action house won a bid, it also uses the executor pool service to execute the external task.

Executor Service(s)
An actual task executor is one logical unit that can execute some task. It can register itself at the executor pool and then receives tasks form the executor pool service. There will be multiple instances of executors in the system with different capabilities.

Possible flow of tasks:



🟩 Task created by a user that can be executed internally

🟦 Task created by a user that cannot be executed internally and we need to outsource

🟪 Task from another group that we won during auction