

Architecture Decision Records

- 001-Architectural Style
- 002-Service Granularity
- 003-Communication
- 004-Queueing Tasks in Executor Pool
- 005 – Auction House Integration
- 006 – Uniform HTTP API
- 007 – Data Ownership
- 008 – Code Sharing


001-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 adaptations.

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

001-Architectural Style Retrospective

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

Retrospective

When we take a look at the implemented system we see the following results:

- The system's fault tolerance is very good. For instance if the auction house is unavailable we can still execute internal tasks. Vice-versa if the executor-pool is unavailable all task get executed externally. If the task and roster services are unavailable we still can bid for external tasks and execute their tasks.
- The scalability is also quite good. For instance we can increase the computational resources of one service without affecting the others. Due to the state-less ness of most of our logic we even could have multiple instances how certain service and implement horizontal scaling.
- Testability is also good. We were able to test every microservice independently by mocking the microservices. (We used Wiremock) This made integration testing very easy.

When we look at the downsides of our architecture we see the following pattern:

- The implemented architecture is quite complex, especially the asynchronous communication lead complex interaction patterns. Also because we have quite a lot of services the propagation of task updates (i.e. a task finished) takes many steps.
- We repeated ourselves quite a lot. I.e. there is a lot of boilerplate code for the operational aspects of every microservice (e.g. logging, testing). Of we implemented a monolithic system we had less duplicated code.

Learnings

The asynchronous execution of task was quite complicated to be implemented with the current architecture. We think that a event-driven approach could also have been a good fit for this project. Some aspects like task scheduling and updating task status would be very simple if we had a central message broker with different task queues.

002-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:

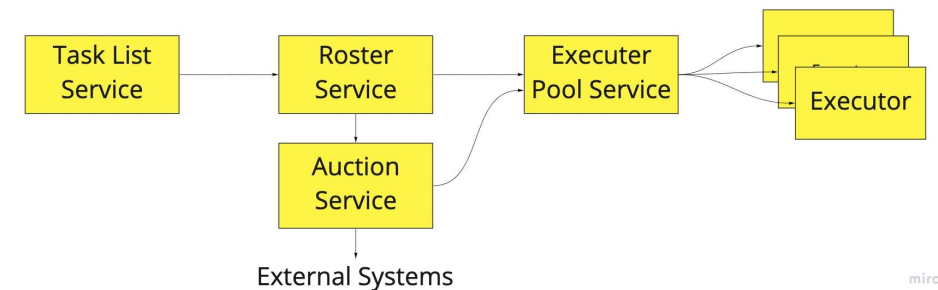
- **Service functionality:** The domains auction-house, executors, task list and roster have different functions. According to Single Responsibility Principle, we separate them into different services.
- **Scalability and throughput:** There would be many users and operations in the task list and roster 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.

002-Service Granularity

Service Description

Task List Service

The task list service provides all functions that are directly used by our end user. Mainly for CRUD 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 executor 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.

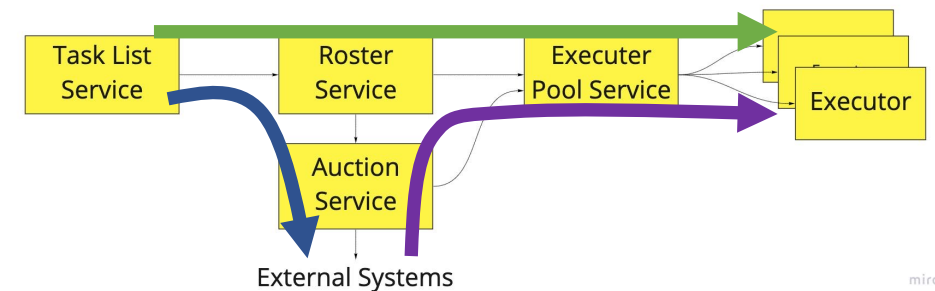
Action Service

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 from 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

002-Service Granularity Retrospective

Retrospective

Looking at our system we think that we made a good choice with the service granularity. Overall, we think that we found a good compromise between “Separation of Concerns” and “Keep it Simple”. The execution-pool and auction house separation led to a very natural way to split the responsibility of the services. Auction house for all auction related logic, and the execution pool for all executor related logic. We never had a argument which service should be responsible for something which is a sign that we spited up the services quite well.

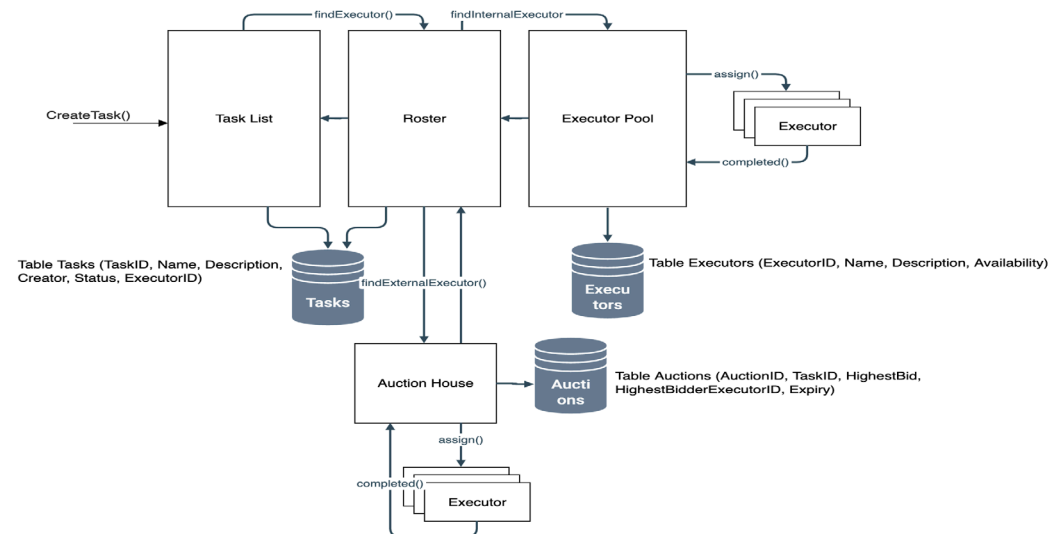
The only thing that we would implement differently next time is the roster service. It has only very little logic inside. It only decides if a newly created task goes to the executor-pool or to the auction house. So basically just a if-else statement. For that simple logic we think that creating a own microservices is a bit over-engineered. Having a own service for this small responsibility adds more latency to the system and makes it less reliable. It would have been easier to implement this logic within the tasks service. Therefore we think that it would have been simpler, less resource intensive and more performant merge the tasks and roster service into one microservice.

003-Communication

Context

In ADR001 we identified **interoperability**, **fault tolerance** and **workflow** as the most important system characteristics for the tapas application. Furthermore, we opted for an application with great **maintainability** and **simplicity** as the most important implicit quality attributes.

Here, we discuss the **method of communication** between our systems micro-services, in order to achieve the desired quality attributes.



Decision

After initial implementation of the basic functionality of the different micro-services, our group discussed how those micro-services should communicate. The agreed upon method was **asynchronous communication**.

Reasoning: We decided on this method of communication because our aim is to build a decoupled system with good scalability and performance, especially for certain quanta. **Executors Pool** and **Roster** are the quanta that will handle most of the workflow and communication. They must therefore be able to handle in- and outgoing communications simultaneously. This also allows for great **scalability** and the handling of many tasks. Finally, services are more **robust**, because when a service stops working during runtime, the rest of them can stay operational, instead of having to wait for that service to fixed.

Consequences

Communication times: Objects and values need to be transferred via request messages and setting up of remote processes between services, increasing communication times. But this is mitigated by the fact that, as a team we do not opt for the quickest possible response times but for a good overall workflow and correct execution of the tasks.

Statelessness: In cases where a service breaks down it might lead to it not knowing the state it was before the breakdown occurred. We will try to mitigate this by implementing databases for the various services to keep a log of tasks and their state and be able to return to their previous state, should a breakdown occur.

004-Queueing Tasks in Executor Pool

Context

After integrating the **Auction House** in accordance with the interoperability's team guidelines, we run into a problem with the queueing of the **external tasks**. If all our executors are occupied at the time an external task arrives, it will get rejected and will not be executed. Therefore, we need to implement a **mechanism** to deal with this problem.

Decision

We opted to implement a **queueing mechanism** within the **Executor's Pool**.

Reasoning: With the implementation of such a mechanism, the Executor's Pool can temporarily store all the tasks that are sent for execution and send them to the Executors when they are free.

We also decided not to use a queue for reasons specified under **Consequences**.

Consequences

With this queueing mechanism **no tasks** will ever get **rejected**. Every time the Executor's Pool receives a new task, either external or internal, it will check if an executor capable of executing it is free. If not, Executor's Pool adds the task to the queueing mechanism.

We opted to **not** use a **queue** in its definition, because it would mean that we have to implement a priority order. This would lead to executors remaining free because the next task in the queue would not be of the type relevant to that Executor. Therefore, we opted to implement a **queueing mechanism**, where every time an executor is made available, the Executor's Pool will search within that queue for the next available task that can be executed by that Executor. This way, tasks will have no unreasonable delays in their waiting time for execution.

One **problem** that we identified with the queueing mechanism is, that due to the Executor's Pool need to search through a lot of tasks, there will be added latency to the system.

005-Auction House Integration (I/II)

Context

Define communication between the tapas application and the auction house.

Tasks, which cannot be executed internally, need to be passed to the auction house to setup an auction for external execution.

Decision

Tasks are put on the roster, which checks whether the task can be executed internally or externally. In case of external execution the internal controller for the auction house is called via an HTTP Post to setup a new auction. The body of the message contains the task. In the auction house the task is further passed to the auction launcher, which will initiate the auction.

Consequences

- Different microservices for both internal and external execution would keep tapas running for some cases even if one service fails, keeping availability high
- Internal/external routing in roster keeps separation of execution for both subsequent services
- Only the auction house communicates with other systems in a standardized way leading to good interoperability. The rest of the system only communicates with the auction house via the internal controller

005-Auction House Integration (II/II)

Context

Define communication between the tapas application and the auction house.

Won auctions necessitate the execution of the given tasks. These tasks need to be passed to the tapas application, which subsequently propagates it to execution. When the task is finished, it will need to be returned group the task has been executed for.

Decision

Externally via auctions received tasks are wrapped to have the same attributes as internal tasks plus a flag indicating its external origin is added. Once the task is wrapped, it gets directly passed to the executor pool using the same endpoint as for internal tasks. The executor pool will then handle it as a normal task. After execution flagged tasks are routed to the auction house instead of directly patching tasks in the internal task list. The auction house proceeds by creating a HTTP Patch to mutate the task of the group it received the task from.

Consequences

- Communication with other groups remains solely in the auction house microservice (HTTP Patch) which benefits interoperability
- Direct routing to the executor pool keeps the internal task list separated from external tasks
- Same endpoints for execution can be used as for internal tasks
- A queue for an executor could lead to longer waiting times as no priorities are assigned to tasks

Status:	Accepted
Date:	13/11/2021
Decided by:	InterOp Group

006-Uniform HTTP API

Context

Auction houses of all groups need to be able to communicate with each other, while still hiding individual implementation details. This necessitates the definition of standardized knowledge across all groups including standard uniform identifiers, HTTP verbs, media types and response codes. Internal communication between the auction house and the tapas application is individual to each group and documented in ADR 005.

Decision

Events for auction starts are received via a message queue, which is listened on either the MQTT broker or PubSubHub. Relevant auction data is structured in a unified JSON schema (media type auction+json) providing information on unique identifiers of auction, auction house and task as well as adding information about the task type (unified across groups) and a deadline. After internal checks for eligible executors, auction houses pose a bid using the unified JSON schema bid (media type bid+json), which is composed of unique identifiers for the auction, the bidder and the bidder's tasklistURI. Subsequently, the winning bidder receives the complete task sent to its auction house using the unified JSON schema of media type task+json. On top of the usual attributes of class task, the bidder might also receive input data depending on the given task type. The bidder accepting the task is implied by the action of patching the task's state on the auctioneer's task list using the task URI. After acceptance the bidder proceeds with the actual execution by wrapping the external task into an internal task and handling it according to internally specified processes. As soon as the task is finished, another patch is made changing the auctioneer's task to executed. If there is any output data coming from the execution of the task, it will also be delivered via this patch. A more detailed explanation including response codes and task types is provided under the following [link](#).

Consequences

Relying only during the first step on MQTT and WebSub to subscribe for AuctionStartedEvents and then proceed with usual HTTP communication leads to simpler and already used ways to communicate after receiving Events. Broadcasting events across all groups is necessary in the first step, but afterwards other groups do not need to have further information about an individual auction except for the winner. Thus, we agreed on a fire and forget method for posing bids in the next step. Other groups do not need information about other groups' bids also implying a secret auction. With fire and forget no information on posed bids has to be stored internally (also not necessary as the size of the bid has no impact). After receiving the complete task a bidder automatically knows, that he won and can proceed with the acceptance patch. This procedure keeps the body for preceding messages small, as the actual task is only sent to the winning bidder. Directly accessing other groups' task lists via patches allows simple data exchange on states and output data before and after execution, while directly keeping DBs up to date. Wrapping received tasks to create internal tasks hides internal implementation from other groups making interoperability easier. The task class could be left as it is and only the values for two attributes need to be agreed upon (task states and output data). However, there are some drawbacks. Adjustments to the whole procedure would be necessary whenever posing bids would have an actual impact for example in terms of group budgets. Further, directly accessing other groups task list would be insecure in an untrusted environment. What could potentially also lead to some issues is the handling of error response codes (i.e. task deleted and winning bidder wants to patch, auction house cannot execute won task anymore, ...). However, this needs to be handled internally for each group (i.e. discarding external task, restart auction, ...)

007-Data Ownership

Context

Once we implemented the different services, we needed to also implement **tables** for the services to **log information** on previously executed tasks. We also need tables for our services to increase the overall **fault tolerance** of the system. The tables implementation must adhere to the bounded contexts between services.

Decision

We opted to implement the tables using mainly the **single ownership** technique.

Namely, the **executors pool** and the **auction house** will have a table of their own each.

For the **roster** and **task list**, we opted for the joint ownership technique. That means that both those services will be able to read and write in the same table.

Consequences

Using the single ownership technique for the executors pool and the auction house and the joint ownership technique for the roster and task list means we adhere to the bounded contexts defined before the start of the implementation.

The system is also highly decoupled. The only data those services share are the tasks themselves. Other than that, having a single table for each bounded context enables those services to read/write their own data on their separate tables.

One downside to that decision is that we will need more memory to ensure that all three tables can be implemented.

Another downside is that data consistency could be lost in cases where the system breaks. For example, if the roster breaks down right as a task is getting executed, the finished task might not get logged in the roster/task list table, while it will be logged in the executors pool table. That means that data synchronization might be required going forward.

008-Code Sharing

Context

Code reuse simplifies programming tasks and workload and is usually easy to handle in monolithic architectures. However, in the case of distributed systems code reuse becomes more difficult. Possible techniques include code replication, shared libraries, shared services and sidecars.

Decision

Operational parts like code for monitoring, tracing and logging is done via shared libraries. Few domain classes such as the task class are used in multiple services. These are replicated in each service. All other code does not rely on same code.

Consequences

- Code replication in multiple services keeps the bounded context and simplifies code sharing. However, the effort for code changes increases dramatically. Also there might be some inconsistencies across services.
- Shared third party libraries are easy to implement and keep effort for maintenance on a low level. Also changes are directly reflected in all services.
- Shared libraries could possibly lead to problems in keeping track of dependencies. However, as there are not that many microservices we believe the risk is manageable. Further, version deprecation and communication are potentially difficult. However, by relying on professionally managed third party libraries this cost is expected to be small.