# Architecture Decision Records

001 – Architectural Style finished Gian-Luca
002 – Service Granularity finished Gian-Luca
003 – 003-Asynchronous Task Execution
004 – Queueing Tasks in Executor Pool
005 – Auction House Integration
006 – Uniform HTTP API
007 – Data Ownership
008 – Code Sharing Gian-Luca
009 – Test Driven Development Gian-Luca
010 – Handling of Internal Tasks
011 – Updating Task Status
012 – Handling Auction House response 406
013 – Uniform Executor API

TAPAS 3 Group

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

# 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 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

# 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 favourable. For instance we could increase the computational resources allocated to a single service without affecting the others. Due to the statelessness of most of our logic we could even have multiple instances of certain services running simultaneously and thereby make use of horizontal scaling.
- Testability is also good. We are able to test every microservice individually by mocking input and output of other microservices. (We use Wiremock) This enables thorough integration testing of services.

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

- The implemented architecture is quite complex, especially the asynchronous communication can lead to complicated interaction patterns. Also, because we have several services, the entire process from the creation of a task to its completion will require quite a few requests between the services.
- We repeated ourselves quite a bit. I.e. there is a lot of boilerplate code for the operational aspects of every microservice (e.g. logging, testing). If we implemented a monolithic system we would probably end up with fewer lines of duplicated code.

## Learnings

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

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

# 002-Service Granularity

## Context

With the help of an event storming session, we identified the following domains:

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

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

Disintegrators:

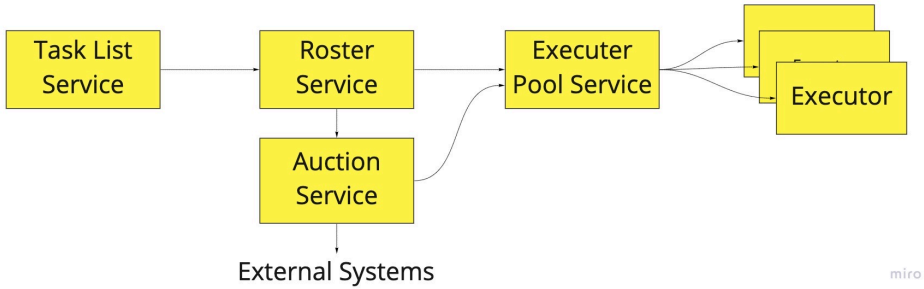
- **Service functionality:** All the identified domains have different purposes and will therefore encompass distinct functionality. According to Single Responsibility Principle, we should separate them into different services.
- **Scalability and throughput:** We assume that we would encounter most of the requests and operations in the task list and roster domain but only a few in the auction house, executors pool and executor's domain. However, this is strongly dependent on missing domain specific requirements and knowledge. If case there would be many auctions happening, this assumption may not be valid anymore.
- **Fault Tolerance**: Fault tolerance is one of the driving characteristics that we identified. Therefore, we could separate the services in such a way that sub-processes could still be executed by the relevant services in case other services fail.

Integrators:

- **Data relationships:** Both roster and task list should have consistent and identical data. Since tasks are central for the core operation, a single source of truth could be beneficial for indicating the current state of tasks.
- **Performance & Responsiveness:** Higher service granularity implies an increased number of requests and therefore higher latency. However, responsiveness of the whole system is not crucial since the executors vary in execution time. Thus, low latency is only necessary for the task list as a microservice, where the user directly interacts with the application.

## Decision

We decided to split up the services according to the five 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 more time to complete than others. We therefore expect the need for asynchronous communication especially between the executor pool and actual executors. In microservice based architecture, asynchronous communication is used to decouple systems, and to increase scalability and performance. We can benefit from these advantages by using the described service granularity.
**Consistency:** With this design we could end up with eventual consistency because the data is stored across multiple different services. Microservice based architecture is a distributed architecture, which enables 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 central mediator. However, it is also possible to build a central 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 |

# 002-Service Granularity

## Service Description

**Task List Service**
The task list service provides all functions that are directly used by our end user. These are mainly CRUD operations for task related information. Additionally, the task list service handles updates to the task objects (E.g., status and output data). The task list service forwards the tasks to the *Roster Service,* which is responsible for assigning the task.

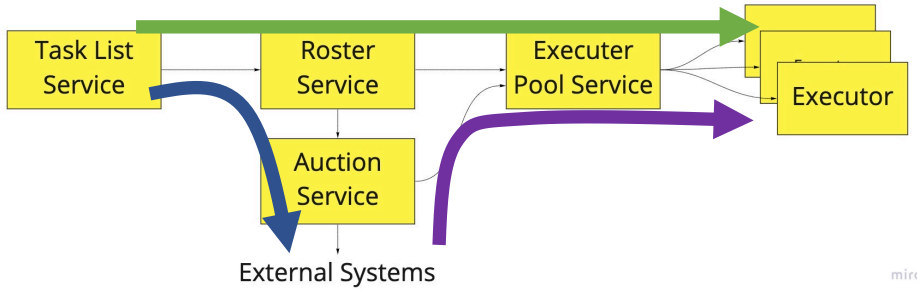
**Roster Service**
The main job of the roster service is to decide if a task should be executed internally or by an external executor via the action house. If the task can be internally executed, it forwards the task to the executer pool service. Otherwise, the task will be sent to the auction house. Hence, the main responsibility of the roster service is to orchestrate this assignment.

**Executor Pool Service**
The executor pool acts as an interface and bookkeeper for all the different internal executors. It keeps track of which internal executors are available or occupied, their capabilities and addresses. Additionally, the service handles the registration of new executors and the queueing and forwarding of tasks. If it receives a task, it will put it into a queue and forward it to the correct executor if one becomes available. The executor pool does receive both internal and external tasks.

**Auction Service**
The responsibility of the auction service is to manage all ingoing and outgoing auction requests. It therefore places bids if another organization publishes an auction and it creates an action itself, if the roster forwards a task to this service. If the action house won a bid, it communicates with the executor pool service to execute the external task.

**Executor Service(s)**
An actual task executor is one logical unit that can execute a task of a given type. 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 separation of executor pool and auction house services, led to a very natural division of responsibilities and functionality. The auction house became in charge for all auction related operations, while the executor pool is solely responsible for all interactions with the relevant executors. We never had an argument within the group about which functionality a certain service should be responsible for. This could be interpreted as a sign that the services were separated quite well.

The only thing that we would probably implement differently in hindsight is the roster service. The entire service only serves minor purpose and is solely responsible for determining if a newly created task should be handed over to the executor pool service or to the auction house service. Basically, the services functionality consists of just an if-else statement. For that simple logic we think that creating an entire microservice is a bit over-engineered. This adds more latency to the system and makes it less reliable and less understandable. 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 if we would have merged the task list and roster services into one microservice.

| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 003-Asynchronous Task Execution

## Context

Tasks take an undefined amount of time to execute, and some may take rather long. Synchronous communication in our system would imply that the executor pool service would be blocked every time it forwards a task to an executor and remains waiting for a response for an undefined, potentially long amount of time.

## Decision

To deal with this problem we decided to implement asynchronous communication between the roster, executor pool and executor services. That means that every time the executor pool accepts a task, it adds it to a queue (see ADR 004 for more details) and is free to accept more tasks from the roster immediately after. Furthermore, the executor services will immediately respond to the executor pool to indicate that a task has been accepted. After it's execution, the executor will independently notify the executor pool by sending a request to the corresponding endpoint.

## Consequences

Using asynchronous communication in the described manner has some important advantages:

1. With this communication method, our system can execute tasks despite long execution times without negative implications for new incoming tasks, since they will still be accepted by the executor pool.
2. Our system is more responsive, since the executor pool service will not be blocked during the execution of a task.
3. We do not need to return a task from the executor pool in case of the corresponding executor not being available anymore. Instead, we can queue the task until the executor becomes available and do not have to handle such a case.

But this method of communication also leads to two negative effects for our system:

1. The executor pool becomes more complex as it needs to keep track of all tasks in the queue, and update the tasks on the task list service every time it assigns a task, or a task is completed.
2. The error handling capabilities of our system decreases, because in the case our executor pool service crashes, we will lose the queue of tasks. We could handle this error by sending all open tasks that should be executed internally from the task list service once again if the executor pool service becomes available again. On the other side, the executors would need to resend a task completion notification to the executor pool.

| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 004-Queueing Tasks in Executor Pool

## Context

In our initial implementation design of our system, we observed a low utilization of resources if an Executor was occupied. Internal and External tasks would not be accepted by the Executors pool during that time period.

## Decision

To deal with that problem, we opted to implement a queueing mechanism into the executor pool service. This mechanism will accept tasks and queue them up according to the order of their arrival. This allows our Executors pool service to accept tasks when all the suitable executors are occupied.

## Consequences

This way we can adequately utilize our systems resources and make it more scalable by accepting a large number of tasks. Another advantage is that external tasks do not need to be returned if they cannot be executed immediately. Internal tasks will also be queued if not executed. Finally, there is less need for error handling since with the queue we keep a state of the task execution order even if another service crashes during runtime.

There are also some disadvantages that we identified. Namely, we don't know how long a task will take to execute because we don't know the size of the queue and the exact execution time of every task. We can deal with it by providing an estimate time of execution according to task type by the number of tasks with this type that are queued. Another downside of this queueing mechanism is that it adds some more complexity to the executor pool service and some tasks might be completed faster by creating an auction instead, if the queue is too long. If we would calculate the estimated completion time for a task, we could handle such cases in a more efficient manner and let the roster know that an auction would be a better option.

# 005-Task-Type Handling

| | |
|---|---|
| Status: | Accepted |
| Date: | 13/11/2021 |
| Decided by: | Team |

## Context

Define handling of distinct task types within 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. On the other hand, bids for other auctions should only be placed if the task type can be executed by an executor.

## Decision

Tasks are forwarded to the roster, which evaluates whether the task should be executed internally or externally. This evaluation is done by calling an endpoint on the executor pool with the task object, that indicates if a task of this task type can be executed. In case the task is not executable internally, the roster proceeds to hand over the task to the auction house. Similarly, the auction house will call the same endpoint whenever an external auction is published. This way, the auction house can make sure that the task type can be executed before placing a bid.

## Consequences

- Separation of concerns: Only the executor pool keeps track of the list of executors and enables querying by other services through this endpoint. Therefore, adding or removing executors does not need to be propagated to other services.
- More requests will be made since the this endpoint will be called with every newly created task or externally published auction. We could circumvent this for internal tasks by using response codes from the executor pool to indicate that a task type cannot be accepted. However, the auction house could not call the same endpoint, since it does not yet have the entire task object available before placing the bid.

| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 005-Handling of external tasks

## Context

Task that come from within our system (internal tasks) are sent from the task list service to the roster service and from there to the executor pool. However, at the auction house service we also receive tasks from other groups. We need to decide how those external-tasks are routed to the executor pool.

## Decision

Once we have won a bid and our auction house receives a task object, it forwards it directly to the executor pool service without involvement of the roster service. A request parameter that indicates that the tasks originates from an external organisation is sent along with the request. Upon completion of the task, the executor pool will distinguish if the task is external or internal and send the output data and status updates to the relevant task URI.

## Consequences

One advantage of this decision is that the direct routing of external tasks to the Executors pool keeps them separated from the internal tasks that are forwarded through the Roster.

Another advantage is that the same endpoint for execution can be used for both internal and external tasks.

We also identified two downsides with our decision. Namely, the executor pool service needs to keep track of the origin of tasks. This is done by retrieving the value from the provided request parameter and store it in an additional property of the task object.

The second downside is that due to the implementation of the queueing mechanism (defined in ADR 004), external tasks may take longer to be executed in comparison with other organisations depending on their handling. However, the tasks will not be returned by us in case no executor is immediately available, which prevents the need for an auction to be restarted by other organisations.

Additionally, we could also simply distinguish internal and external tasks based on the presence of the original task URI attribute of the task object. However, since this is also a user-defined parameter, we should not blindly trust it and rely our business logic on it.

| | |
|---|---|
| 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 rest of the tapas application is individual to each group and documented in ADR 005 for our case.

## 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 suitable executors, auction houses place a bid using the unified JSON schema bid (media type *bid+json*), which is composed of unique identifiers for the auction and information about the bidder. Upon expiry of the auction, the winning bidder receives the complete task object 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 will be required to update the task directly on the auctioneer's task list service using the relevant task URI. After acceptance, the bidder proceeds with the actual execution according to internally specified processes. As soon as the task is finished, the current state and output data of the auctioneer's task should be updated accordingly. 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 newly created auctions and then proceed with usual HTTP communication leads to simpler and already provided 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 since 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 the execution. This procedure keeps the number of preceding requests small, because 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. Handling received tasks individually, hides internal implementation from other groups and thus makes interoperability easier. The task class could be left as it is and only the values for two attributes need to be updated (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 by each group (i.e. discarding external task, restart auction, …).

# 007-Data Ownership

| | |
|---|---|
| Status: | Accepted |
| Date: | 21/11/2021 |
| Decided by: | Team |

## Context

Once we implemented the different services, we also needed a way for the services to store information about tasks, executors, auctions or whatever objects might be relevant for a service. By enabling a persistent storage for our services, we can also increase the overall fault tolerance of the system. The information stored by each service should adhere to the bounded contexts.

## Decision

We opted to implement the storages using the single ownership technique.

Namely, the executor pool will have a proprietary database where it keeps track of all executors, the task list service will have a database for all internal tasks and their current status, and finally, the auction house will keep track of all the auctions that were created in another separate database.

## Consequences

Using the single ownership technique for the executor pool service, the auction house service and the task list service, 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 which are being passed around between the services and only stored persistently by the task list service. Other than that, having a single database for each service enables them to read/write only the data that is relevant to them.

One downside to that decision is that we will need more resources to ensure that all three databases are running.

Another downside is that data consistency could be lost in cases where the system breaks. For example, if the executor pool breaks down right as a task is getting executed, the final update to the task might not be written to the task list database. Such cases could be handled by resending tasks whenever a service becomes available again.

Finally, complexity is added to the system, as three services will be performing read/write operations at any time during runtime.

# 008-Code Sharing

| | |
|---|---|
| Status: | Accepted |
| Date: | 21/11/2021 |
| Decided by: | Team |

## Context

Within our 6 microservices (tasks, roster, executor pool, auction house, executors 1&2) we have some duplicated code. There are two main types of duplicated code:

- Operational Aspects (Logging, Swagger,…)
- Domain Objects (Task object)

Because this code is duplicated 6 times it is sometimes cumbersome to change those shared aspects.

## Decision

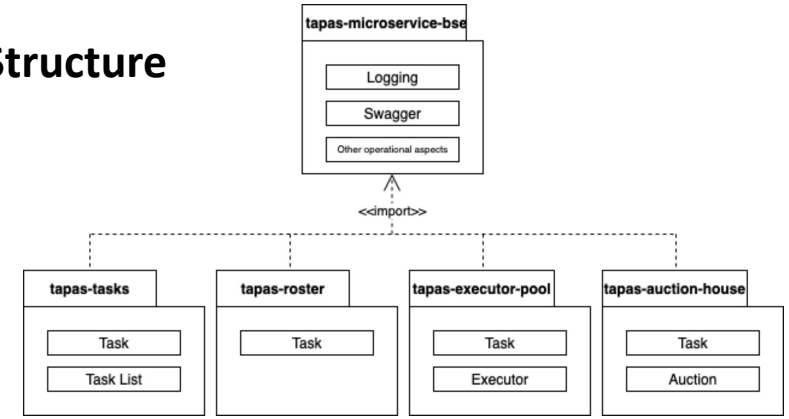
We create a shared library that contains operational parts like code for monitoring, tracing and logging. For those aspects we create a shared library that can be used by all microservices. However, we don't share any domain objects or other domain logic.

**Reasoning:**
Because Java Spring Boot is the strategic technology stack of the tapas system it make sense to use a shared library that acts as a base for all operational aspects. One alternative would also be to use a sidecar container for that, however, this would require us to implement a service mesh architecture which we don't plan to do. Also, with a spring library we can inject application logic directly into the microservices which would not be possible with a sidecar approach.
We don't share any domain objects because we believe that there should be a clear separation between our bounded-contexts. If we start sharing domain logic between microservices chances are high that we start building a distributed-monolith.
See: https://martinfowler.com/bliki/BoundedContext.html

## Consequences

- We need to write operational aspects of microservice only once. It is now possible for example to change to logging behaviour of all microservices by changing only the shared library. This keeps the effort for maintenance on a low level.
- 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.
- The task object model code is almost identical in every microservice. This makes it cumbersome to change any thing on the task object. However, we don't think that the task model will change very often.

## Packet Structure



# 009-Test Driven Development

| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

## Context

- The complexity of the tapas system is quite high
- We have multiple microservices that interact with each other via HTTP
- Some members of our team are experienced with test driven development
- The Tapas system should not be buggy

## Decision

We decided that we implement the Tapas system with a test-driven development approach although the teachers plan to introduce testing only at the end of the course. We focus on integration tests of the different microservices. We implement those in integration tests with Spring Tests (MockMVC), Mockito to stub repositories and Wiremock to stub HTTP APIs that the microservice depends on.

## Consequences

- We will be faster during development because we will catch many bugs early and our system will be less error prone.
- The development of a microservice is very convenient because we basically can just use our integration tests instead of manual smoke-testing.
- Because we focus on integration tests our tests will be mostly implementation agnostic. This means we can change the implementation without the need to also change the test.
- The build of our tapas system will take significantly longer because Spring Test are quite slow in execution.

Excerpt of a Spring Boot Test:

```
// ACT
mockMvc.perform(
        post( urlTemplate: "/roster/newtask/")
        .contentType("application/task+json")
        .content(requestBody))
        .andExpect(status().isOk());
```

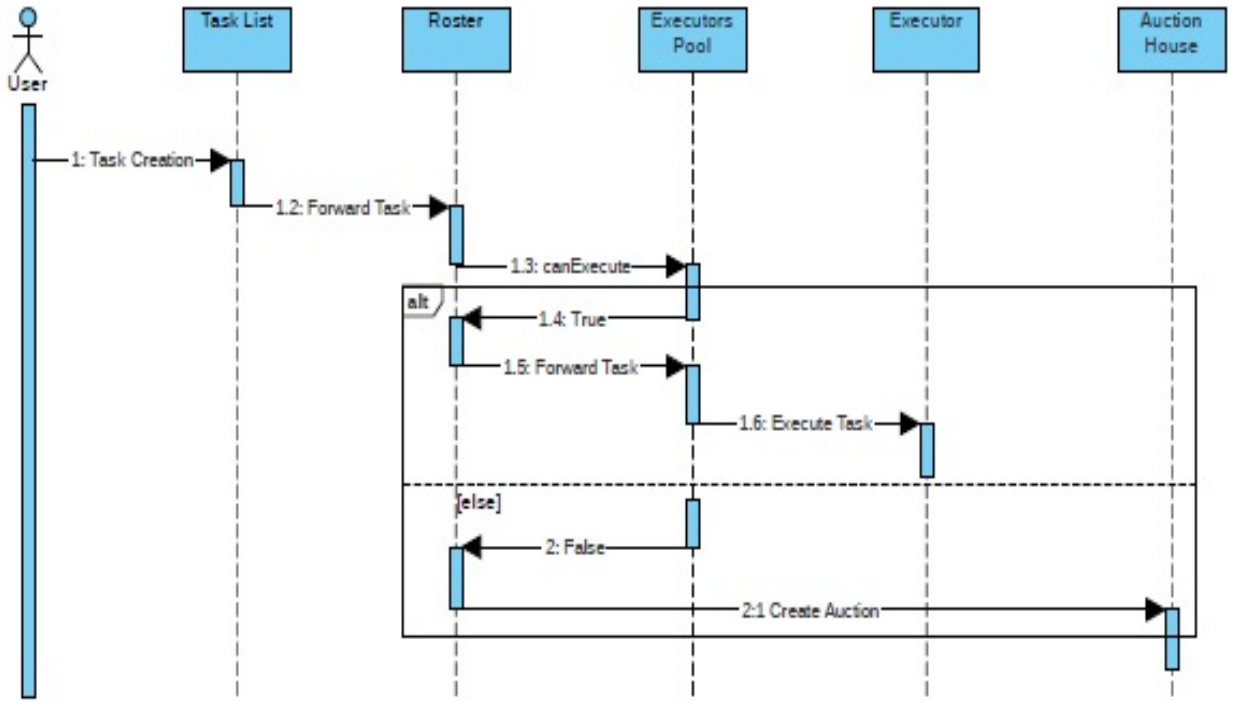
| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 010 – Handling of Internal Tasks

## Context

Definition of communication between internal services.

Whenever a task is created by a user it needs to be passed between services. We also need to check if we can execute the task internally or not, given our systems capabilities.




## Decision

Newly created task are added to Task List service, saved to the repository of that service and then forwarded to the Roster service. In order to check if the task can be executed internally or not, the Roster service calls the canExecute endpoint in Executors pool service. The Executors pool, which is the single source of knowledge for our Executors capabilities, checks if there exists a suitable Executor within our system to execute the given task. If that is the case, the task is forwarded to the Executor via the Executors pool. If we don't have a suitable Executor, the Roster service sends the task to the Auction house for an auction to be created and the execution to be assigned to the bid winner. All internal communications take place using http.

## Consequences

Routing of internal tasks through the Roster means we conform to our initial design choices for the system.

Calling the canExecute endpoint from the Executors pool also means that we have a single source of truth for our executors and their capabilities. This way, every service has a distinct functionality and conforms to the "single responsibility principle".

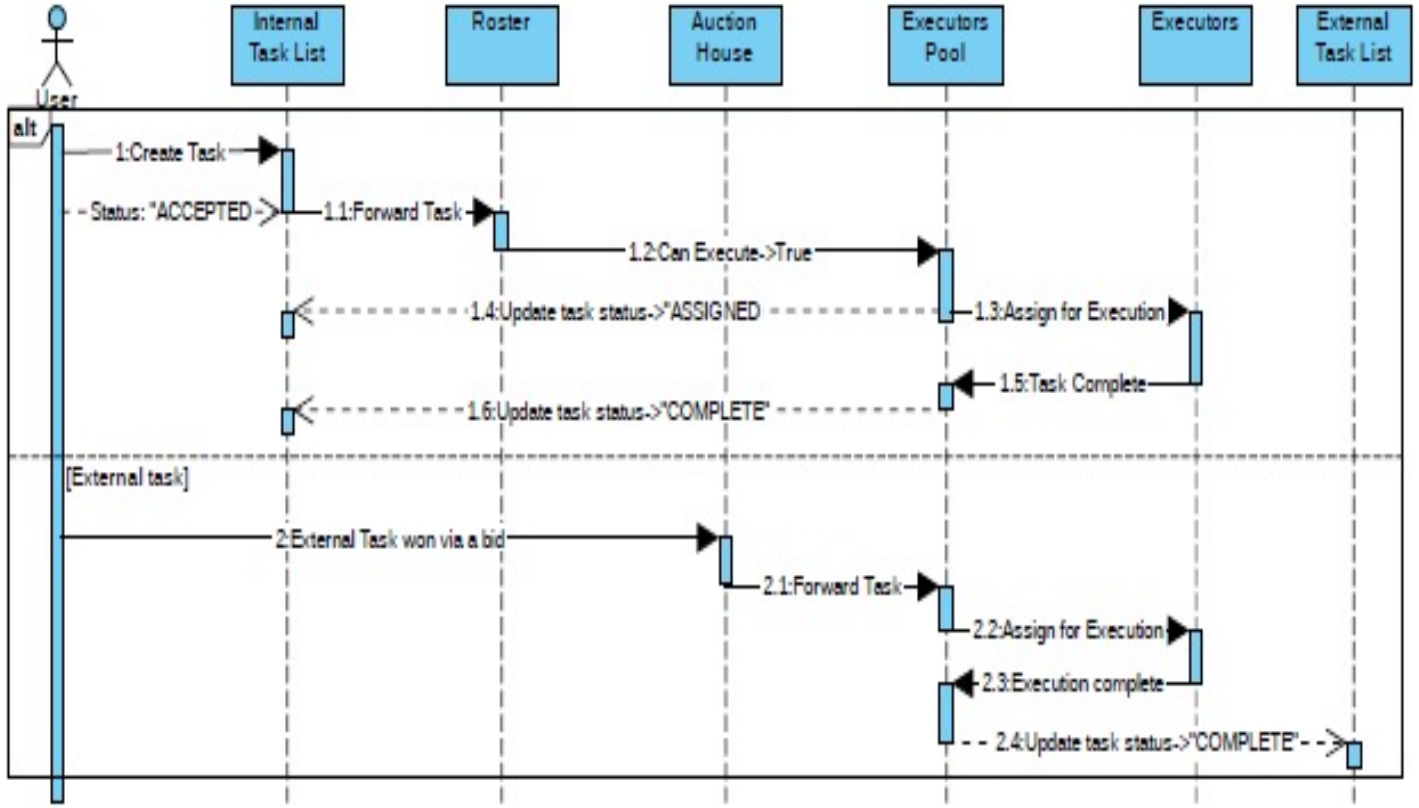
The only apparent disadvantage of this decision is added latency to the system, because the Roster has to retrieve the capabilities of the executors from the Executors pool for every task created. This could affect the response times of our system when there is a large influx of tasks.

| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 011 – Updating Task Status

## Context

Everytime tasks are created, assigned for execution and finished execution the task list needs to be updated. This procedure is different for internal and external tasks. For external tasks, the updating is defined in the uniform API documentation, but for internal tasks we needed to decide on how the tasks are updated in the Task list.




## Decision

- Once a task is successfully created by a user, the Task list service writes a new line in the table with all the attributes of the task and adds a status attribute, which at this stage is "ACCEPTED".
- When a task can be executed internally, as described in 010-Handling of Internal Tasks, it is forwarded to the Executors pool. Once an Executor is available to execute the task, the Executors pool assigns the task to that Executor and then through a patch command updates the tasks status to "ASSIGNED" directly in the Task list.
- Finally, when the Executor finishes executing the task, it backpropagates the output of the execution to the Executors pool. The Executors pool, again through a patch command, directly updates the status of the task to "COMPLETE".
- For external tasks that we need to execute, the status is updated directly to the Task list of the external organisation the task belongs to. Our Executors pool does the updating through a patch command to the given external Task list URI. The difference with internal tasks is that we only update their status once, when the execution is completed.

## Consequences

Updating the task status directly from the Executors pool means we update them as soon as their status has changed. We avoid any unnecessary delays and added complexity to our system by not back-propagating the task status information through the Roster back to the Task list.

This can have two disadvantages:

1. By not back-propagating task related information through the Roster, we limit its functionality.

2. Once the Executors pool has sent the patch to either the internal or the external Task list, there is no way or mechanism that informs the Executors pool in case the patch command fails or is interupted (by for example the Task list being not operational). In other words, there is no way for the Executors pool to know if the task status is updated, leading to a potential lack of eventual consistency.

| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 012 – Handling Auction House response 406

## Context

When our organisation creates an auction, and an external organisation wins this auction via a bid, we need to send them the task that was auctioned. But sometimes failures can occur when handing over the task. In case the task cannot be executed anymore, we should receive an HTTP status code 406 according to the documentation of the uniform interface API. We should properly handle such responses to make sure that the task will be executed.

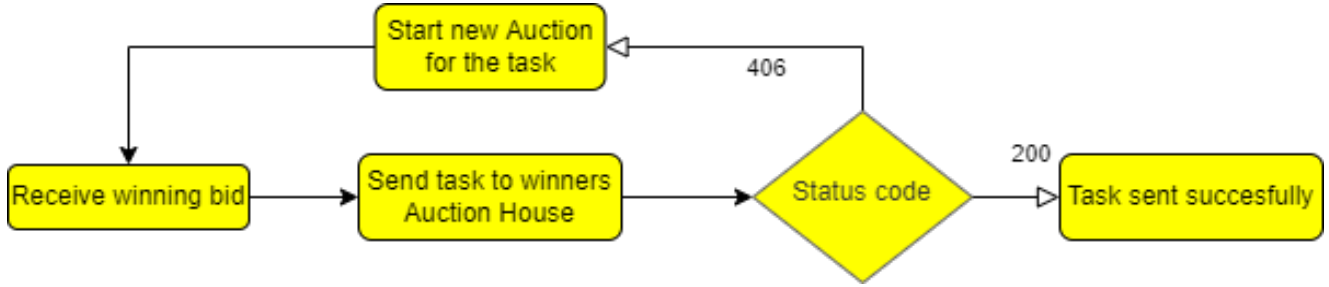
## Decision

To deal with the problem specified, we decided to implement a mechanism that launches a new auction for the task. More specifically, whenever the response code is 406 instead of 200, our Auction house will create a new auction object and accept new bids from other organisations.

## Consequences

We assume that the executor of the other organization was probably assigned to a task after the bid was placed and before the auction has ended. With this implementation we can provide an opportunity for retry to this organisation or other organisations.

By handling the response in such a way, it can be the case that multiple auctions will be launched for the same task. This implies that a delay can occur when a task is executed externally. We could avoid being stuck in an auction publishing loop by setting a fixed limit on how many times an auction can be restarted. Additionally, we could exclude the auction that provided the response from bidding on auctions of the same task. However, this should be dependent on the underlying business logic.



| | |
|---|---|
| Status: | Accepted |
| Date: | 25/11/2021 |
| Decided by: | Team |

# 013 – Uniform Executor API

## Context

At least two executors must be added to the executor pool with the possibility, that more executors of different types could be added at some point. When a tasks is assigned and sent to the executor, the receiver provides an immediate response to indicate that it has successfully received the task. As soon as the execution is completed, the executor sends a request back to the executor pool service to advertise the task completion. Any potential output data will be sent along with this request.

## Decision

To facilitate the addition of new executors as well as the communication between executor pool and executors, we agreed upon a uniform standard for communication. All executors are started using an identical "/execute/" endpoint, which consumes the task identifier and the task input data. When the executor receives a task, it immediately responds with a HTTP 202 status code. If the task is executed successfully, the executor calls the uniform "/completion/" endpoint of the executor pool and provides the task identified as well as any output data if necessary. Additionally, an endpoint is created to add executors to the executor pool at runtime.

## Consequences

- Simple to add new executors to the executor pool at runtime.
- Executor logic can be viewed as blackbox that processes input data and returns the corresponding output data.
- Potentially less flexibility when having more complex executors.