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
009 – Test Driven Development
010 – Handling of Internal Tasks
011 – Updating Task Status
012 – Handling Auction House response 406

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 Retroperspective

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

Retroperspective

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 increate 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). If 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 an 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

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

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:

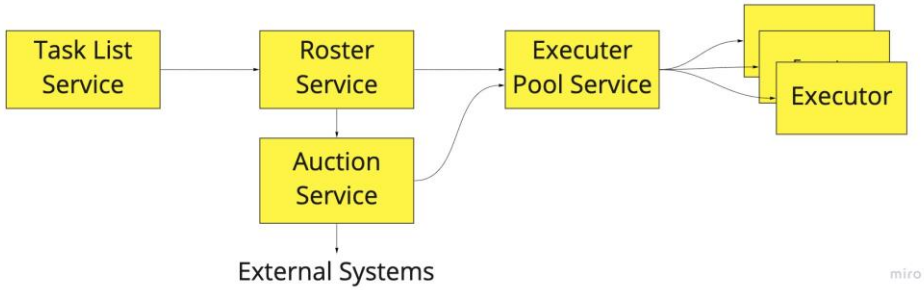
- **Service functionality:** The domains auction-house, executors, task list and roster have different functions. According to Single Responsibility Principle, we separate them into different services.
- **Scalability and throughput:** There would be many users and operations in the task list and roster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

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*.

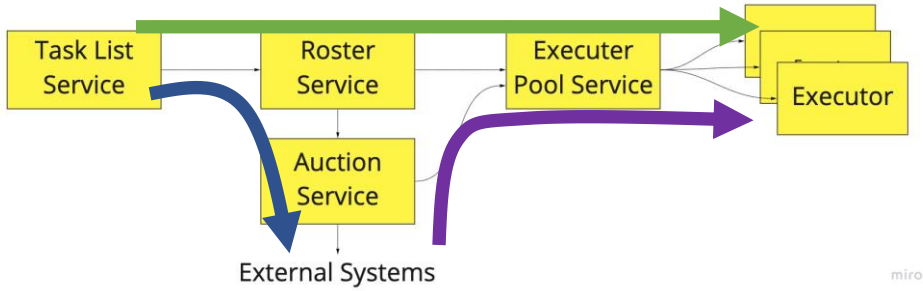
Roster Service
The main job of the roaster service is to decide if a task should be executed internally or by an external executor via the action house. If internally executed, it forwards the task to the executer pool service. Therefore, the main responsibility of the roster service is to orchestrate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

002-Service Granularity Retroperspective

Retroperspective

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 an 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 an if-else statement. For that simple logic we think that creating an own microservices is a bit over-engineered. Having an 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.

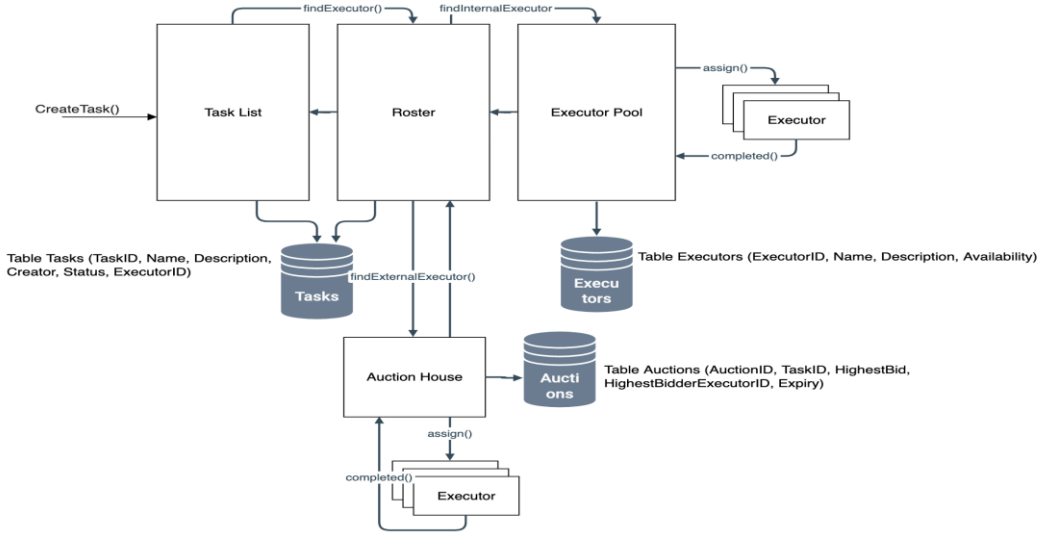
Status:	Accepted
Date:	25/11/2021
Decided by:	Team

003-Asynchronous Task Execution

Context

Tasks take an undefined amount of time to execute, and some may take too long. Synchronous communication in our system would mean that the Executors pool would be blocked every time it accepts a task for execution and for undefined, potentially long amount of time.

Click to add text



Decision

To deal with this problem we decided to implement asynchronous communication between the Roster, Executors and Executors pool. That means that every time the Executors pool accepts a task, it adds it to a queue (see next ADR for more details) and is free to accept more tasks from the Roster or assign them for execution.

Consequences

Using asynchronous in the way we do has two important advantages:

1. With this communication method, our system can execute tasks with long execution times without negative implications for new incoming tasks, as they are still accepted by the Executors pool.
2. Our system is more responsive, since it can handle a large amounts of tasks regardless of execution times.

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

1. Our system is more complex as it needs to keep track of tasks in the Executors pool queue, and patch the task list every time it accepts a task, or a task is completed.
2. The error handling capabilities of our system decrease, as in the case our Executors pool service crashes, we will lose the queue of tasks, while they are still updated in the Task list as "ASSIGNED". This can be dealt by patching the tasks status after it is assigned to an Executor.

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 when that was the case.

Decision

To deal with that problem, we opted to implement a queueing mechanism into the Executors 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 some 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 Executors pool service.

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

005-Task-Type Handling (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

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

005-Handling of external tasks

Context

Define communication between the Auction house and our system. When an external task is received, we need to forward it for execution and after its execution is complete, we need to return the output along with the status "COMPLETE" to the external organisation the task belongs to.

Decision

Once we have won a bid and our Auction house receives a task object, it forwards it directly to the Executors pool (doesn't go through the Roster). A request parameter that indicates the origin of the external task and the URI it needs to be returned to is also added to the task object and forwarded to the Executors pool, which handles both the execution and the return of the output and status to the external organisation.

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 after external and internal tasks arrive to the Executors pool.

We also identified two downsides with our decision. Namely, the Executors pool need to keep track of the origin of external tasks. This is done by adding a request parameter in the endpoint of the Executors pool alongside with a Boolean check for internal/external tasks.

The second downside is that due to the implementation of the queueing mechanism (defined in ADR 004), external tasks may take long amounts of time to executed, which is dictated by the state of the queue.

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

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

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 the single ownership technique.

Namely, the Executors pool will have a table where it keeps track of our executors, the Task list service will have a table where it keeps track of all the internal tasks and their current status, and the Auction house will have a table where it keeps track of all the auctions that we created or won.

Consequences

Using the single ownership technique for the Executors pool, the Auction house and the 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 enable those services to read/write their own data on their own 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.

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

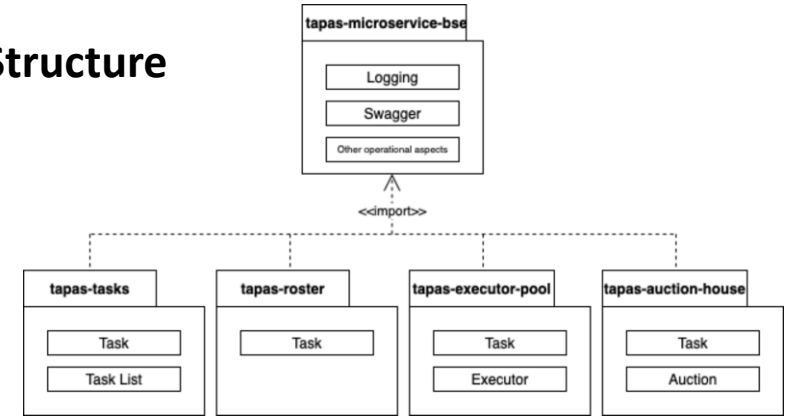
We create a shared library that contains operational parts like code for monitoring, tracing and logging. For those aspects we create a shared library that can be used by all microservices. However, we don't share any domain objects or other domain logic.

Reasoning:
Because we use 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 bounded-context. 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 because this would also require a change in the uniform auction house API.

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.

Excerp of a Spring Boot Test:

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

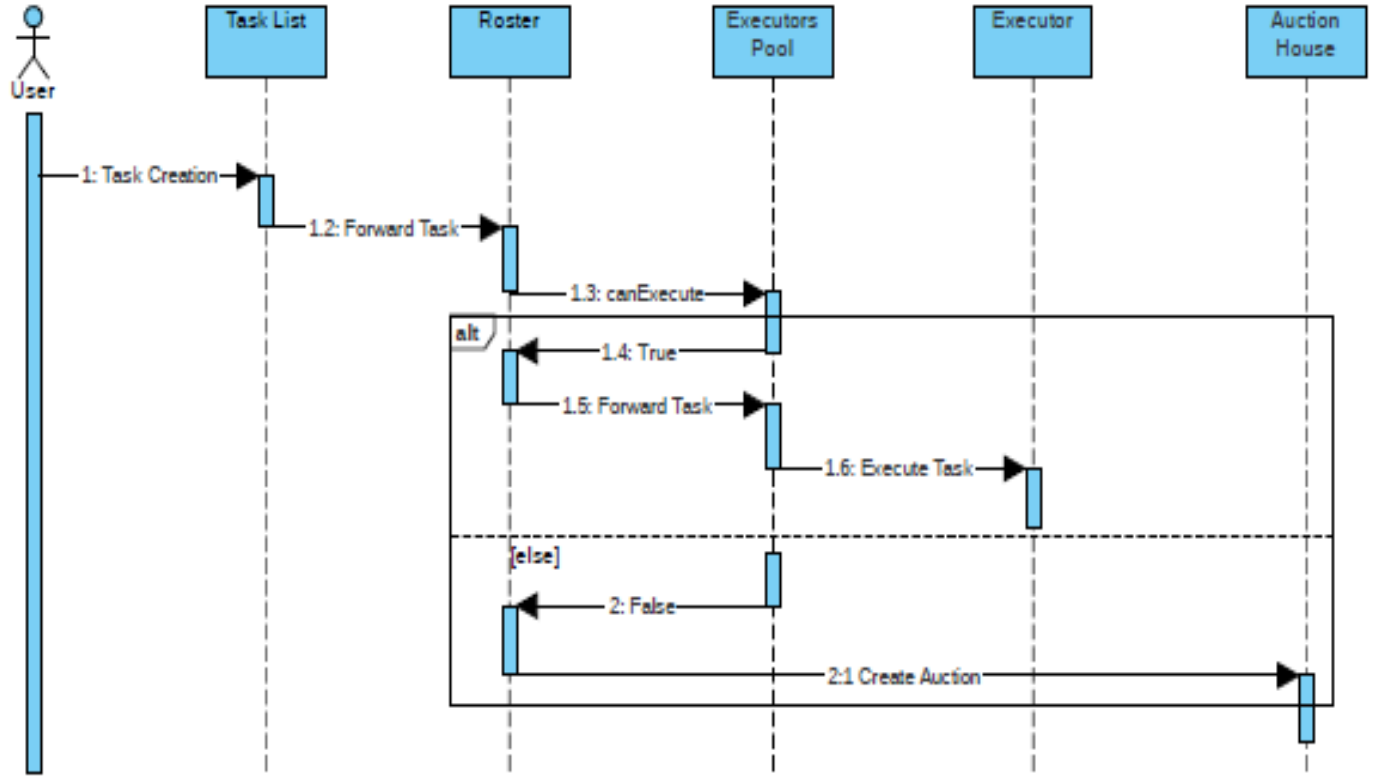
Status:	Accepted
Date:	25/11/2021
Decided by:	Team

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".

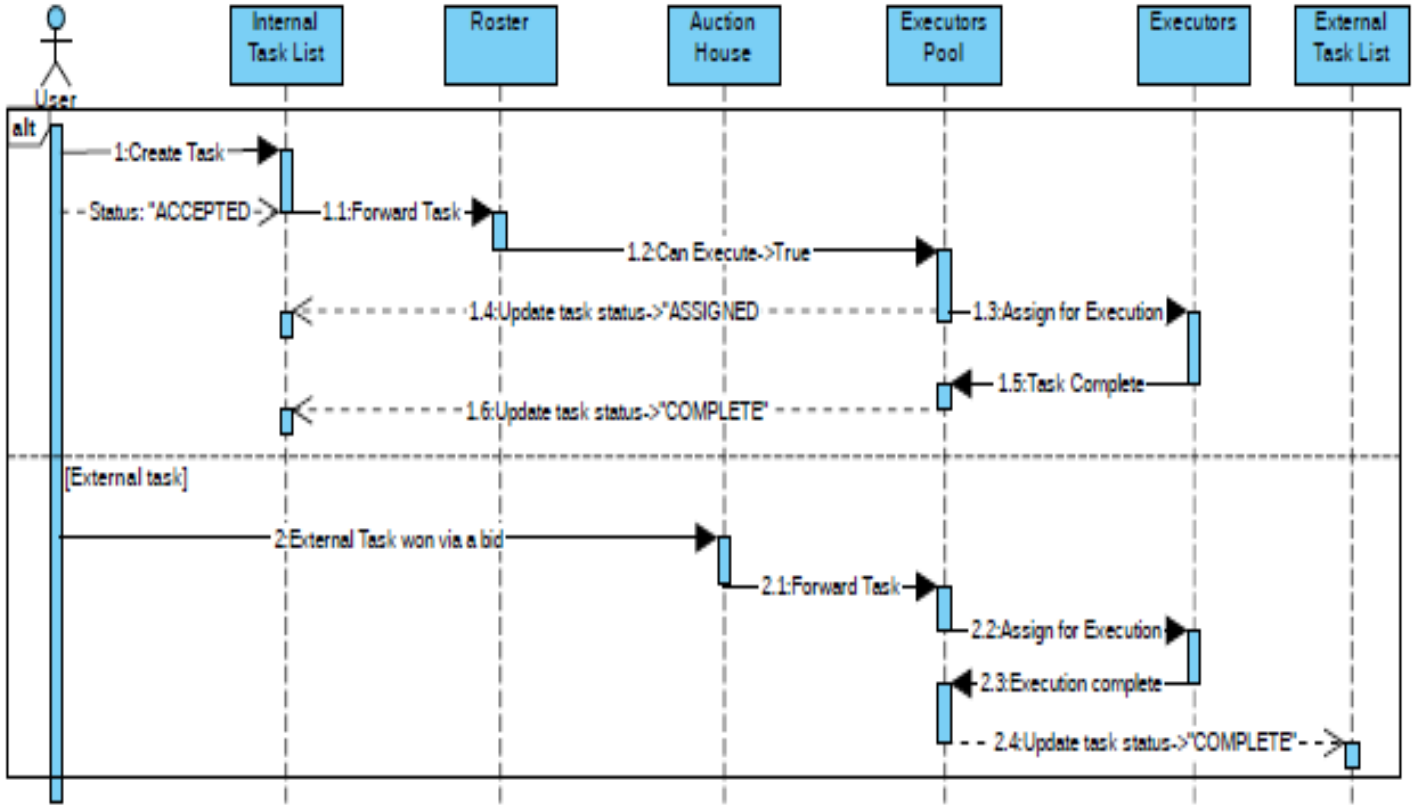
The only apparent disadvantage of this decision is added latency to the system, because the Roster has to retrieve the capabilities of the executors from the Executors pool for every task created. This could affect the response times of our system when there is a large influx of tasks.

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 during our sending of the task. In the case the task never arrives due to a failure, we receive an HTTP status code 406. In that case we need a mechanism that deals with such cases.

Decision

To deal with the problem specified, we decided to implement a mechanism that resends tasks to the winning organisations Auction house URI. More specifically, whenever the response code is 406 instead of 200, our Auction house will resend the auction task object until it receives a status code 200.

Consequences

With this mechanism we can make sure that the task that is assigned for external execution will always arrive to the Auction house of the organisation that won the bid.

A downside is that in extreme cases, where for example the external organisations Auction house is down for a long amount of time, it may also create delays in our system's Auction house. This mechanism also adds complexity our Auction house, but it is a needed tade-off as we need to handle 406 responses.