Record architecture decisions

Date: 2022-09-26

Status

Accepted

Context

We need to record the architectural decisions made on this project.

Decision

We will use Architecture Decision Records, as described by Michael Nygard.

Consequences

See Michael Nygard's article, linked above. For a lightweight ADR toolset, see Nat Pryce's adr-tools.

Implement as Microservice Architecture

Date: 2023-09-28

Status

Accepted

Context

In order to fulfill the application's requirements best, we need to evaluate and define a target architecture. TAPAS is an application that allows users to create tasks based on which the system searches for an appropriate executor in the executor pool.

The (revised) architecture characteristics which were selected are used to decide on a specific architectural style.

The 7 selected characteristics are: * Scalability * Elasticity * Modularity * Interoperability * Recoverability * Fault Tolerance * Evolvability

of which, Scalability, Elasticity, and Modularity are considered the top 3 driving characteristics.

Based on those characteristics, we select the following architecture styles for further consideration: * Microservices * Event-Driven

Decision

Both the Microservices, and Event-Driven architecture rank highly when compared to our selected architecture characteristics, mainly on elasticity, evolvability, fault tolerance and scalability.

However, the event-driven adds significant overhead due to asynchronous communication. Also, when compared to the Microservices architecture, the Event-Driven architecture is technically partitioned (vs. domain partitioned).

Hence, we decided to implement a Microservices architecture based on the above reason.

Consequences

By using the Microservices architecture, we are able to add changes to our system more flexibly (compared to other architectural styles) as we do not test the entire application (but only the affected architectures).

To allow for efficient (i.e., low latency) communication amongst the services, we will develop all components individually and create an API layer to handle client requests.

However, this will lead to higher initial overhead in the implementation of use cases.

Dedicated service for Task List Domain

Date: 2023-09-28

Status

Accepted

Context

In our TAPAS application, we use domain-partitioning to guide the overall structure.

In particular, we use the following 5 domains, which were established during the event storming at the beginning of the semester:

- Task List (ADR03) - current ADR
- Roster Domain (ADR04)
- Auction Domain (ADR05)
- Executor Domain (ADR06)
- Executor Pool Domain (ADR07)

The central requirement of the Task List Domain is the management of tasks, allowing users to modify the organization's task list (i.e., add or remove tasks) as needed.

Decision

We will implement the Task List Domain as a dedicated service within the TAPAS application. This decision is primarily informed by our consideration of Granularity Disintegrators and Integrators. According to our Event Triggers and Domain Events, there is a subcategory of requirements dedicated to receiving tasks from the user, storing these tasks, forwarding these tasks to the application, and communicating the results of these tasks to the user. As a result of this shared logic around Tasks, we decided to create a Task List Domain to integrate these actions surrounding Tasks.

This decision offers the following advantages and disadvantages:

Advantages:

- Isolating the Task List from other domains separates most Task logic from other services and allows for focused implementation of Task-related logic under a single service.

Disadvantages:

- Introducing a dedicated service may increase development complexity (e.g., interservice communication, data consistency). This makes ensuring coordination between microservices more difficult.

Consequences

Implemented as a dedicated service, the Task List service needs to communicate with other services (e.g., the Roster Service and Executors) to forward task input, receive task output, and update task status.

Dedicated service for Roster Domain

Date: 2023-09-28

Status

Accepted

Context

In our TAPAS application, we use domain-partitioning to guide the overall structure.

In particular, we use the following 5 domains, which were established during the event storming at the beginning of the semester:

- Task List (ADR03)
- Roster Domain (ADR04) - this ADR
- Auction Domain (ADR05)
- Executor Domain (ADR06)
- Executor Pool Domain (ADR07)

The central requirement of the Roster Domain is the assignment of tasks to the Executors. The Roster service is central as it coordinates amongst the Executors (Pool), Task List, and the Auction Domain. Once a task is created by the user, the Roster service assigns the task to an executor, either from the Executor Pool or the Auction.

Decision

We will implement the Roster Domain as a dedicated service within the TAPAS application. Given the user requirements, the Domain Events, and the Event Triggers we brainstormed, we decided to isolate all matching and assignment activities between Tasks and Executors to a single domain. This aligns with Granularity Disintegrators and Integrators. Because these activities do not fully fit into the Task or Executor Pool Domains. Isolating them to their own domain helps the executor and task logic to be isolated and hence the matching logic to exist separately as well.

This decision offers the following advantages and disadvantages:

Advantages:

- Focus on Coordination: Centralizing task-to-executor coordination simplifies this middleware functionality.

Disadvantages:

- Development Complexity: Introducing a dedicated service for matching and assignment logic increases initial development complexity (e.g., inter-service communication, data consistency)
- Service Centrality: Under the current architecture, the Roster communicates with all the other services and is at the center of the application. This lowers fault tolerance across the application as if this service has issues, it will impact the whole application.

Consequences

Implemented as a dedicated service, the Roster service needs to communicate with other services (e.g., Executor Pool, Task List, Auction) to assign tasks to executors and start implementation of tasks.

Dedicated service for Auction Domain

Date: 2023-09-27

Status

Accepted

Context

In our TAPAS application, we use domain-partitioning to guide the overall structure.

In particular, we use the following 5 domains, which were established during the event storming at the beginning of the semester:

- Task List (ADR03)
- Roster Domain (ADR04)
- Auction Domain (ADR05) - this ADR
- Executor Domain (ADR06)
- Executor Pool Domain (ADR07)

The central requirement of the Auction House is conducting auctions to find suitable external executors when internal capabilities are lacking.

Decision

We will implement the Auction Domain as a dedicated service within the TAPAS application. This decision is informed by our understanding of the user requirements, Domain events, and Event triggers. There is the need to interface with an external service in case our local system does not have enough resources to accomplish a task or in case we can provide some idle resources to external systems. Due to the complexity of communicating with external TAPAS applications, this logic is placed in its own Auction Domain to limit its impact on the rest of the internal application logic. We believe this aligns with the Granularity Integrators and Disintegrators by helping separate internal and external concerns.

This decision offers the following advantages and disadvantages:

Advantages:

- Separation of Concerns: Due to the added complexity of communicating with external systems. Having this logic isolated, allows us to develop our internal Task Execution logic and have the Auction Domain function as a transalator between our implementation and other external systems.

Disadvantages:

- Depending on the complexity of coordinating with external systems, the Auction Domain may have to handle a substantial amount of functionalities that may interfere with our current separation of concerns.

Consequences

Implemented as a dedicated service, the Auction service needs to communicate with other services (e.g., the Roster Service) to receive information about tasks, for which there were no internal executors.

Dedicated service for Executor Domain

Date: 2023-09-27

Status

Accepted

Context

In our TAPAS application, we use domain-partitioning to guide the overall structure.

In particular, we use the following 5 domains, which were established during the event storming at the beginning of the semester:

- Task List (ADR03)
- Roster Domain (ADR04)
- Auction Domain (ADR05)
- Executor Domain (ADR06) - this ADR
- Executor Pool Domain (ADR07)

The central requirement of the Executor is completing tasks assigned to it through the roster and communication to the task list the output of the task.

Decision

We will implement the Executor Domain as a dedicated service within the TAPAS application. Given our domain events and event triggers, we decided to isolate the execution logic of each task type to the Executor Domain. Each Executor will be a copy of the Executor Domain template. This allows us to conform to the Granularity Integrators and Disintegrators, by isolating and grouping together execution logic under one Domain.

This decision offers the following advantages and disadvantages:

Advantages:

Focused development on Execution logic leads to better executors as more attention can be placed on the logic of each task execution.

Disadvantages:

Interoperability requirements increase as the executors have to communicate with the Roster for task assignment, somehow with the Executor Pool to be kept track of, and with the task list to update users on the status of a given task.

Consequences

Implemented as a dedicated service, the Executor service needs to communicate with other services (e.g., the Roster Service, Task List) to receive task input and forward task output.

Dedicated service for Executor Pool Domain

Date: 2023-09-27

Status

Accepted

Context

In our TAPAS application, we use domain-partitioning to guide the overall structure.

In particular, we use the following 5 domains, which were established during the event storming at the beginning of the semester:

- Task List (ADR03)
- Roster Domain (ADR04)
- Auction Domain (ADR05)
- Executor Domain (ADR06)
- Executor Pool Domain (ADR07) - this ADR

The central requirement of the Executor Pool is the management of executors. The Executor Pool is designed to be dynamic, allowing for the addition or removal of Executors as necessary. It should store current executors in the application.

Decision

We will implement the Executor Pool Domain as a dedicated service within the TAPAS application. Given the domain events and event triggers we brainstormed, we decided to isolate all the logic that pertains to organizing tasks. In accordance with the Granularity Integrators and Disintegrators, this allows for all logic for keeping track of the system execution resources to be centralized and easily maintained.

This decision offers the following advantages and disadvantages:

Advantages:

- Isolated execution resource tracking and organization allows for better transparency and maintenance of executor resources in the application. Especially in the form of database implementation.

Disadvantages:

- Isolating these would result in added communication needs for other services to be aware of current executor resources.

Consequences

Implemented as a dedicated service, the Executor Pool service needs to communicate with other services (e.g., the Roster Service, Executors) to find executors based on requested capabilities.