Architectural Design

Taking into account the business requirements, my proposal is to implement a system based on **Event Sourcing**, following a **Domain Driven Design** and the **CQRS** pattern.

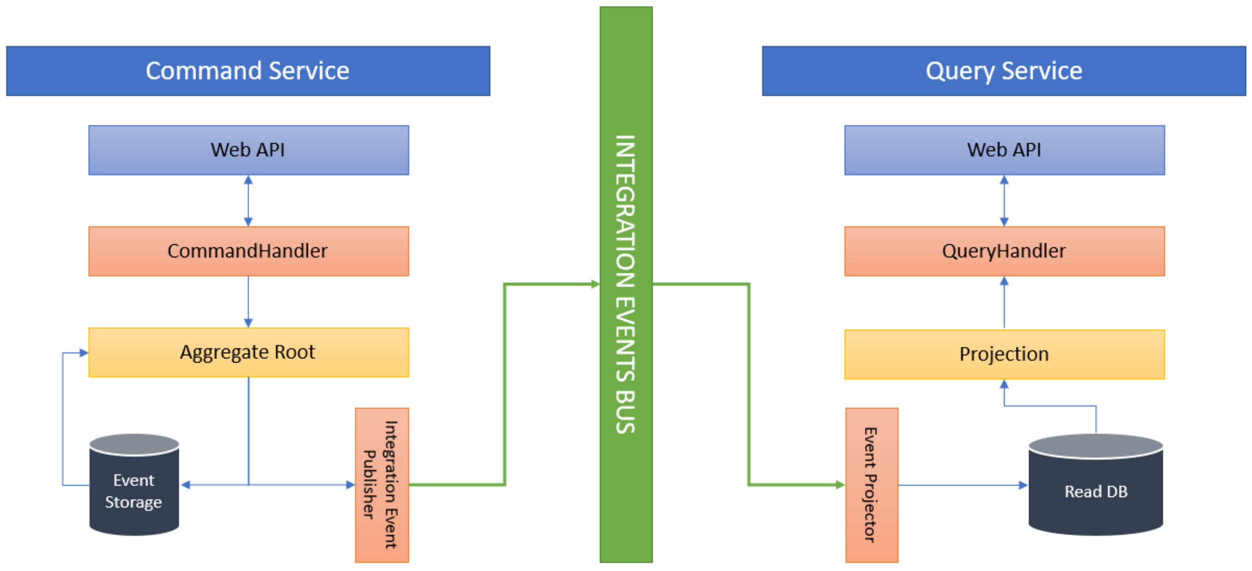
By using Event Sourcing, we are accepting that the only source of truth in our system are the domain events, that will represent any change on the state in our model.

These domain events are raised by our aggregate roots, which, by using the *Apply* method, would make the appropriate state changes over the entities/domain objects behind scenes.

Thanks to using DDD, we would keep all the domain logic encapsulated in the model (domain objects such as aggregate roots, entities, value objects), we would have a common and ubiquitous language shared with business, or a proper separation across domains and subdomains.

With CQRS we might have a strong separation between write and read operations, they would be totally decoupled and would bring a better performance and scalability.

All these patterns would coexist within a **Clean Architecture**, so inner layers would not depend on outer layers, and the Application layer would be agnostic to any specific 3rd party library or specific technology.



Command Service

- The purpose of this microservice is to handle write operations (commands).
- Before processing the command, firstly we need to get all the domain events related and create a new instance of the aggregate root - just to hydrate the aggregate root and have it updated to the correct state.
- Once we have the aggregate root ready, we can invoke the related method/s and let the aggregate root make the proper changes on the state.
- All the new domain events produced by the aggregate would be:
  - Stored in the Event Storage.
  - Mapped to an Integration Event and published into the message bus.

Query Service

- The purpose of this microservice is to handle read operations (queries).
- It will consume the relevant integration events and will project the data in a database.
- The information is stored in the database, and it's ready to be served when requested.

Some remarks:

- The Command Service can receive commands from a Web API, Message Bus or any other transport infrastructure.
- Since we are using Event Sourcing and the source of truth are the events, the Read DB can be dropped and then regenerated by replaying all the events.
- If an aggregate root has a high volume of events, we can use the snapshot approach as an optimization – instead of hydrating our aggregate with millions of events.
- At the **QueryService**, it is recommended to use a non-relational database for optimization purposes.
- It is important that our ***CommandHandlers*** are transactional, just to make sure we don't have inconsistency issues.

Real flow – Command handling:

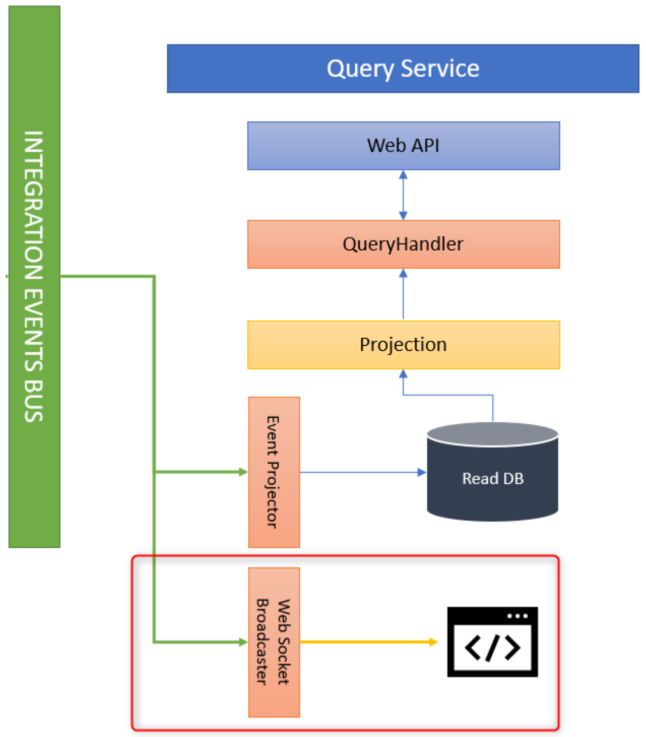
1. A new Command ***MoveRoverBackwardCommand*** is requested in the **Command Service**.
2. It retrieves all the domain events for the ***Grid***
3. It gets the following **domain events**:
    - *GridCreated*
    - *RoverCreated*
    - *RoverMovedForward*
    - *RoverMovedForward*
    - *RoverTurnedRight*
    - *RoverMovedForward*
4. It creates a new instance of the ***Grid*** aggregate, passing the list of the domain events as parameter. Then, the Grid invokes the *Apply* method for each event to update internally the state.
5. The method ***grid.MoveRoverBackward()*** is invoked**.**
6. The aggregate root produces a new event ***RoverMovedBackward*** and applies it.
7. The domain event is stored in the Event Storage.
8. The domain event is mapped to an integration event (since will be consumed outside) and it is published to the Message Bus.
9. The transaction ends, a ***CommandResult*** is returned.
10. The **Query Service** receives the integration event.
11. It checks if the sequence number of the event is the expected one – just to avoid event handling in the wrong order.
12. The event is projected in the database, updating the related rows/fields.

Real flow – Query handling:

1. A new query ***GetGridDetailsQuery*** is requested in the **Query Service**.
2. It gets the Grid with the information of the Rover, the current position and facing, etc.
3. Information is mapped to some Data Transfer Objects, following the agreed scheme.
4. Information is returned to client.

Live Updates

A possible approach to achieve the live updates in the front-end is to add a new Integration Event Consumer within the **Query Service** (like the ***Event Projector***) called ***Web Socket Broadcaster***, that consumes an integration event, gets some information from the Read DB (if needed) and sends a message to the Web Socket clients.


INTEGRATION EVENTS BUS
Query Service
Web API
QueryHandler
Projection
Event Projector
Read DB
Web Socket Broadcaster