ACME Parking System MVP Architecture Report

1. Overview

The ACME Parking System utilizes RabbitMQ as the central message exchange hub. A physical transponder reader sends the transponder ID to a RabbitMQ queue which the Gate Service listens to.

The **Gate Service** retrieves the transponder ID from the queue once a transponder ID has sent to the correspond RabbitMQ queue.

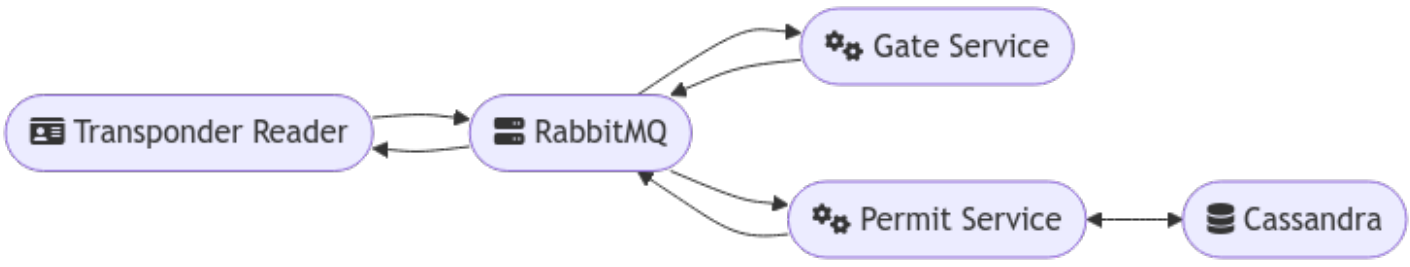
Then, the Gate Service sends a validation request to a RabbitMQ queue, which the **Permit Service** monitors.

Once receiving the request, the **Permit Service** interacts with the **Cassandra Database** (currently data stored in-memory, will be upgrade to Cassandra in the next phase) and sends the validation result back to a RabbitMQ queue monitored by the Gate Service.

Based on the validation result, the **Gate Service** sends a command to a RabbitMQ queue, which the physical transponder reader monitors, to indicate whether to open or close the gate.

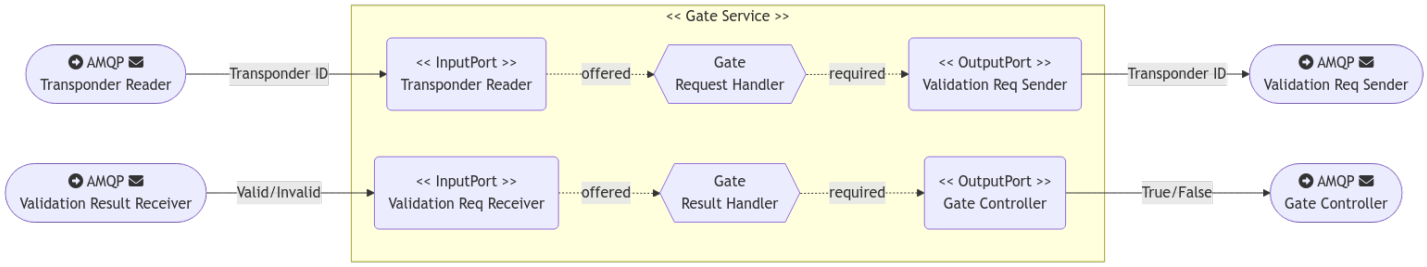
Note:

1. The transponder in the diagram below is a physical device that sends messages to RabbitMQ, not a service.
2. The Cassandra database is not implemented in the MVP; data is currently stored in memory.



2. Service Architecture & Design

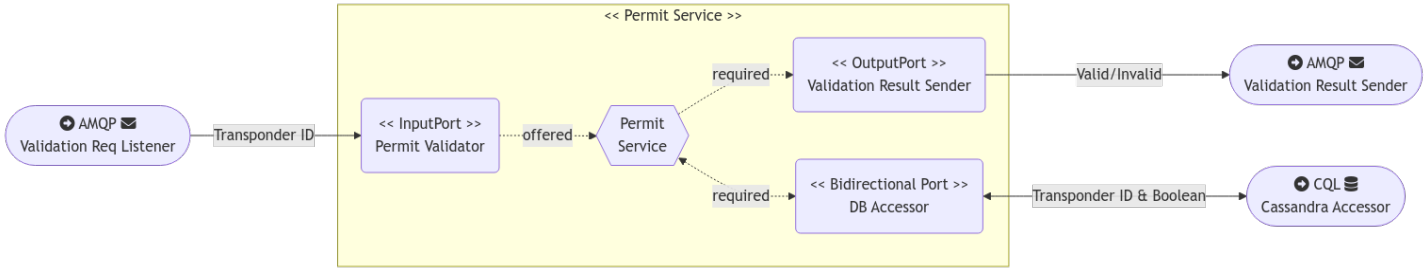
Gate Service Diagram



Note:

There are two business models in the Gate Service, which are **Gate Request Handler** and **Gate Result Handler**. Two business models do not interact with each other but each acts like an individual service (see Separation of Responsibilities).

Permit Service Diagram



Choice of Design

Hexagonal Architecture

The system is designed to follow hexagonal architecture with ports and adapters design patterns. Both the Gate Service and Permit Service have input and output ports as **interfaces** that isolate the interaction between services and external infrastructures. This allows more flexibility and scalability in code by replacing adapters without changing the main logic of the business module.

Ports & Adapters

Both Gate Service and Permit Service will only interact and listen to RabbitMQ, therefore both services have ports for universal connections and adapters for specific infrastructures usage (e.g. AMQP for RabbitMQ).

The reason that Gate Service does not directly build a connection with a physical gate/transponder device and listen for input (e.g. Send an HTTP request to Gate Service) is to ensure modularity and scalability although that will be a simpler solution. The protocol and physical devices might evolve in the future, therefore, leaving a port (i.e. Transponder Reader port) will be easier for future changes. Even though RabbitMQ needs to handle a bit more traffic this way, a parking system typically will not have tons of data needed to process at the same time.

Separation of Responsibilities

Each service only focuses on the functionality that lies in its responsibility which ensures each can be **functional independently**:

- The **Gate Service** only handles the logic and interaction with the transponder and gate control.
    - **Gate Request Handler** is responsible for processing the initial input and sending a validation request.
    - **Gate Result Handler** processes the validated result and determines if the gate should be open or stay closed.
- The **Permit Service** only handles the permit validation process and interact with database.

In this way, the Gate Service has two different pipelines that can decouple and asynchronously handle incoming requests and outgoing results.

Why DTO is not used?

Even though DTO can provide layer separation and improve data privacy and performance, the data transfers between the message broker and services are all simple strings with only one word. Therefore, DTO can lead to unnecessary overhead in this MVP and increase code complexity without a clear benefit.

3. System Advantages

Scalability (Horizontal)

Both Gate and Permit service is decouple from each other and asynchronously communicate with RabbitMQ. Therefore, they are encapsulated and deployed with docker container. That means a new instance of any of the two service can be launched easily though docker replication functions and automatically add to the swarm.

Similarly, both RabbitMQ and Cassandra can achieve horizontal scalability by adding more nodes to the cluster under the same docker networking configuration.

Availability and Fault Tolerance

RabbitMQ with multiple nodes can form a cluster that provides high availability and takes over the connections and requests when one node fails.

Cassandra cluster guarantees high data availability by fault tolerant system (see Disaster Recovery).

The system supports running multiple replications of the Gate and Permit Services with docker to ensure there is no single point of failure.

RabbitMQ and Docker built-in load balancer ensure the traffic will be automatically and dynamically redirected to other running containers and services to minimize downtime.

Disaster Recovery

Cassandra guarantees data durability with replication through consistent hashing and virtual nodes. In short, Cassandra will always have backup data available on different virtual nodes and can minimize data migration progress and evenly distribute data across the cluster with consistent hashing.

When services are not available, the messages still queue in the RabbitMQ and can keep processing once the services are online. If all RabbitMQ nodes fail, the persistent messages and durable queues can ensure the messages are written to the disk. Also, the acknowledge mechanism ensures that the message will not be popped until it has received an ACK from the service.