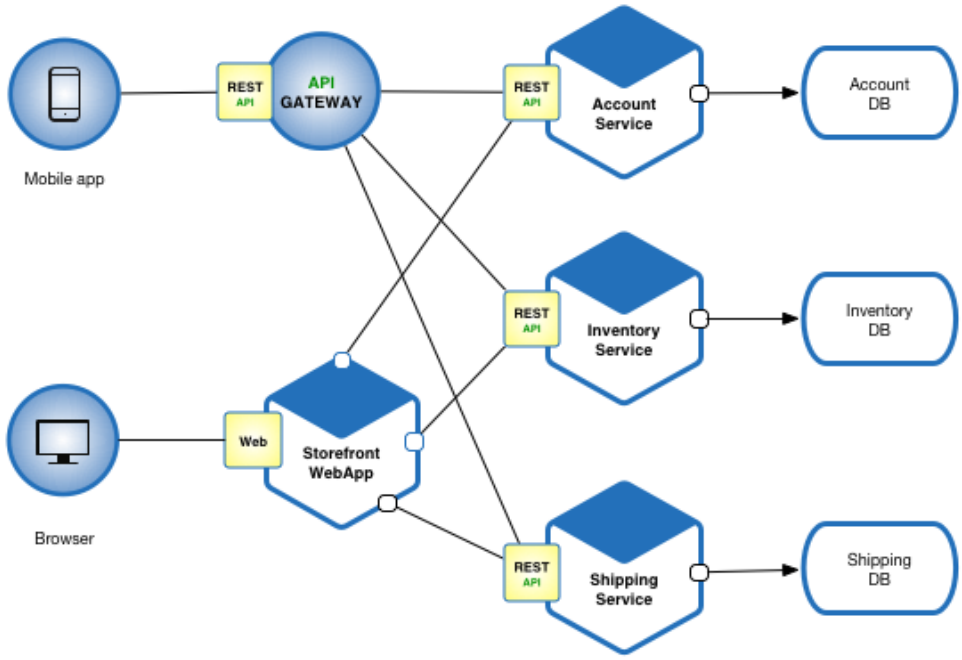
A: Microservices



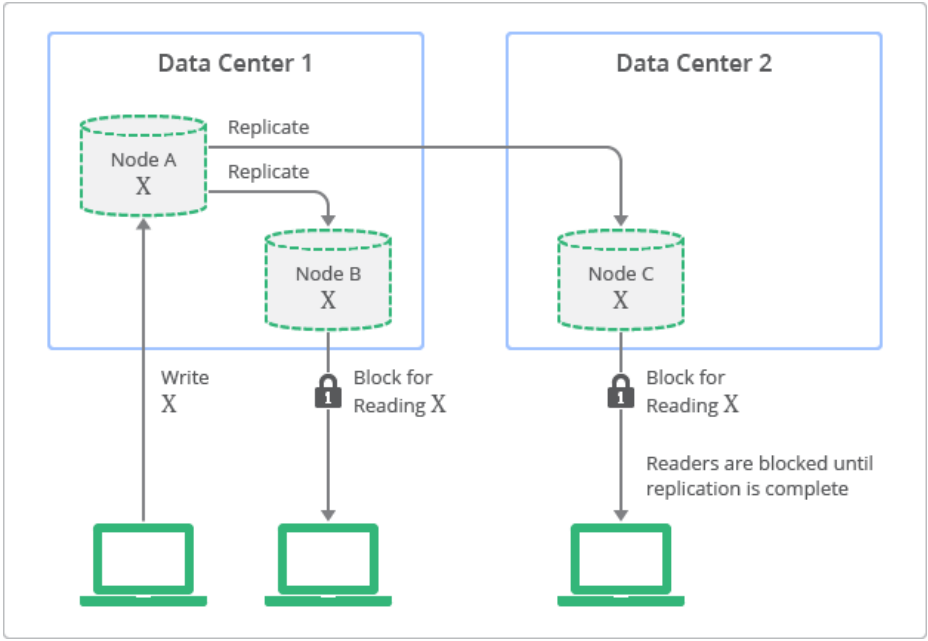
Each service is:

- Highly maintainable and testable - enables rapid and frequent development and deployment
- Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
- Independently deployable - enables a team to deploy their service without having to coordinate with other teams
- Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams

Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. Services can be developed and deployed independently of one another. Each service has its own database in order to be decoupled from other services.

Microservices was included in our case to provide excellent modularity of the differing services and providing a level of fault tolerance with independent down-time required for guaranteeing the most depended upon services the airport application must provide.

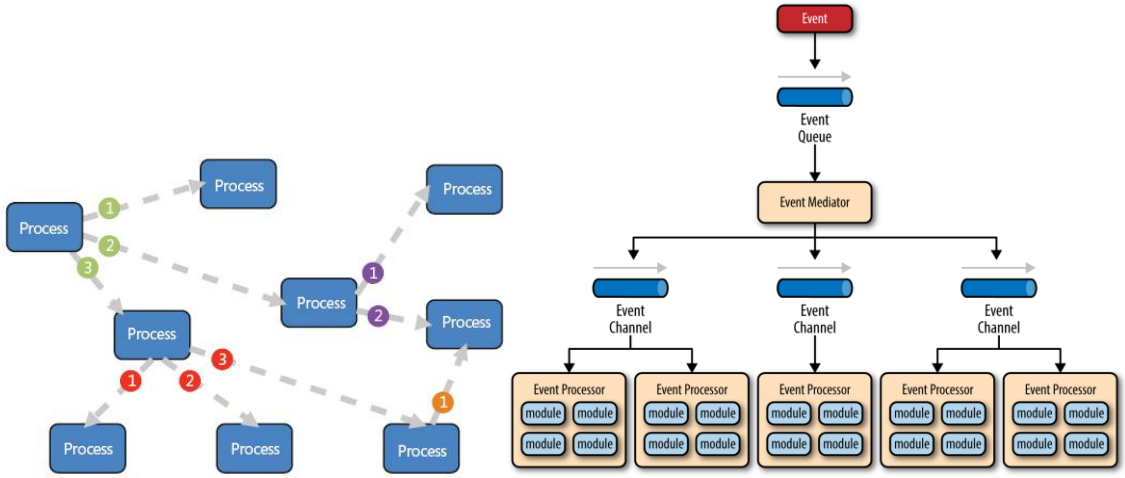
B: Eventual consistency



A distributed system maintains copies of its data on multiple machines in order to provide high availability and scalability. When an application makes a change to a data item on one machine, that change has to be propagated to the other replicas. Since the change propagation is not instantaneous, there's an interval of time during which some of the copies will have the most recent change, but others won't. In other words, the copies will be mutually inconsistent. However, the change will eventually be propagated to all the copies, and hence the term "eventual consistency". The term eventual consistency is simply an acknowledgement that there is an unbounded delay in propagating a change made on one machine to all the other copies. Eventual consistency is not meaningful or relevant in centralized (single copy) systems since there's no need for propagation. Various distributed systems address consistency in different ways because there's a tradeoff between speed, availability, and consistency. In some systems, the machine where the change originates will simply send asynchronous (and possibly unreliable) messages to the other machines and declare the operation as successful. This is fast, but at the cost of potential data loss if the originating machine fails before the replica(s)have received the update. Other systems send synchronous (blocking) messages to all other machines, receive acknowledgements, and only then, declare the operation as successful. These systems favor consistency and availability at the cost of performance. Finally, a system might implement some variant of these two extremes (e.g. wait for acknowledgements from a majority of the replicas).

Eventual consistency is useful in an application where all the different services are distant from each other and operate apart from each other. Using events, the central data storage can be updated according to generated events later to not impair the functionality of the microservices.

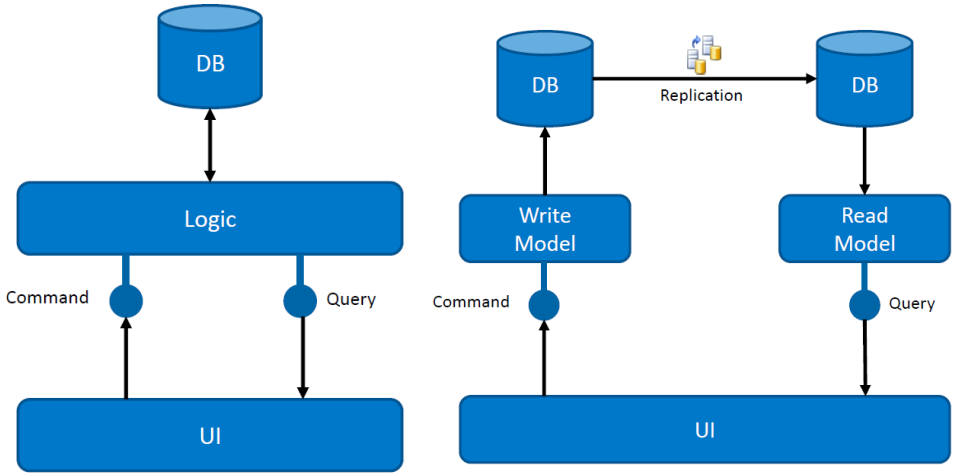
C: Event driven architecture



This architectural pattern may be applied by the design and implementation of applications and systems that transmit events among loosely coupled software components and services. An event-driven system typically consists of event emitters (or agents), event consumers (or sinks), and event channels. Emitters have the responsibility to detect, gather, and transfer events. An Event Emitter does not know the consumers of the event, it does not even know if a consumer exists, and in case it exists, it does not know how the event is used or further processed. Sinks have the responsibility of applying a reaction as soon as an event is presented. The reaction might or might not be completely provided by the sink itself. For instance, the sink might just have the responsibility to filter, transform and forward the event to another component or it might provide a self-contained reaction to such event. Event channels are conduits in which events are transmitted from event emitters to event consumers. The knowledge of the correct distribution of events is exclusively present within the event channel.

In the airport application it is imperative that all different targets are notified as soon as possible of the latest changes within the application. Therefor implementing event driven architecture patterns based on messaging is a useful solution to the problem. This way, the different microservices can get dynamically notified of the different changes within other context of the application.

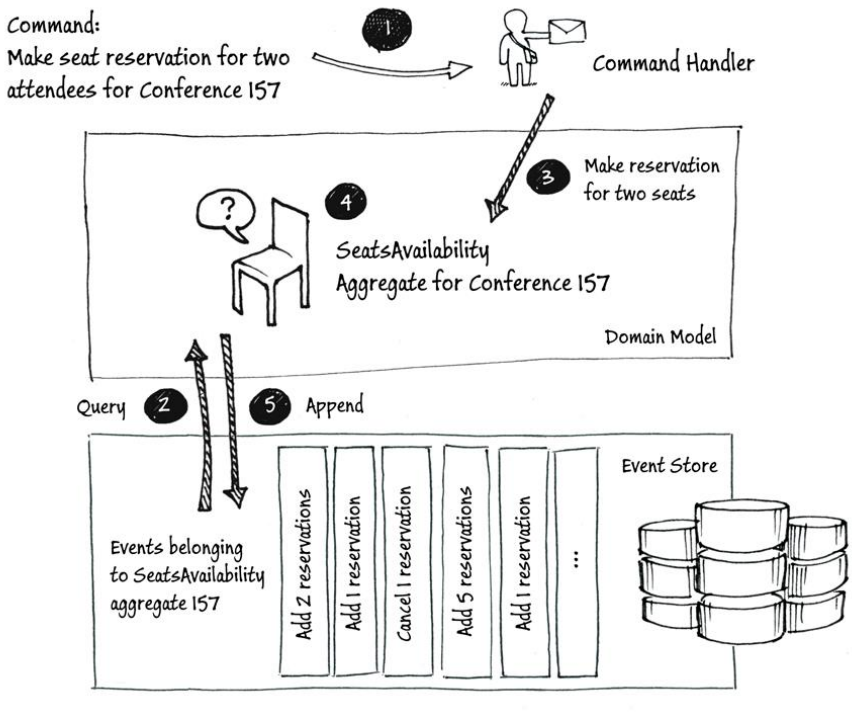
D: CQRS



In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy. For example, on the read side, the application may perform many different queries, returning data transfer objects (DTOs) with different shapes. Object mapping can become complicated. On the write side, the model may implement complex validation and business logic. As a result, you can end up with an overly complex model that does too much. Read and write workloads are often asymmetrical, with very different performance and scale requirements.

To ensure that different queries will provide the data that is requested by the part of the application, using CQRS is the ideal solution to the problem of interconnecting the different services the airport application uses.

E: Event sourcing



Event sourcing is a great way to atomically update state and publish events. The traditional way to persist an entity is to save its current state. Event sourcing uses a radically different, event-centric approach to persistence. A business object is persisted by storing a sequence of state changing events. Whenever an object's state changes, a new event is appended to the sequence of events. Since that is one operation it is inherently atomic. A entity's current state is reconstructed by replaying its events.

Implementing event sourcing is useful for guaranteeing that different access requests are logged for further audits, this might be useful for the context of a flight tower auditing the historical flight data and making further plans based on already available data through event sourcing (and therefor logging).