Project Architecture Document

for

PostOffice

Version 1.0

Prepared by

Ihor Huralskyy, Taras Kizlo, Mariia Kunyk, Nazarii Tymtsiv

27 March, 2021

Table of Contents

Architecture

The PostOffice application supports division between client and server using **Angular** and **.Net Core** in each side correspondingly. A **single-page application (SPA)** is used to make the website feel more like a native app.

The architecture proposes a **monolith** approach with elaborate **DDD/CQRS** patterns.

WebSocket is the communication protocol between client apps and server. It has been used for synchronous and asynchronous communication as well.

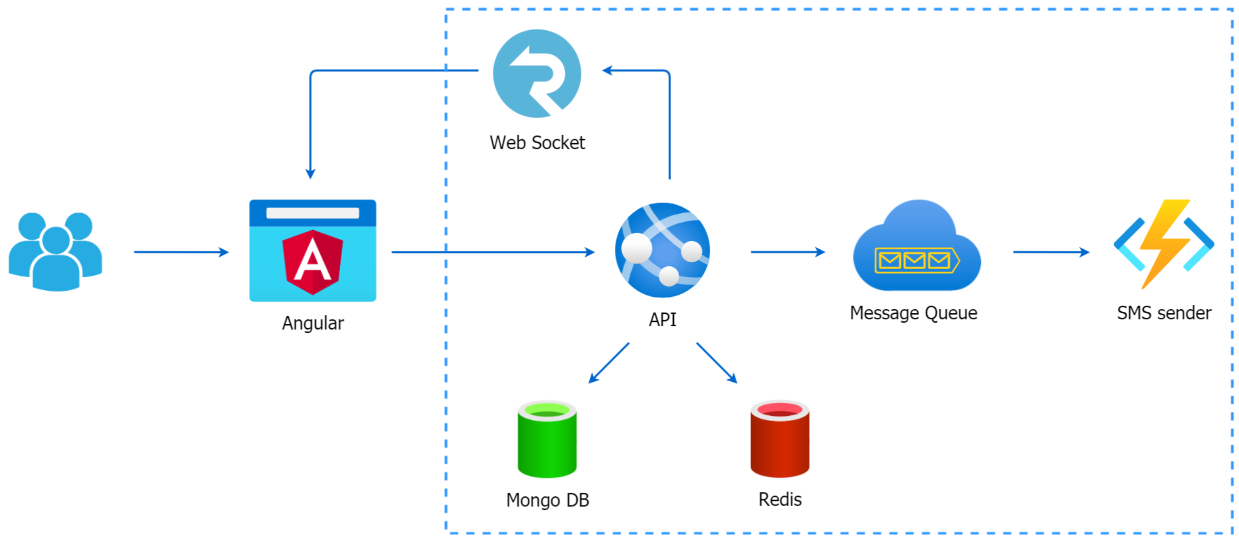
MongoDb is used as a main source of data to store aggregates and domain events. Idempotency and distributed locking is maintained by **Redis**.

Domain events are handled in the server itself, by using MediatR, a simple in-process implementation the **Mediator** pattern. Cross-cutting concerns (logs, transaction etc) are handled by decorators.

Message queues are based on Azure queues, to convey integration events.

Sms sender is third party tool that notify clients and is triggered by messages in queue.

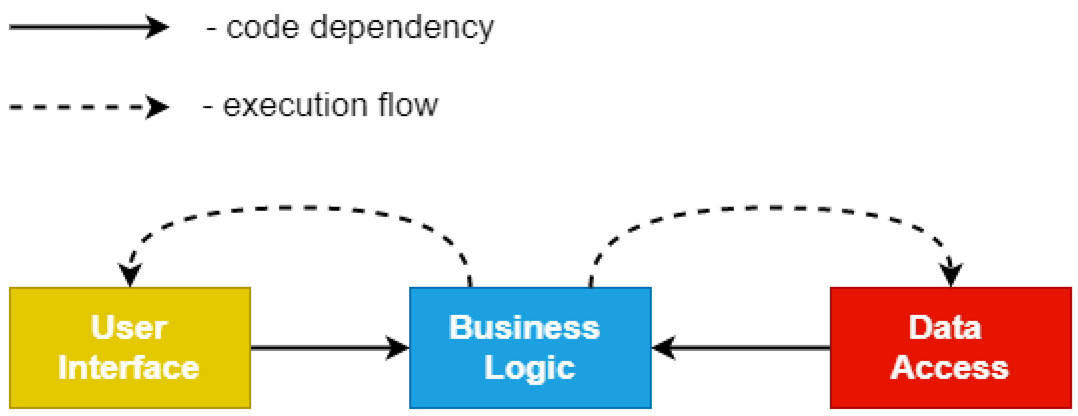
Architecture:



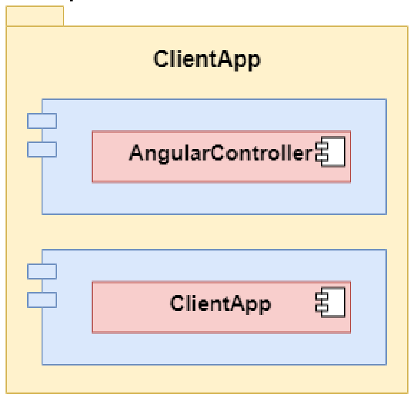
This image in good quality can be reviewed [here].

Component diagram

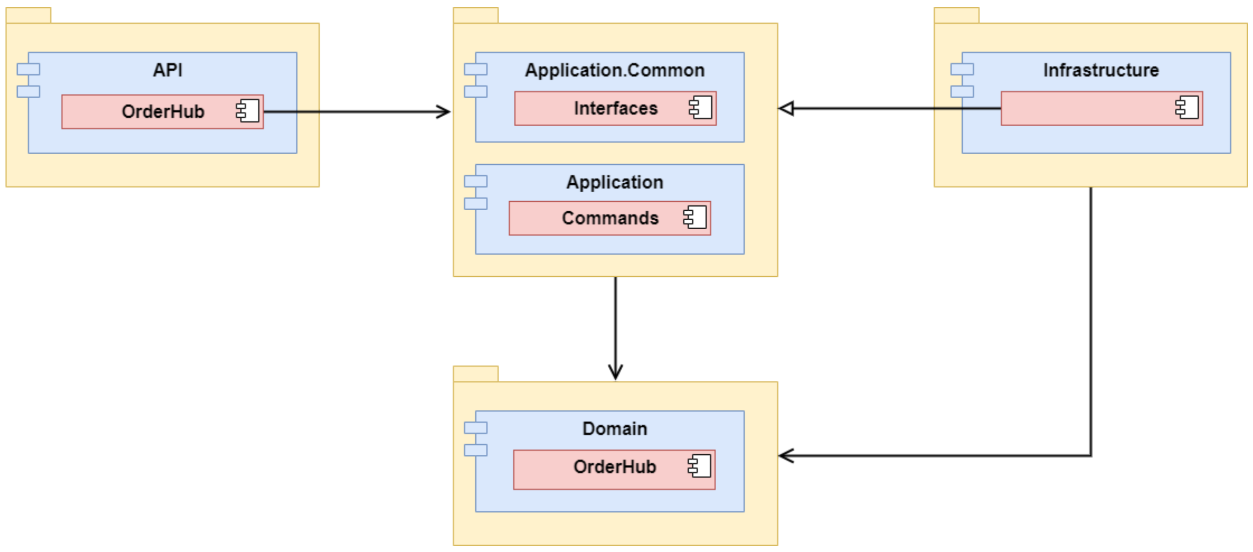
Current project use **Onion architecture**. This way allows us to deploy components independently of each other and reduce the amount of changes to rigid modules. Although the business logic layer use infrastructure and user interface layers it does not have direct dependencies on those. That is where we get use of dependency inversion principle and separated interface pattern.



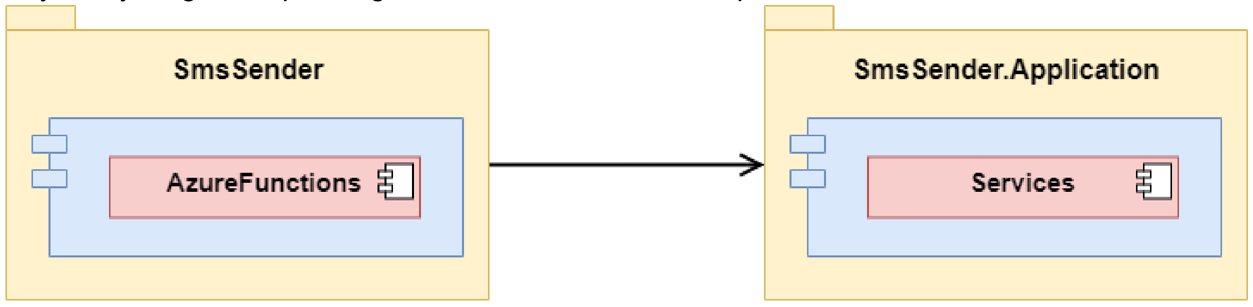
Let's dive into each module's structure. All modules are split by vertical slices. The First one is **ClientApp**. It contains all required dependencies and describe its domain area.



The Main module is more complex that is why it uses ideas of **ports and adapters**. The domain code is separated from the technical details and implementation. Application module declare interfaces that is implemented in infrastructure, so it can be independent as well.



For **SmsSender** logic is separated from user interface but still deployed as a single unit. This way everything is coupled together and still remains independent of environment.



Class diagram

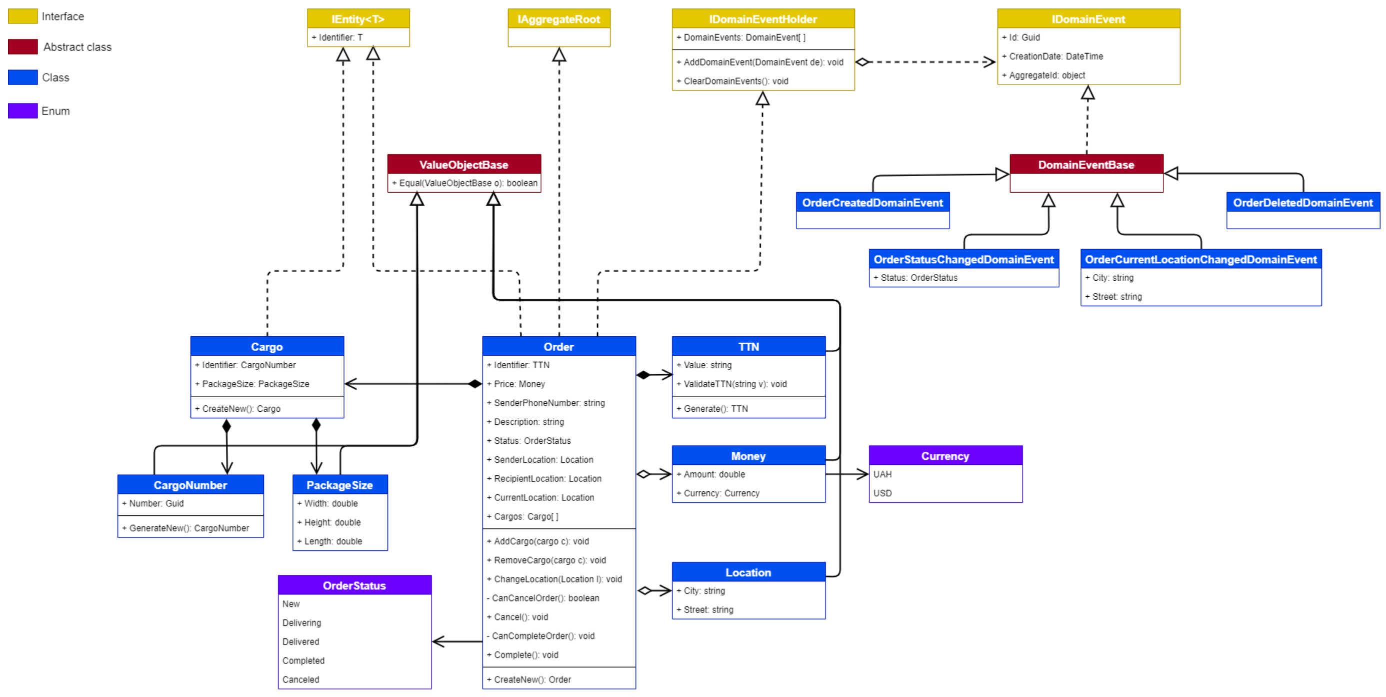
This section contains a class diagram of the PostOffice domain.

The main entity is **Order** as it is a transaction scope union for another entity (**Cargo**). That is why it is marked as **AggregateRoot**.

Order also produce a couple of domain events: *OrderCreated*, *OrderStatusChanged*, *OrderCurrentLocationChanged* and *OrderDeleted*.

There are other building blocks that inherit from **ValueObjectBase**. They don't have an identity and only serve for simplifying developing process by keeping models as close as possible to real world objects.

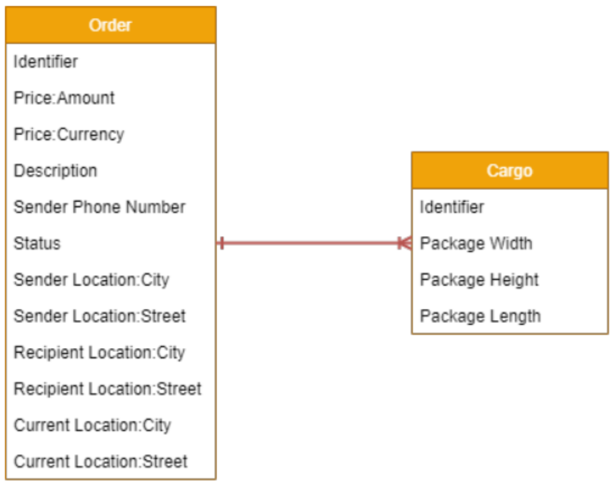
Class diagram:



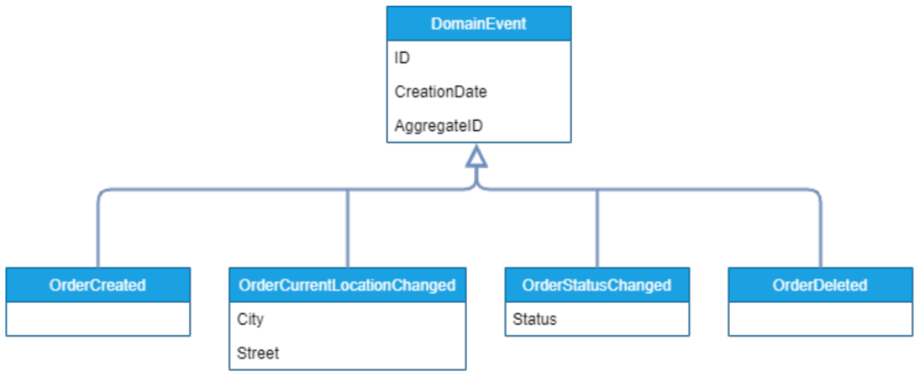
This class diagram image in good quality can be reviewed here.

Entity Relationship Diagrams

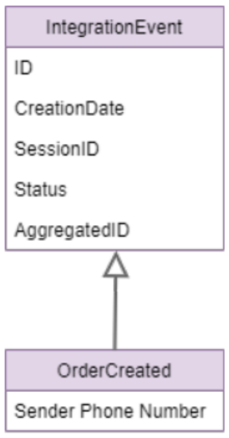
Our project has two entities: Order and Cargo. The relationship between them is one to many(one Order has many Cargoes).



Aggregation state is built from domain events. In this way we store order snapshots only when they are created and reply to events on every fetch.



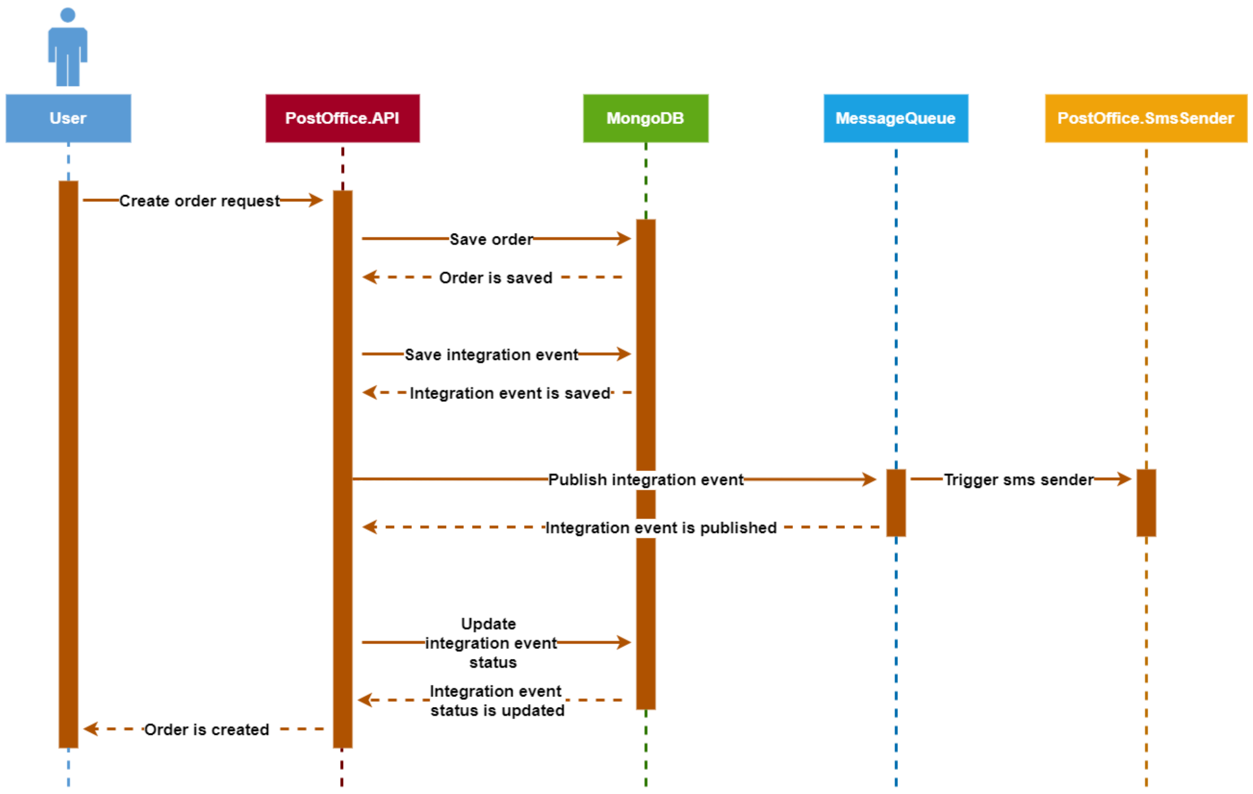
Integration events are used to maintain data consistency. They are logged before sending with “InProgress” status and if the transaction will commit successfully the status will be updated to “Published”.



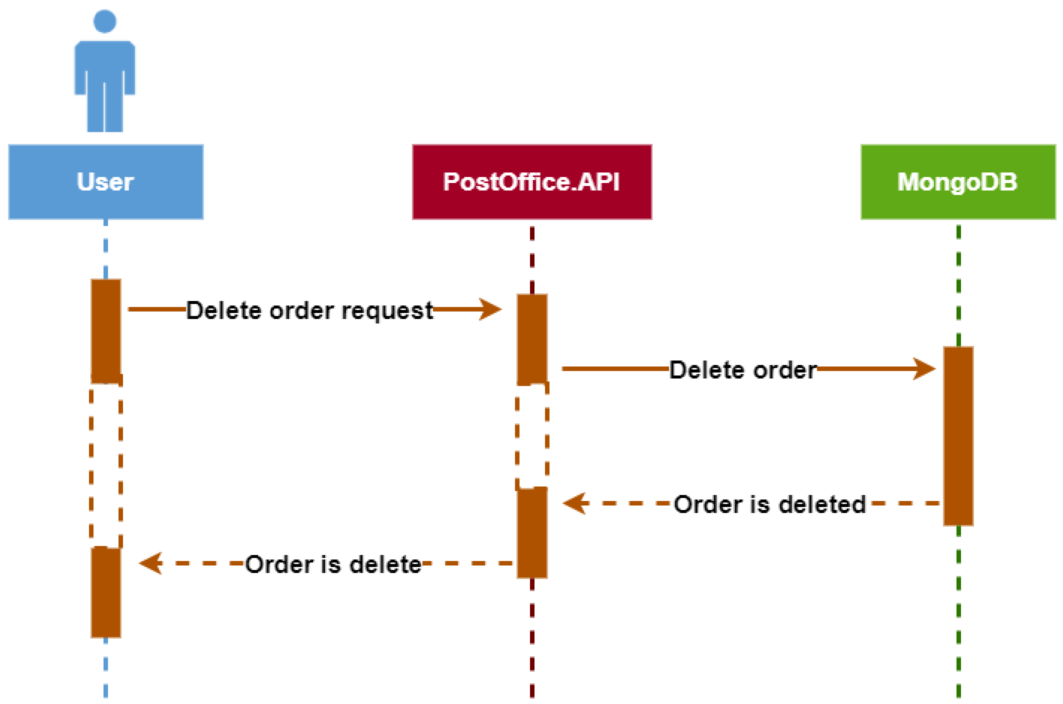
Sequence Diagrams

This page covers the interaction between different components in PostOffice. Since the majority of components behave the same way we will only discuss the ones what does not.

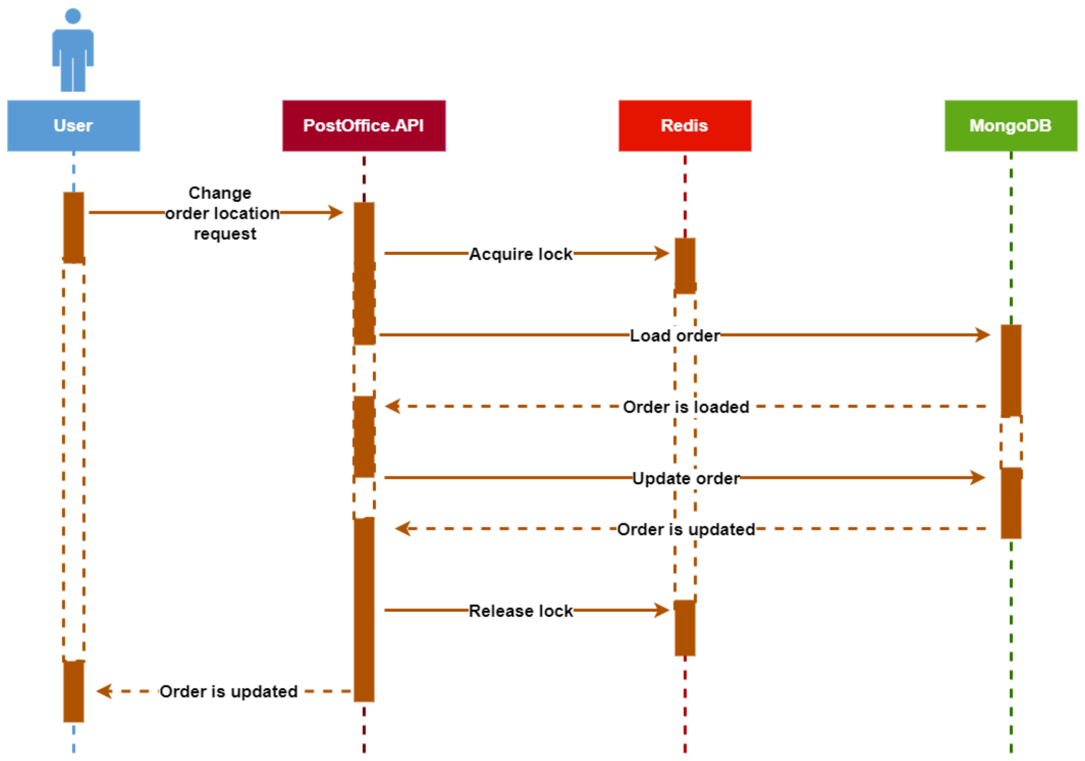
The flow of **creation order** involves the use of integration events since that affects other system such as SmsSender. To maintain data consistency we store integration event before sending them to message queue and update their status depending on where there were successfully published or not. On the other hand it might seem like messages could be lost when SmsSender goes down. But this will not happen. The key point here that instead of pushing messages from queue to service we use pulling model. So service maintain its own loads.



Let's just go to simpler but still interesting approach. In **delete order** request we affect fewer services but the communication between those are asynchronous. We achieved this through use of WebSocket protocol. This way session state is not kept for the entire period but is restored on response.



This use case also get use of asynchronous communication. Another interesting idea it brings is resource blocking. Since order could be updated by different employees and those updates should be independent we use pessimistic lock here. This way the resource is blocked before any interaction with it and released at the end.


User
PostOffice.API
Redis
MongoDB
Change order location request
Acquire lock
Load order
Order is loaded
Update order
Order is updated
Release lock
Order is updated