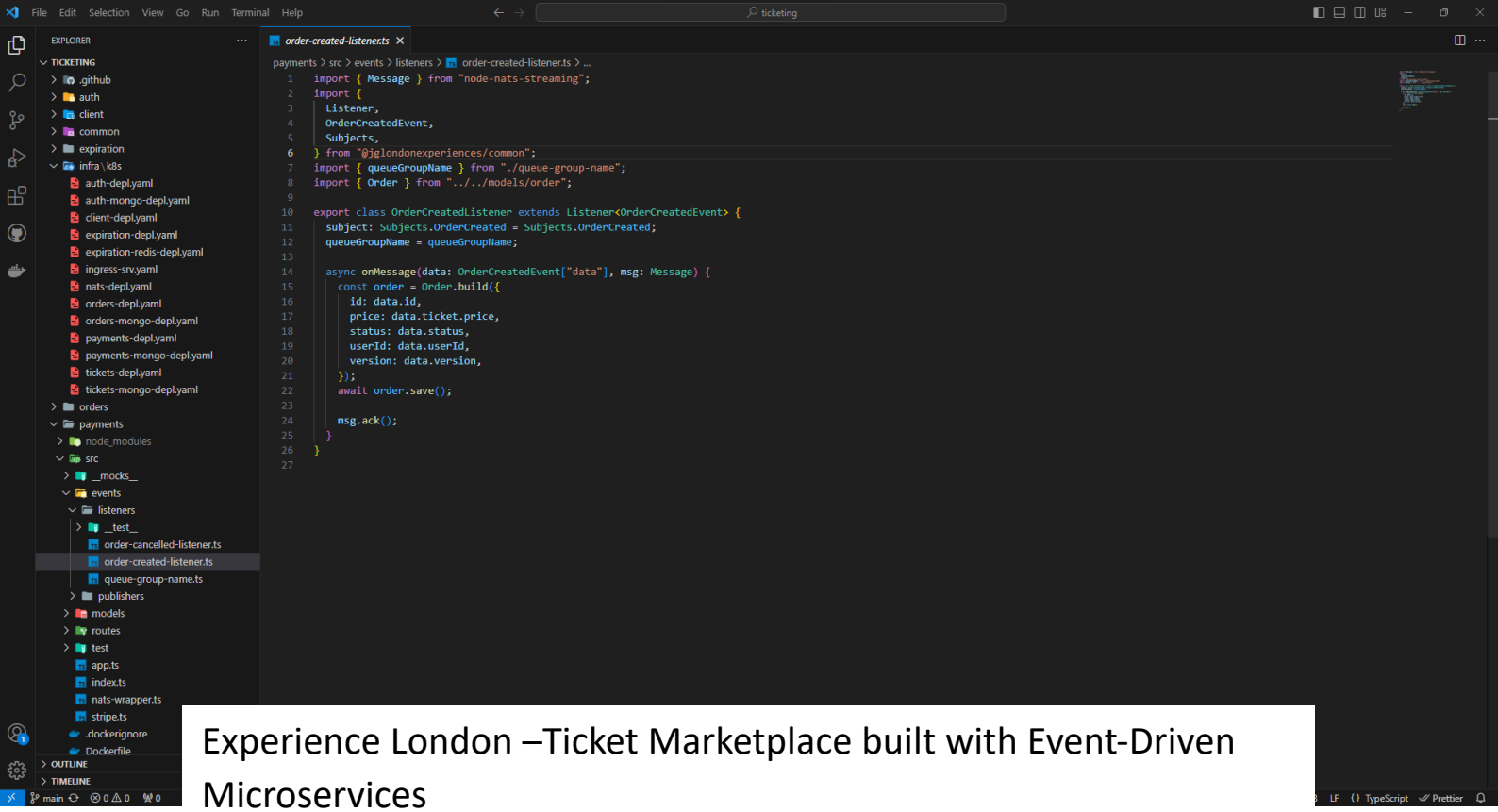

Experience London –Ticket Marketplace built with Event-Driven Microservices

Technical Documentation

This Document outlines the **key challenges** faced when developing this **full stack Event-Driven Microservice** system, along with the information and **solutions** required to solve these problems.

By Jake Gallagher

1. Compartmentalisation of Data
2. Server-side Rendering
3. Concurrency

Compartmentalisation of Data

One of the key issues of any Microservice based architecture is how to split out the different services and what data should they contain, unfortunately this question has no correct answers although it does have many wrong answers. The general consensus between the leading innovators of the area is that each service should encapsulate one of the business capabilities/functions such as authorisation or making payments. This however creates a problem when designing how a service should store and interact with data especially when the data in question is primarily controlled by another service. The best way to solve this problem is with data duplication, storing only what is necessary for the service to carry out its role, this removes the need for any dependencies between services and adheres closely to microservice methodology.

Server-Side Rendering

Utilising Server-Side Rendering from within a Kubernetes cluster poses an interesting networking problem. When navigating or making a request to a relative route the browser will interpret this as a request made to the same domain as it is currently on, for example if we make a request to '/api/users/currentuser' whilst on 'http://ticketing.dev' then the request will be sent to 'http://ticketing.dev/api/users/currentuser' this is not the case when attempting to send a request from within the docker container. When using getServerSideProps the request to the api is not generated by the browser but by the server that is running the NextJS instance, because of this the domain component of the route will be the host:port of the instance, this will result in the full URL looking similar to '127.0.0:80/api/users/currentuser' and coupled with the fact that each docker container is contained within it's own pod/scope/context this URL will result in a ECONNREFUSED error.

To solve this problem, we can use whatever load balancer you have setup with your cluster, this allows you to send only the relative path and you don't have the problem of being required to know what service the API call needs to go to as your load balancer will have been set up with this when creating the service. However, you must remember to setup and forward on any items contained withing the request that will be needed by the receiving service for example cookies.

Concurrency

Concurrency is a big challenge with Event-Driven services, Messages/events can end up being received/processed in the wrong order due to many reasons. The best way to approach this issue is to rely on the service that is responsible for that event data to provide accurate versioning information. This is relatively easy when using MongoDB with Mongoose as it has version numbers built into the model schema.

The process for this is when an event is created, the service will include the version number along with the event details and the listener is able to use this version number to check if it has received all events before the one being processed, if it has then it the event will be handled, and if not then the event will not be acknowledged and will get resent by the event bus, hopefully by this time any earlier events have been handled and the out of order event will now be able to be handled.