Lessons from App #1

The *big challenge* in microservices is data

Different ways to share data between services. We are going to focus on async communication

Async communication focuses on communicating changes using *events* sent to an *event bus*

Async communication encourages each service to be 100% self-sufficient. Relatively easy to handle temporary downtime or new service creation

Docker makes it easier to package up services

Kubernetes is a pain to setup, but makes it really easy to deploy + scale services

Painful Things from App #1

Lots of duplicated code!

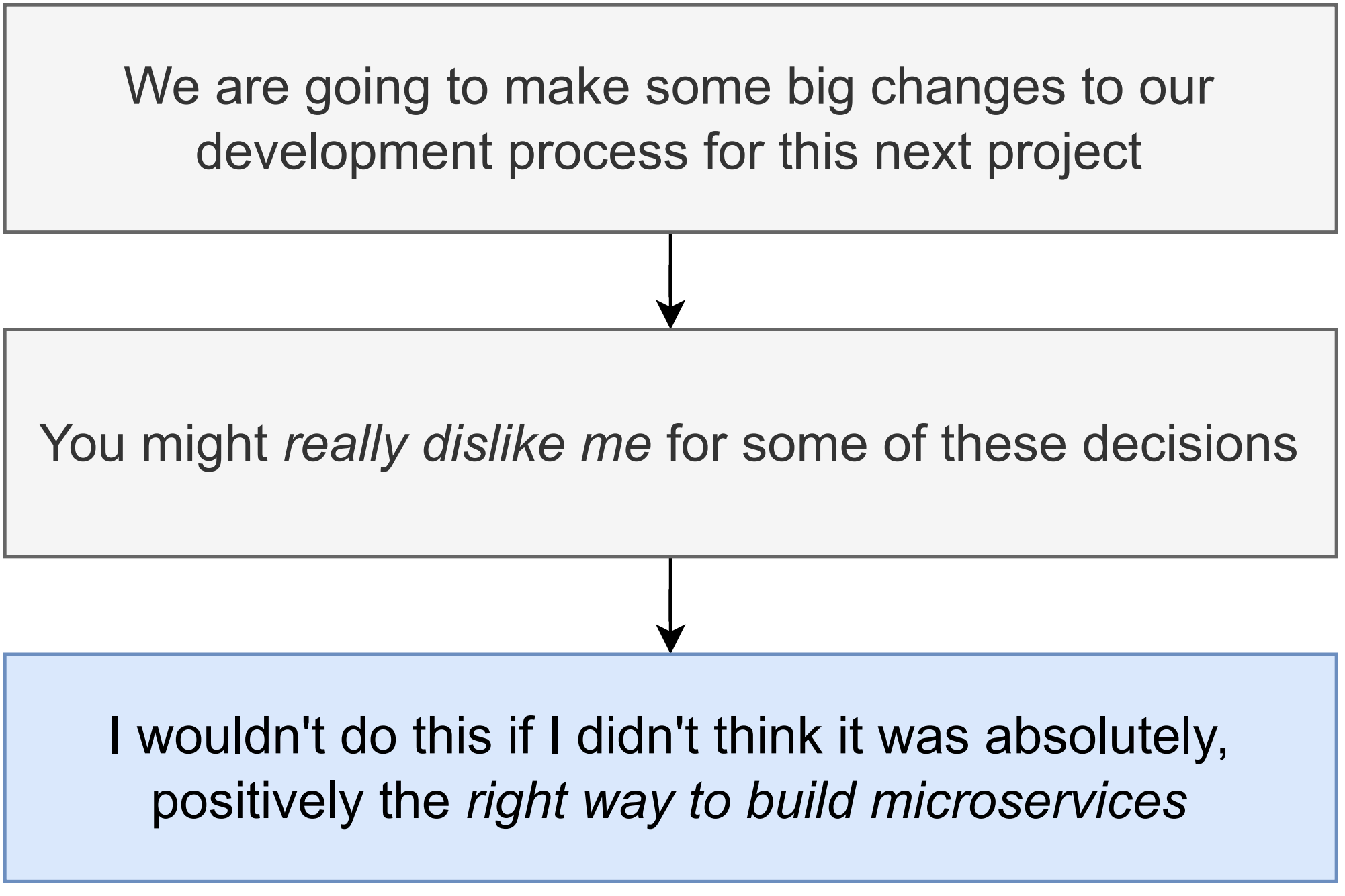
Really hard to picture the flow of events between services

Really hard to remember what properties an event should have

Really hard to test some event flows

My machine is getting laggy running kubernetes and everything else...

What if someone created a comment after editing 5 others after editing a post while balancing on a tight rope....


We are going to make some big changes to our development process for this next project
You might *really dislike me* for some of these decisions
I wouldn't do this if I didn't think it was absolutely, positively the *right way to build microservices*

Painful Things from App #1	Solutions!
Lots of duplicated code!	Build a central library as an NPM module to share code between our different projects
Really hard to picture the flow of events between services	Precisely define all of our events in this shared library.
Really hard to remember what properties an event should have	Write *everything* in Typescript.
Really hard to test some event flows	Write tests for as much as possible/reasonable
My machine is getting laggy running kubernetes and everything else...	Run a k8s cluster in the cloud and develop on it *almost as quickly* as local
What if someone created a comment after editing a post while balancing on a tight rope....	Introduce a lot of code to handle concurrency issues

Ticketing App

Users can list a ticket for an event (concert, sports) for sale

Other users can purchase this ticket

Any user can list tickets for sale and purchase tickets

When a user attempts to purchase a ticket, the ticket is 'locked' for 15 minutes. The user has 15 minutes to enter their payment info.

While locked, no other user can purchase the ticket. After 15 minutes, the ticket should 'unlock'

Ticket prices can be edited if they are not locked

Page 1

https://www.draw.io

GitTix Sign Up Sign In

Tickets For Sale

Basketball Game - $20

Classical Concert - $100

Rock Concert - $100

Football Game - $40

Page 1

https://www.draw.io

GitTix Sign Up Sign In

Sign Up

Email

Password

Submit

Page 1

https://www.draw.io

GitTix Sign Up Sign In

Sign In

Email

Password

Submit

Page 1

https://www.draw.io

GitTix My Orders Sell Tickets Sign Out

Tickets For Sale

Basketball Game - $20

Classical Concert - $100

Rock Concert - $100

Football Game - $40

Page 1

ticketing.dev/tickets/J2342jpkg

GitTix My Orders Sell Tickets Sign Out

Rock Concert

Price - $40
Status - Available

Purchase

Page 1

/orders/alskdjfaksdjfkp

GitTix My Orders Sell Tickets Sign Out

Purchasing Rock Concert

You have 30 seconds left to order

Pay

Page 1

https://www.draw.io

GitTix My Orders Sell Tickets Sign Out

Purchasing R

You have 30 secon

Pay

Credit Card

Expiration

Submit

Page 1

/tickets/new

GitTix My Orders Sell Tickets Sign Out

Create a Ticket

Title

Price

Submit

User

Name	*Type*
email	string
password	string

Ticket

Name	*Type*
title	string
price	number
userId	Ref to User
orderId	Ref to Order

Order

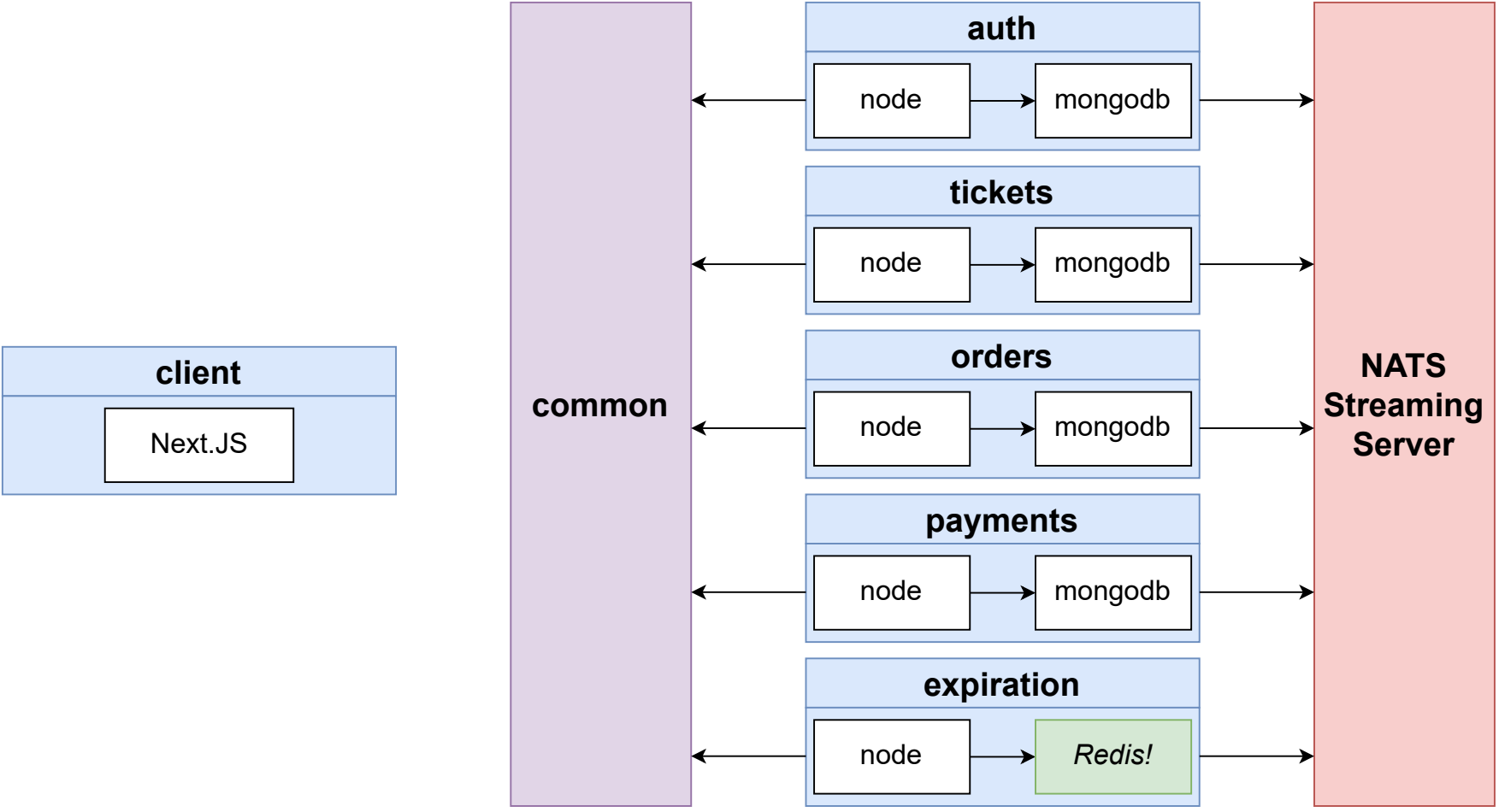
Name	*Type*
userId	Ref to User
status	Created | Cancelled | AwaitingPayment | Completed
ticketId	Ref to Ticket
expiresAt	Date

Charge

Name	*Type*
orderId	Ref to Order
status	Created | Failed | Completed
amount	number
stripeId	string
stripeRefundId	string

Services

auth	Everything related to user signup/signin/signout
tickets	Ticket creation/editing. Knows whether a ticket can be updated
orders	Order creation/editing
expiration	Watches for orders to be created, cancels them after 15 minutes
payments	Handles credit card payments. Cancels orders if payments fails, completes if payment succeeds


client
Next.JS
common
auth
node
mongodb
tickets
node
mongodb
orders
node
mongodb
payments
node
mongodb
expiration
node
Redis!
NATS Streaming Server

We are creating a separate service to manage each *type of resource*

↓

Should we do this for every microservices app?

↓

Probably not? Depends on your use case, number of resources, business logic tied to each resource, etc

↓

Perhaps 'feature-based' design would be better

Events

UserCreated

UserUpdated

OrderCreated

OrderCancelled

OrderExpired

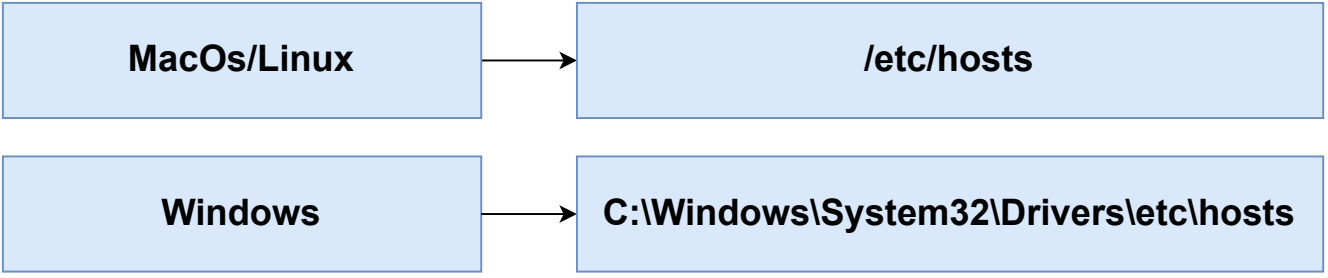
TicketCreated

TicketUpdated

ChargeCreated

auth			
Route	Method	Body	Purpose
/api/users/signup	POST	{ email: string, password: string }	Sign up for an account
/api/users/signin	POST	{ email: string, password: string }	Sign in to an existing account
/api/users/signout	POST	{}	Sign out
/api/users/currentuser	GET	-	Return info about the user

kubernetes.github.io/ingress-nginx


MacOs/Linux
/etc/hosts
Windows
C:\Windows\System32\Drivers\etc\hosts

Unskippable HTTPS warning in Chrome?

thisisunsafe