DevOpsCon

by devmio

Workshop: Unlocking Scalable and Real-Time Data Access for Developers

Introductions

- Tom Hacohen
  svix

- Jerry Yang
  zoom

Agenda

- 900 - 945 - Introduction to EDA
- 945 - 1030 - Coding Block 1 - Traditional API Foundation

- 1030 - 1100 - Break

- 1100 - 1130 - API vs EDA
- 1130 - 1230 - Coding Block 2 - Transitioning to Webhooks

- 1230 - 130 - Lunch

- 130 - 200 - Reliability, Security and Observability
- 200 - 300 - Coding Block 3 - Enhancing your Service

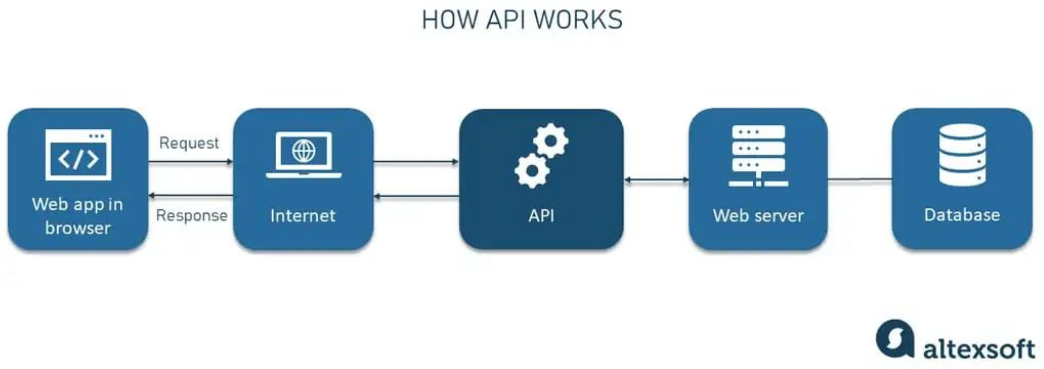
- 300 - 330 - Break

- 330 - 400 - Remaining Challenges and Best Practices
- 400 - 430 - Closing Statements/Questions

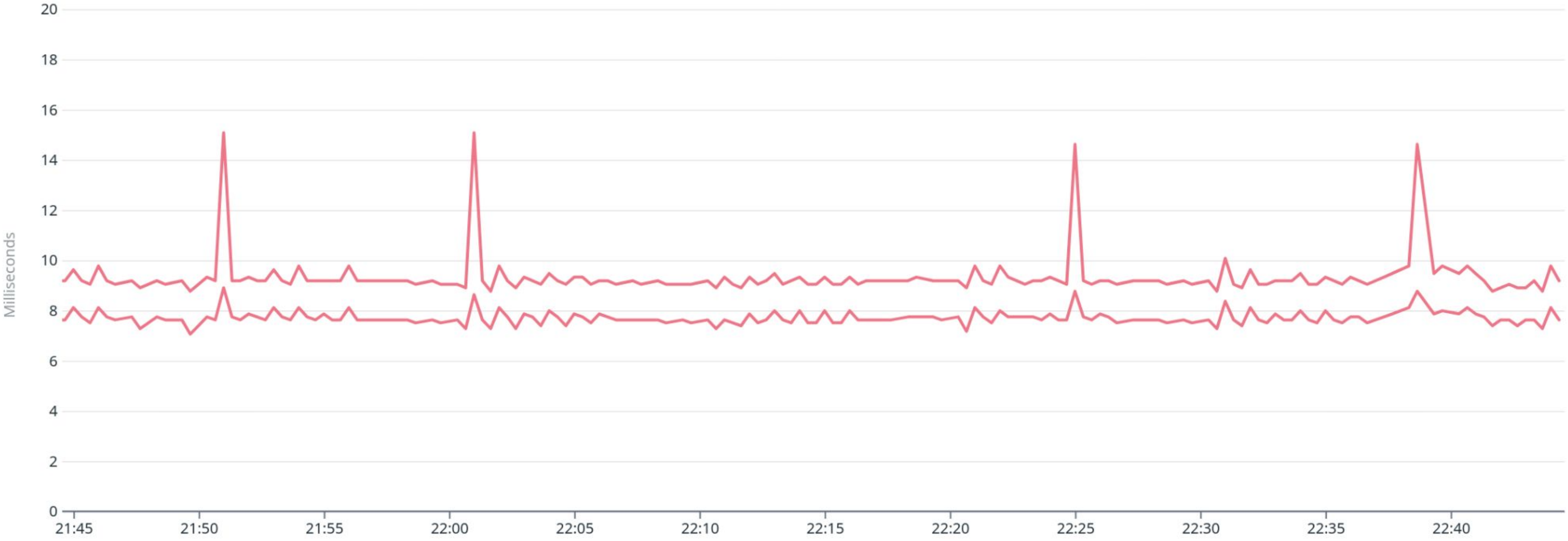
First, some background…

HTTP APIs and Synchronous Communication

- API is the foundation of the internet, REST is the most common pattern
- Request response system
  - Make a request -> do something -> return a response.
- Client to server and server to server.



Oftentimes API calls are fast.



But the desired operations are slow…

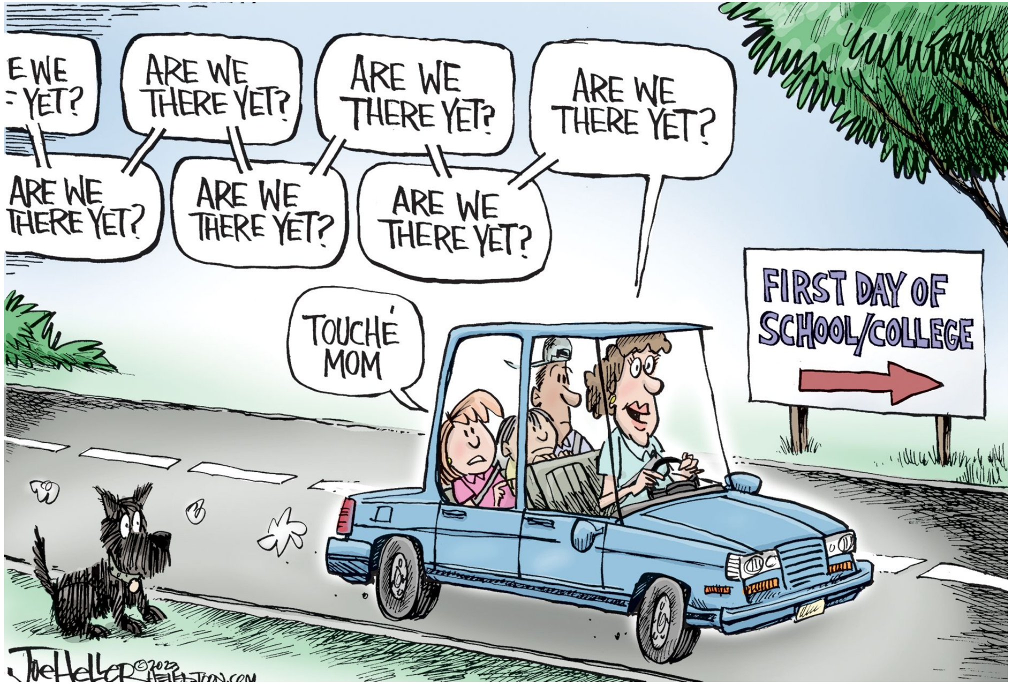
- Training an AI
- Processing a video
- Finding an Uber driver

Usually APIs follow a request-response

But sometimes they rely on external events

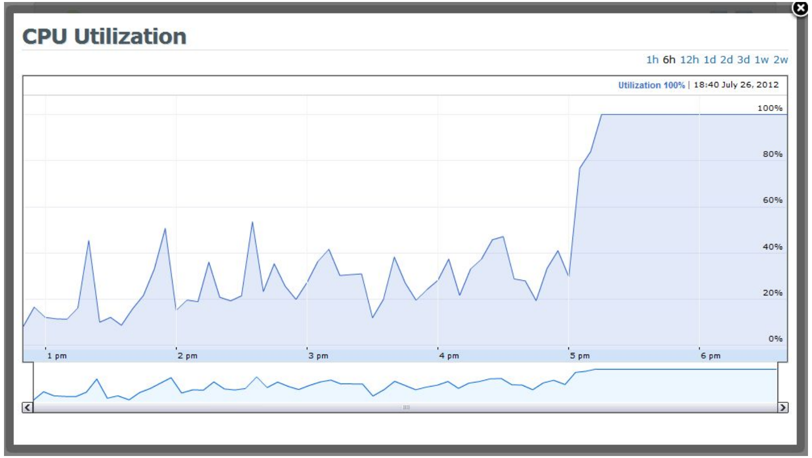
- Email received
- Package delivered
- Fraud detected

Solution: polling!



But polling is full of problems

- Data is only as real-time as the polling interval
- Creates load on the server
- Requires long-running tasks that are durable



Better Solution: Event Driven APIs

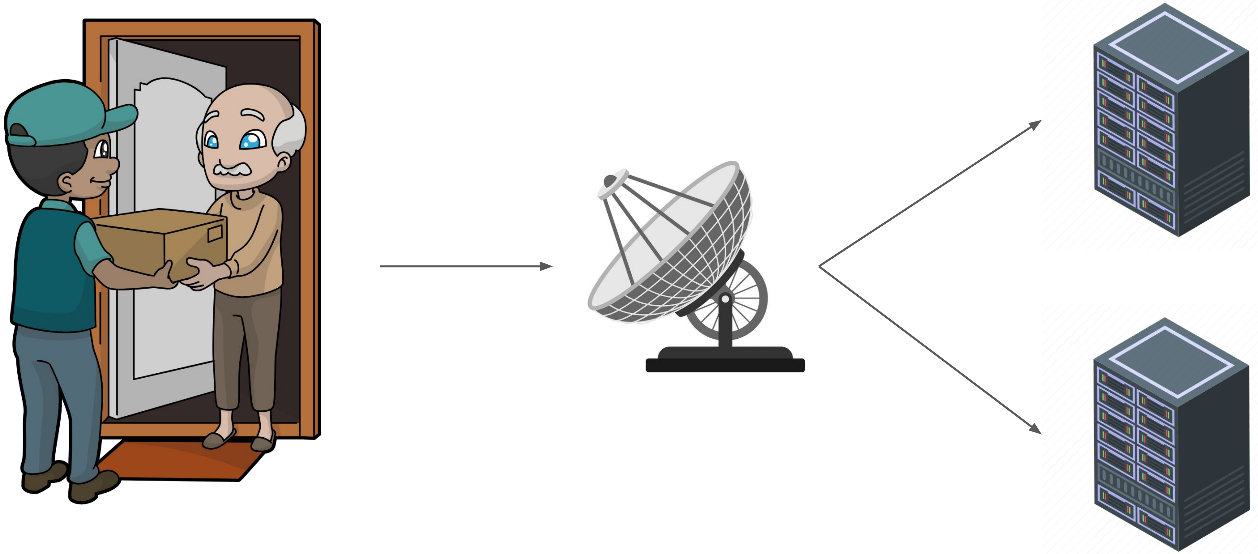
- Get notified when events happen. No need to “ask”.
    - Got an email? Get notified.
    - Package delivered? Get notified.
    - Fraud detected? Get notified.
    - AI training finished? Get notified.

Or more broadly: Event Driven Architecture (EDA)

- Real time data processing
- Natural fit for asynchronous events and workflows
- Pub/Sub relationships - can be one-to-one/one-to-many/many-to-many
- Loose coupling
    - Easy to scale producers
    - Easy to scale consumers

EDA in Practice

- Generate events at the source (producers)
- Manage the flow of events (brokers)
- Deliver events to the targets (consumers)



Benefits of Adopting EDA

↓

Ecosystem Enablement

The event-driven model helps simplify the integration of third-party services and extensions, enabling platforms to easily expand their ecosystem.

Efficient Resource Allocation

Enhances resource utilization by dynamically allocating based on event-driven demand.

Integrity

Maintains system integrity and accuracy, even as it scales, through coordinated, event-based reactions.

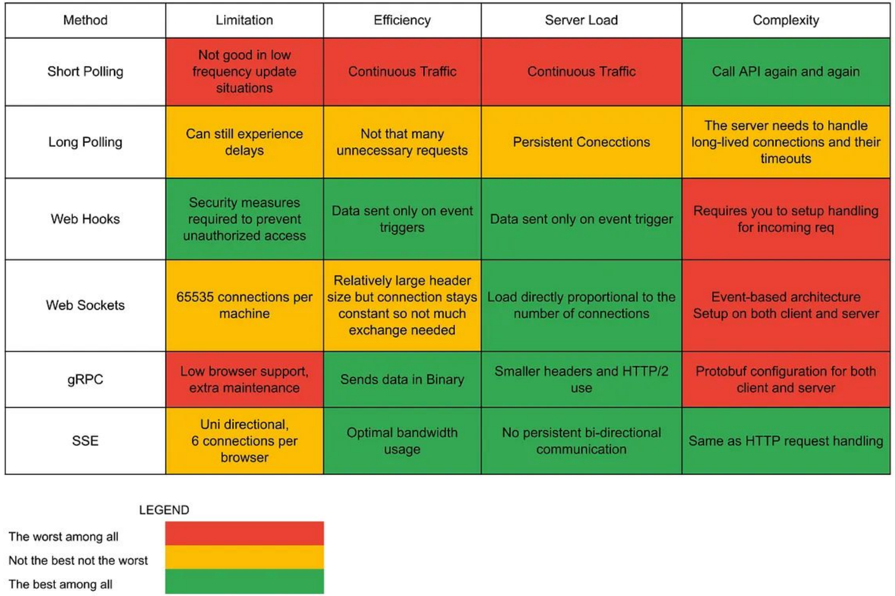
Enhanced Data Consistency

Supports data consistency across distributed systems via event sourcing and immutable event logs.

Cost Savings

Helps reduce operational expenses through enhanced resource use and streamlined development processes.

EDA in Web Technologies

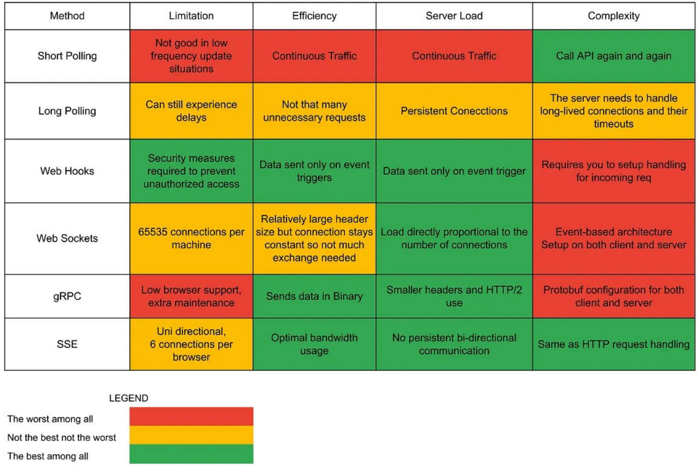


Method	Limitation	Efficiency	Server Load	Complexity
Short Polling	Not good in low frequency update situations	Continuous Traffic	Continuous Traffic	Call API again and again
Long Polling	Can still experience delays	Not that many unnecessary requests	Persistent Conecctions	The server needs to handle long-lived connections and their timeouts
Web Hooks	Security measures required to prevent unauthorized access	Data sent only on event triggers	Data sent only on event trigger	Requires you to setup handling for incoming req
Web Sockets	65535 connections per machine	Relatively large header size but connection stays constant so not much exchange needed	Load directly proportional to the number of connections	Event-based architecture Setup on both client and server
gRPC	Low browser support, extra maintenance	Sends data in Binary	Smaller headers and HTTP/2 use	Protobuf configuration for both client and server
SSE	Uni directional, 6 connections per browser	Optimal bandwidth usage	No persistent bi-directional communication	Same as HTTP request handling

LEGEND	
The worst among all	🟥
Not the best not the worst	🟨
The best among all	🟩

Short polling

- Query for status updates, return immediately.



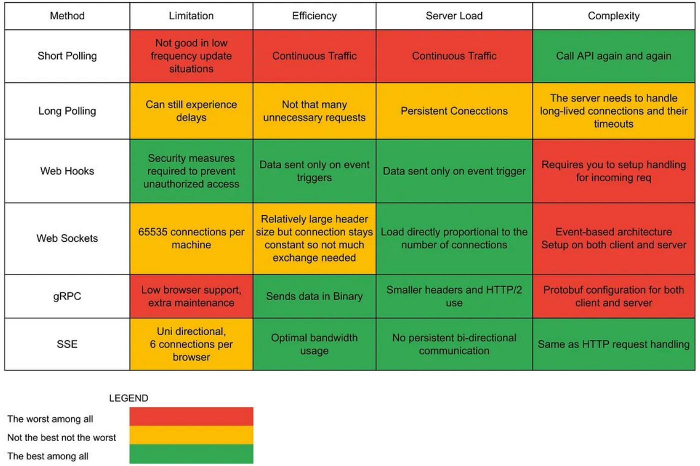
Method	Limitation	Efficiency	Server Load	Complexity
Short Polling	Not good in low frequency update situations	Continuous Traffic	Continuous Traffic	Call API again and again
Long Polling	Can still experience delays	Not that many unnecessary requests	Persistent Conecctions	The server needs to handle long-lived connections and their timeouts
Web Hooks	Security measures required to prevent unauthorized access	Data sent only on event triggers	Data sent only on event trigger	Requires you to setup handling for incoming req
Web Sockets	65535 connections per machine	Relatively large header size but connection stays constant so not much exchange needed	Load directly proportional to the number of connections	Event-based architecture Setup on both client and server
gRPC	Low browser support, extra maintenance	Sends data in Binary	Smaller headers and HTTP/2 use	Protobuf configuration for both client and server
SSE	Uni directional, 6 connections per browser	Optimal bandwidth usage	No persistent bi-directional communication	Same as HTTP request handling

LEGEND

The worst among all	🟥
Not the best not the worst	🟨
The best among all	🟩

Long polling

- Query for status updates, if there’s nothing, wait until there is (up to a timeout).



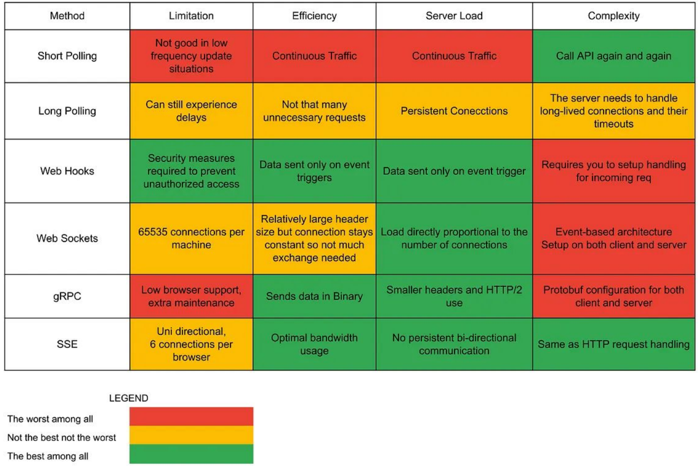
Method	Limitation	Efficiency	Server Load	Complexity
Short Polling	Not good in low frequency update situations	Continuous Traffic	Continuous Traffic	Call API again and again
Long Polling	Can still experience delays	Not that many unnecessary requests	Persistent Conecctions	The server needs to handle long-lived connections and their timeouts
Web Hooks	Security measures required to prevent unauthorized access	Data sent only on event triggers	Data sent only on event trigger	Requires you to setup handling for incoming req
Web Sockets	65535 connections per machine	Relatively large header size but connection stays constant so not much exchange needed	Load directly proportional to the number of connections	Event-based architecture Setup on both client and server
gRPC	Low browser support, extra maintenance	Sends data in Binary	Smaller headers and HTTP/2 use	Protobuf configuration for both client and server
SSE	Uni directional, 6 connections per browser	Optimal bandwidth usage	No persistent bi-directional communication	Same as HTTP request handling

LEGEND

The worst among all	🟥
Not the best not the worst	🟨
The best among all	🟩

Webhooks

- Get an HTTP callback call on updates.



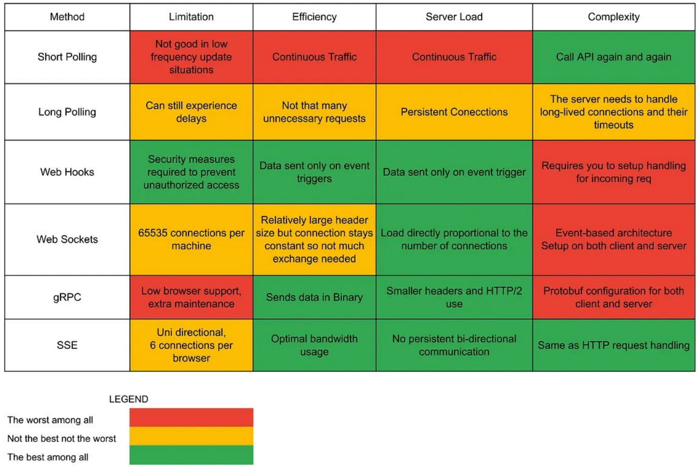
Method	Limitation	Efficiency	Server Load	Complexity
Short Polling	🟥 Not good in low frequency update situations	🟥 Continuous Traffic	🟥 Continuous Traffic	🟩 Call API again and again
Long Polling	🟨 Can still experience delays	🟨 Not that many unnecessary requests	🟨 Persistent Conecctions	🟨 The server needs to handle long-lived connections and their timeouts
Web Hooks	🟩 Security measures required to prevent unauthorized access	🟩 Data sent only on event triggers	🟩 Data sent only on event trigger	🟥 Requires you to setup handling for incoming req
Web Sockets	🟨 65535 connections per machine	🟨 Relatively large header size but connection stays constant so not much exchange needed	🟩 Load directly proportional to the number of connections	🟥 Event-based architecture Setup on both client and server
gRPC	🟥 Low browser support, extra maintenance	🟩 Sends data in Binary	🟩 Smaller headers and HTTP/2 use	🟥 Protobuf configuration for both client and server
SSE	🟨 Uni directional, 6 connections per browser	🟩 Optimal bandwidth usage	🟩 No persistent bi-directional communication	🟩 Same as HTTP request handling

LEGEND

The worst among all	🟥
Not the best not the worst	🟨
The best among all	🟩

WebSockets

- Keep an active connection and get a message when there are changes.



Method	Limitation	Efficiency	Server Load	Complexity
Short Polling	Not good in low frequency update situations	Continuous Traffic	Continuous Traffic	Call API again and again
Long Polling	Can still experience delays	Not that many unnecessary requests	Persistent Conecctions	The server needs to handle long-lived connections and their timeouts
Web Hooks	Security measures required to prevent unauthorized access	Data sent only on event triggers	Data sent only on event trigger	Requires you to setup handling for incoming req
Web Sockets	65535 connections per machine	Relatively large header size but connection stays constant so not much exchange needed	Load directly proportional to the number of connections	Event-based architecture Setup on both client and server
gRPC	Low browser support, extra maintenance	Sends data in Binary	Smaller headers and HTTP/2 use	Protobuf configuration for both client and server
SSE	Uni directional, 6 connections per browser	Optimal bandwidth usage	No persistent bi-directional communication	Same as HTTP request handling

LEGEND

The worst among all	🟥
Not the best not the worst	🟨
The best among all	🟩

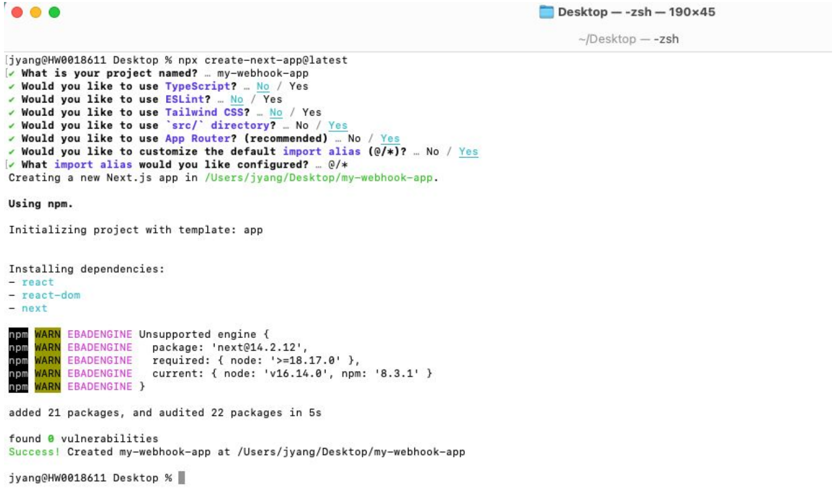
It’s time to build!

Let’s start with building a simple API

Code 1.1

- **Main goal:** Set up a working API backend service service that will be our mock upstream event producer. For this section we will be building:
    - Working API endpoints
    - A basic data storage

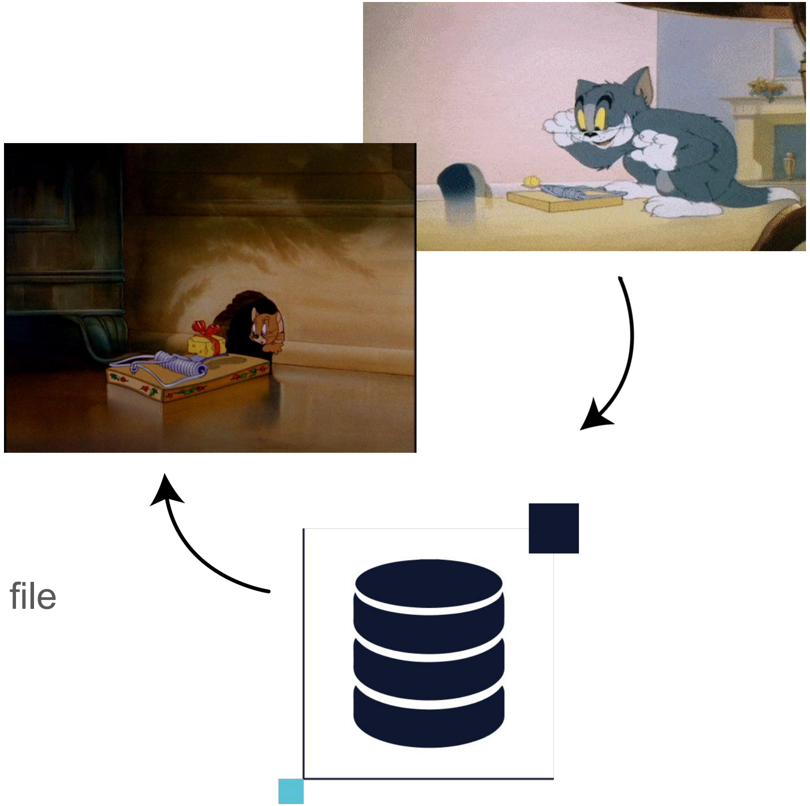
- We will use next.js as a javascript backend framework
- next.js startup guide
    - `npx create-next-app@latest`
    - Requires node.js version 18.17.0+
    - "starter" code



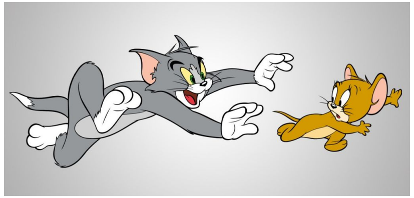
```
jyang@HW0018611 Desktop % npx create-next-app@latest
✔ What is your project named? … my-webhook-app
✔ Would you like to use TypeScript? … No / Yes
✔ Would you like to use ESLint? … No / Yes
✔ Would you like to use Tailwind CSS? … No / Yes
✔ Would you like to use `src/` directory? … No / Yes
✔ Would you like to use App Router? (recommended) … No / Yes
✔ Would you like to customize the default import alias (@/*)? … No / Yes
✔ What import alias would you like configured? … @/*
Creating a new Next.js app in /Users/jyang/Desktop/my-webhook-app.

Using npm.

Initializing project with template: app


Installing dependencies:
- react
- react-dom
- next

npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'next@14.2.12',
npm WARN EBADENGINE   required: { node: '>=18.17.0' },
npm WARN EBADENGINE   current: { node: 'v16.14.0', npm: '8.3.1' }
npm WARN EBADENGINE }

added 21 packages, and audited 22 packages in 5s

found 0 vulnerabilities
Success! Created my-webhook-app at /Users/jyang/Desktop/my-webhook-app

jyang@HW0018611 Desktop % 
```

Code 1.2

- API :
    - GET - check to see if there are any traps
    - POST - add a ‘trap’ to your database
    - POST - diffuse a ‘trap’ (remove)
    - How to implement API in NEXT
- Data Storage/database :
    - SQLite db
    - local data storage in the form of a .json object file
    - in-memory storage array that runs in project



Code 1.3



- if you didn't, go back and set up a SQLite DB
  - We will be creating a new set of REST API and a table to store event subscriptions in the next section - consider what parameters will be needed
- Check components we'll be using soon:
  - Docker desktop (and docker-compose.yml)
  - Another Redis Desktop Manager
  - node dependencies - package.json
- Begin the transition to an EDA service
  - we will be building a webhook in this workshop: the standardwebhooks docs are a great place to start!

DevOpsCon
by devmio

SECTION 2: Advantages of EDA

APIs let you build scripts,
EDA lets you build integrations…

APIs are inherently one-sided

APIs are inherently one-sided

- You can create scripts to make modifications:
  - List / Delete / Send / Pay
- But you can’t react to events:
  - Changed / Received / Failed

APIs are sync EDA is async

- Sync is easier to reason about.
- Async has different failure modes.
- Order of events can get confusing.

API vs. EDA

- main comparisons
    - Communication Produce/consume vs. Request Response
        - Async vs sync
        - Higher possible delay/latency vs. immediate connection
    - Loose coupling vs specific response structures
    - Use Cases - modular/scalable vs. more structured/specific/straightforward

Sync APIs vs. Async Events

- It’s not a this vs. that, it’s a this + that
- Sync APIs are simpler to reason about but don’t work for async
- Async offers decoupling and flexibility:
    - Different systems can respond and scale differently to events
- Async increases latency but is also more real-time
- Polling is operationally expensive for both consumer and send

Webhooks as an example

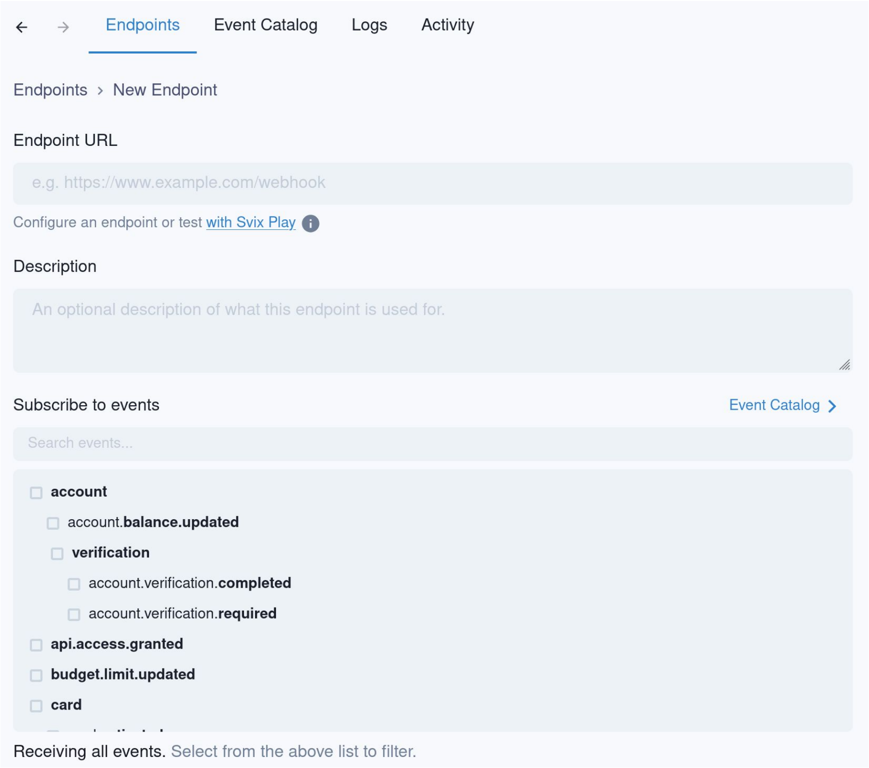
What are webhooks?

Webhooks are a common name for HTTP callbacks, and are how services notify each other of events.

Most common service-to-service EDA

- Webhooks are the most common example of EDA between services.
- Utilized by Stripe, Github, Zoom, Svix, and many others.
- You’ve probably used them yourself.

How do they usually work?



Endpoints › New Endpoint

Endpoint URL

e.g. https://www.example.com/webhook

Configure an endpoint or test with Svix Play

Description

An optional description of what this endpoint is used for.

Subscribe to events

- ☐ **account**
  - ☐ account.**balance.updated**
  - ☐ **verification**
    - ☐ account.verification.**completed**
    - ☐ account.verification.**required**
- ☐ **api.access.granted**
- ☐ **budget.limit.updated**
- ☐ **card**

Receiving all events. Select from the above list to filter.

Live demo of Webhooks - ZOOM

- Zoom App Marketplace: https://devmp.zoomdev.us/

Live demo of a Webhook Portal

Here is an example of webhook management portal.

- Using the pre-built portal included with svix
- https://example.svix.com

In this workshop we’ll use webhooks!

Why webhooks?

- Easy
- Ubiquitous
- Powerful
- Most common
- Great for server to server - most common place.

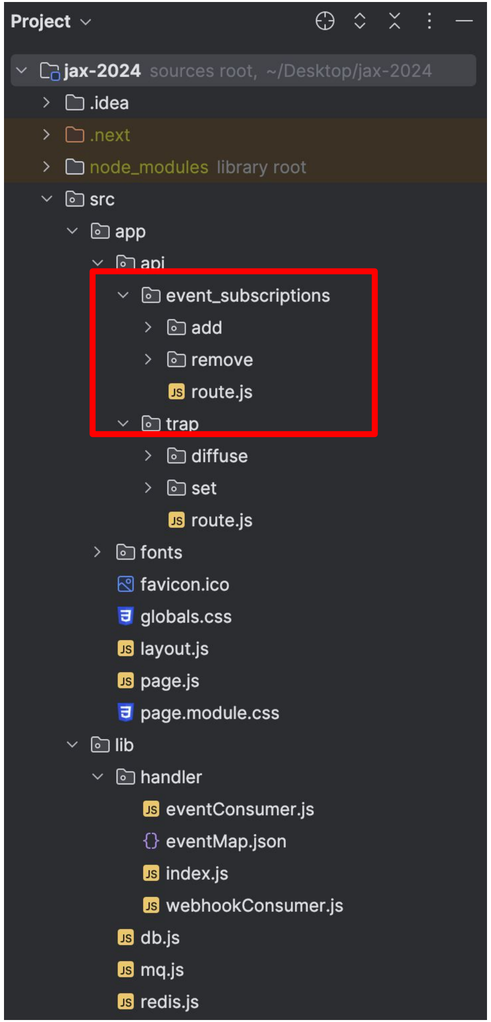
Code Recap!

https://github.com/jerryang1023/jax-webhook-2024/tree/workshop-part-1

Code 2.1

- **Main goal:** Transition our backend service from ‘just’ an API endpoint to an Event Consumer + Webhook Producer
  - Add a second set of API - this time to add/remove/fetch event subscription data, as well as a table in your database to store this information
  - Turn your original API into an **event trigger** - we will produce a webhook when Set/Diffuse is called
  - **Automatically** send webhooks when events occur

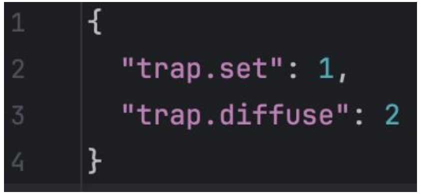
- There are 2 options to use for your webhook test endpoint:
  - [Svix Playground]()
  - [webhook.site]()



```
jax-2024  sources root, ~/Desktop/jax-2024
  .idea
  .next
  node_modules  library root
  src
    app
      api
        event_subscriptions
          add
          remove
          route.js
        trap
          diffuse
          set
          route.js
      fonts
      favicon.ico
      globals.css
      layout.js
      page.js
      page.module.css
    lib
      handler
        eventConsumer.js
        eventMap.json
        index.js
        webhookConsumer.js
      db.js
      mq.js
      redis.js
```

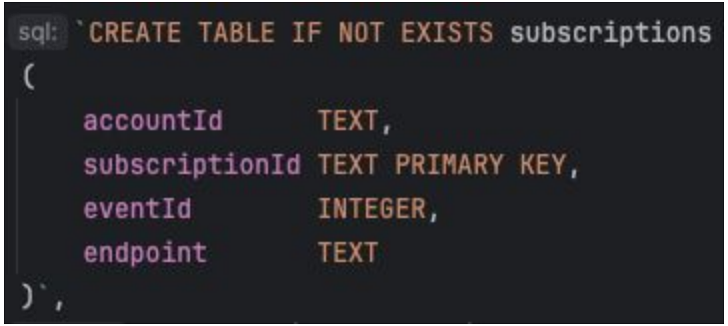
Code 2.2

- Define your events + event subscriptions!



```json
{
  "trap.set": 1,
  "trap.diffuse": 2
}
```

- Create subscription API/database :
  - GET - check existing event subscriptions
  - POST - add a new event subscription
  - POST - remove an existing subscription by subscriptionID



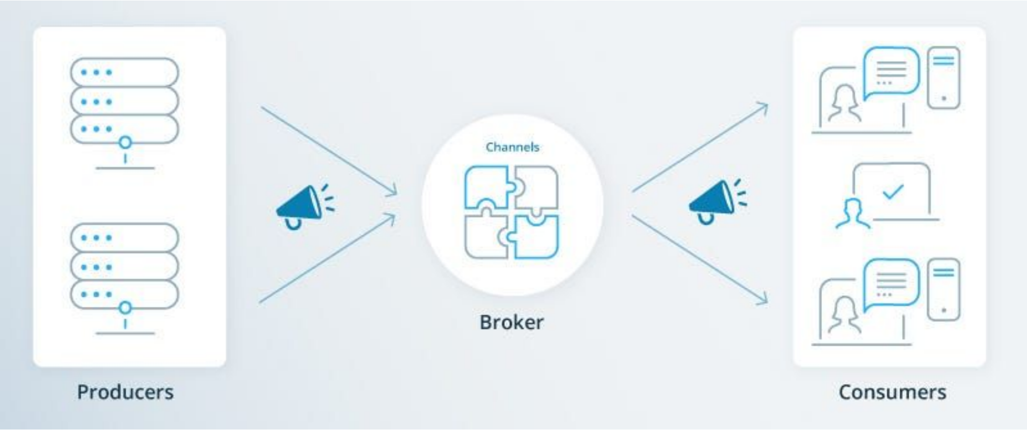
```
sql: `CREATE TABLE IF NOT EXISTS subscriptions
(
    accountId      TEXT,
    subscriptionId TEXT PRIMARY KEY,
    eventId        INTEGER,
    endpoint       TEXT
)`,
```

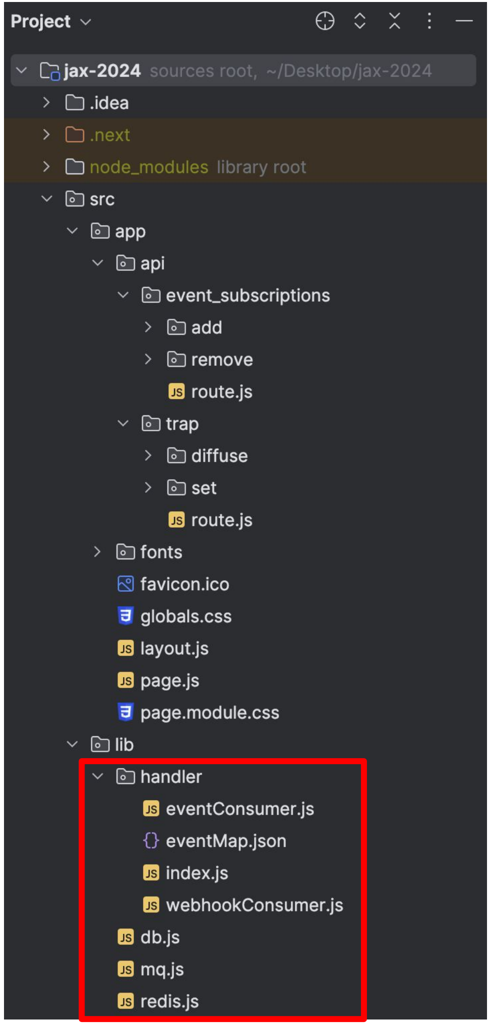
- Modify your old API to produce webhooks to event subscription endpoints
  - … but how will we do this? Inline calls?

We need a message broker!

Code 2.3

- **Main goal #2:** To have a proper Event Consumer/Webhook service, producing Webhooks inline is insufficient. We need to implement a **message broker**
  - Message brokers are a core component of EDA
  - We need a component we and **asynchronously produce** events too and **consume** events from



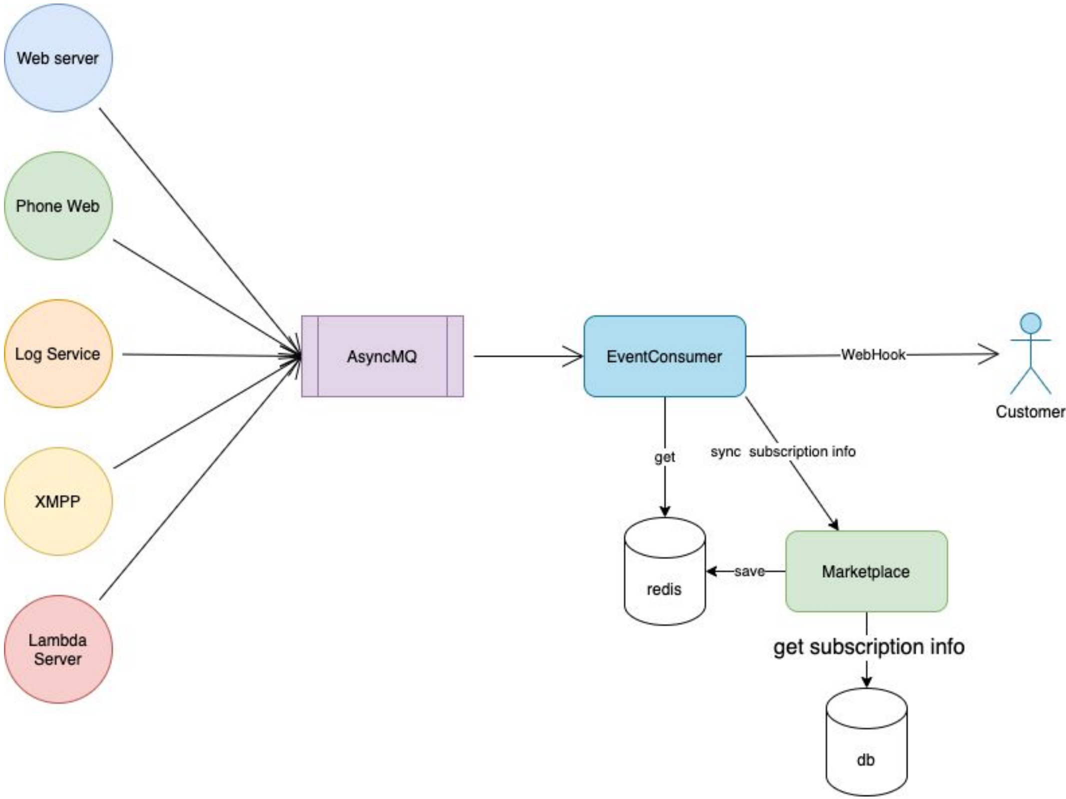


```
jax-2024 sources root, ~/Desktop/jax-2024
  .idea
  .next
  node_modules library root
  src
    app
      api
        event_subscriptions
          add
          remove
          route.js
        trap
          diffuse
          set
          route.js
      fonts
      favicon.ico
      globals.css
      layout.js
      page.js
      page.module.css
    lib
      handler
        eventConsumer.js
        eventMap.json
        index.js
        webhookConsumer.js
      db.js
      mq.js
      redis.js
```

Code 2.4

- In a more sophisticated architecture the producer, broker, and consumer would likely all be their own separate services

- Ex. @ Zoom

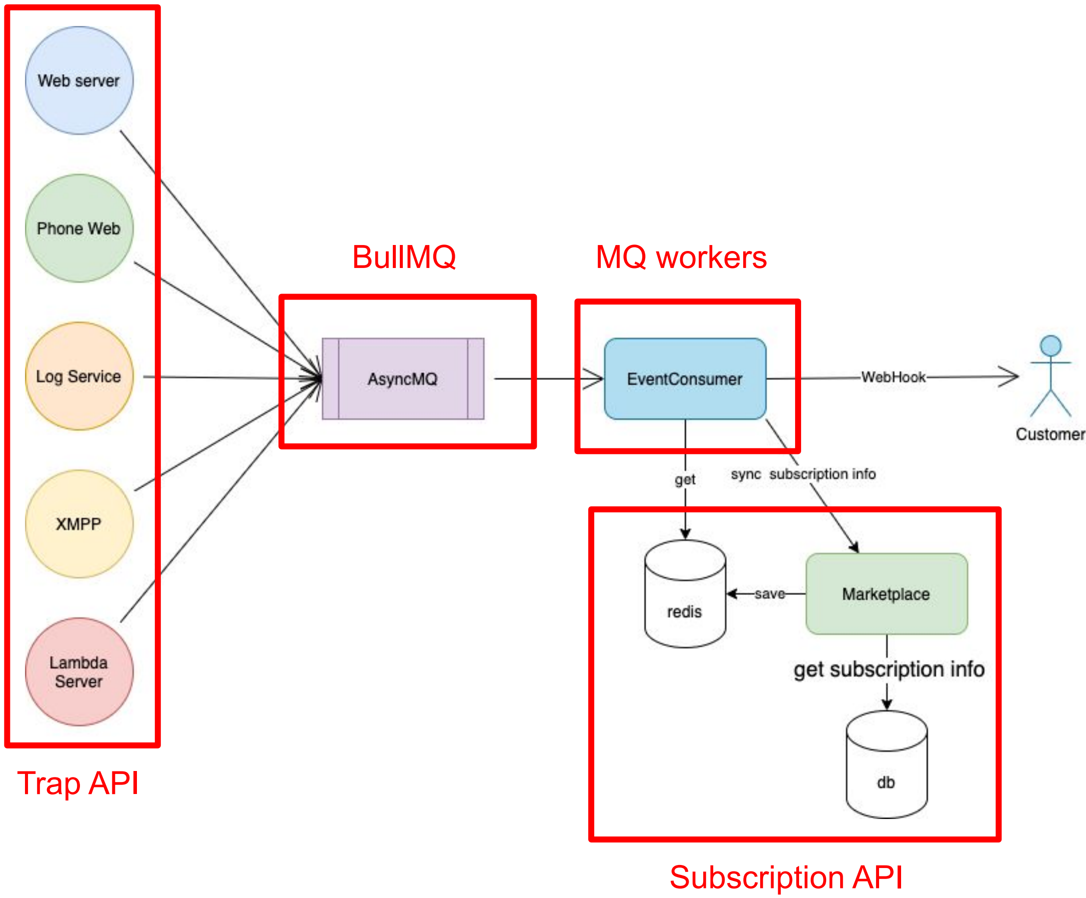


Code 2.4

- In a more sophisticated architecture the producer, broker, and consumer would likely all be their own separate services

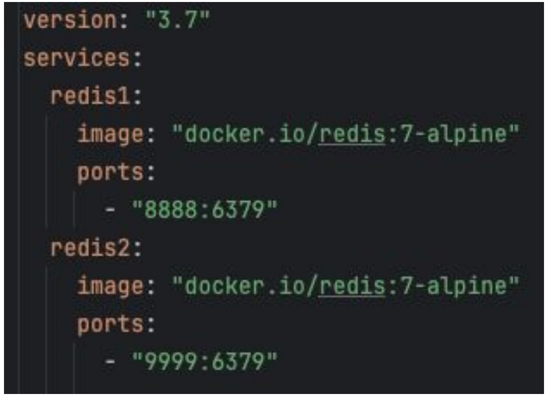
- Ex. @ Zoom

- What we are building is going to be simpler…



Code 2.5

- Start Redis through Docker. We will have 2 different instances for this project:
  - “8888:6379” - this queue will store events produced from our upstream service
  - “9999:6379” - this queue will store webhook events to be consumed by our webhook handlers



```
version: "3.7"
services:
  redis1:
    image: "docker.io/redis:7-alpine"
    ports:
      - "8888:6379"
  redis2:
    image: "docker.io/redis:7-alpine"
    ports:
      - "9999:6379"
```

- We will implement our message broker through [BullMQ]
  - BullMQ is a queue system built on top of Redis
    - Relatively easy to implement!
  - Asynchronous read-out from the queue using Worker threads
    - Threads are easy to scale!

DevOpsCon
by devmio

SECTION 3: Design Best Practices

Downsides of Webhooks

- Unnecessary complexity for simple use-cases (e.g. point-to-point integration)
- Increased latency (but more real-time)
- Common to work with stale data (eventual consistency)
- Ordering of events (can’t be guaranteed)
- Debugging and monitoring (always a pain)
- Learning curve (that’s why we are giving this talk!)

Limitations of EDA vs API

Immediate Data Access

EDA may lag in scenarios requiring instant, synchronous access, where APIs support direct and swift data retrieval.

Point to Point Integrations

For basic client-server exchanges, EDA's overhead may not justify its use over simpler API calls.

Stateless Operations

In applications where each transaction is isolated, EDA's complexity offers little advantage over straightforward API requests.

Monolithic Applications

Applications that are not justified due to scale or complexity, API-based architectures can be easier to implement and manage, offering a more traditional approach to application design.

Simple Interactions

When solutions demand uncomplicated, direct connections, the simplicity of APIs can often outweigh EDA's benefits.

Rapid Prototyping

The agility of APIs in rapid development settings can be more conducive to prototyping than setting up an EDA framework.

Best Practices in Designing EDA

Event Design and Domain Alignment

- Clear event specification
- Domain-driven design

Idempotency and Order Management

- Avoid duplicating
- Helps to ensure event ordering e.g: timestamps

Scalability

- Asynchronous communication
- Seamlessly handle growing load

Robust Error Handling and Security

- Retries and dead letters
- Encryption and access control

Event Sourcing and System Observability

- Logging
- Monitoring

Operational Efficiency

- Dynamic resource allocatio
- Maintenance efficiency

Monitoring is extremely important

- API call: you know if fails.
- Webhook: you don't know if you were supposed to get one.

APRIL 27, 2018

Hello, I was wondering when my order will be shipped? 1:30 PM

Your order has been processed and we are preparing your item for shipment shortly. We will send you a notification on WhatsApp once it has been shipped. 1:32 PM

Thank you! 1:32 PM

APRIL 30, 2018

Your order has shipped and is expected to be delivered on May 5, 2018! 12:00 PM

Reliability of delivery

- Webhooks are often critical ➡ need to succeed in real-time.
    - Not always possible…
    - Server is down? Networking issue? Bug?
- Now is best, “as soon as possible” is second-best.
- Solution:
    - Retry with an exponential backoff
    - Notify on failures
    - Allow for manual redrives

Webhook security (why)

- Webhooks is just an unauthenticated HTTP POST.
- Can come from anyone.
- URL is modifiable by users (so can be sent anywhere).

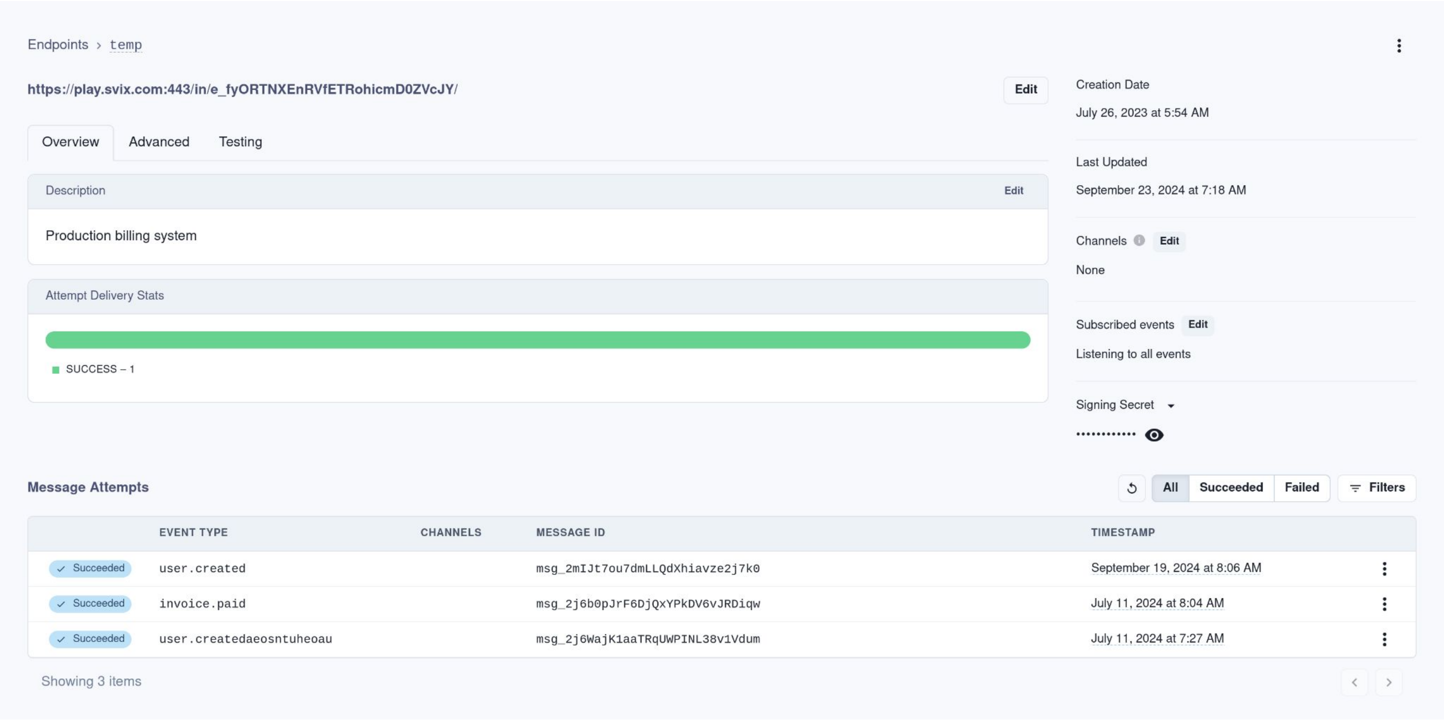
Webhook security (how)

- Sign payloads and timestamps (e.g. using StandardWebhooks)
- Sometimes also verify receiver (e.g using CRC check response)

Additional (optional) mechanisms

- Usually used for compliance reasons.
- Authorization header
- Mutual TLS
- Static source IPs

Observability for webhooks



https://play.svix.com:443/in/e_fyORTNXEnRVfETRohicmD0ZVcJY/

Description

Production billing system

Attempt Delivery Stats

🟩 SUCCESS – 1

Creation Date: July 26, 2023 at 5:54 AM

Last Updated: September 23, 2024 at 7:18 AM

Channels: None

Subscribed events: Listening to all events

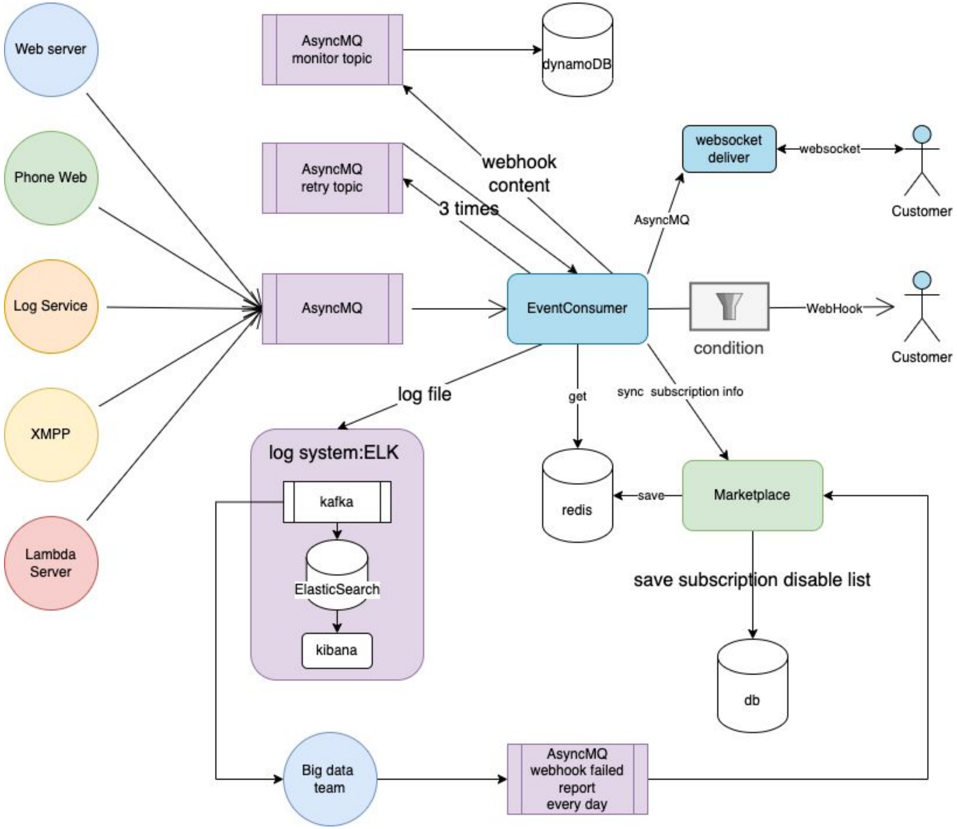
Signing Secret: ••••••••••••

Message Attempts

	EVENT TYPE	CHANNELS	MESSAGE ID	TIMESTAMP
✓ Succeeded	`user.created`		`msg_2mIJt7ou7dmLLQdXhiavze2j7k0`	September 19, 2024 at 8:06 AM
✓ Succeeded	`invoice.paid`		`msg_2j6b0pJrF6DjQxYPkDV6vJRDiqw`	July 11, 2024 at 8:04 AM
✓ Succeeded	`user.createdaeosntuheoau`		`msg_2j6WajK1aaTRqUWPINL38v1Vdum`	July 11, 2024 at 7:27 AM

Showing 3 items

Let’s take another look at Zoom’s architecture…



Code Recap!

https://github.com/jerryang1023/jax-webhook-2024/tree/workshop-part-2

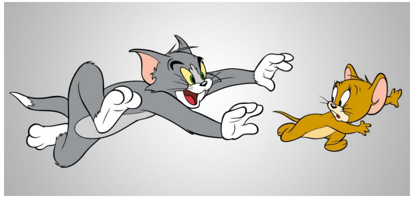
Code 3.1

- **Main goal #1:** Implement reliability in the form of a **staggered retry queue**.
  - Decide on a retry delay interval
  - Add a new retry queue that processes these events!
  - BullMQ has a delay function built in

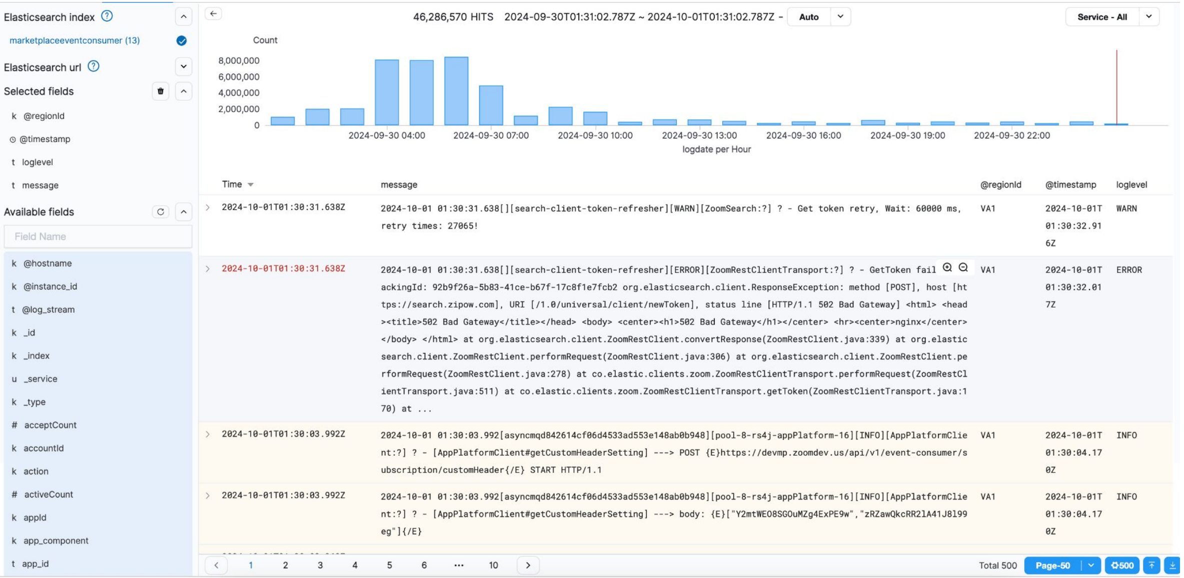
- **Main goal #2:** Implement security in the form of a symmetric signature
  - Standard Webhooks specs
  - Update your webhook producer to match the standard webhook headers
  - Use the standardwebhooks library to sign your webhooks

Attempt	Delay
1	10 s
2	60 s
3	5 min
4	30 min
5	60 min
6	…

Code 3.2

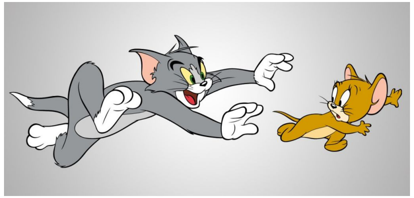


- **Extra goal #1:** try to improve logging in your event producer, message broker, and event consumer
    - Write clear and concise logs printed directly to console or…
    - Formatting logs with extra information and outputting to a file (try the winston javascript library for this)

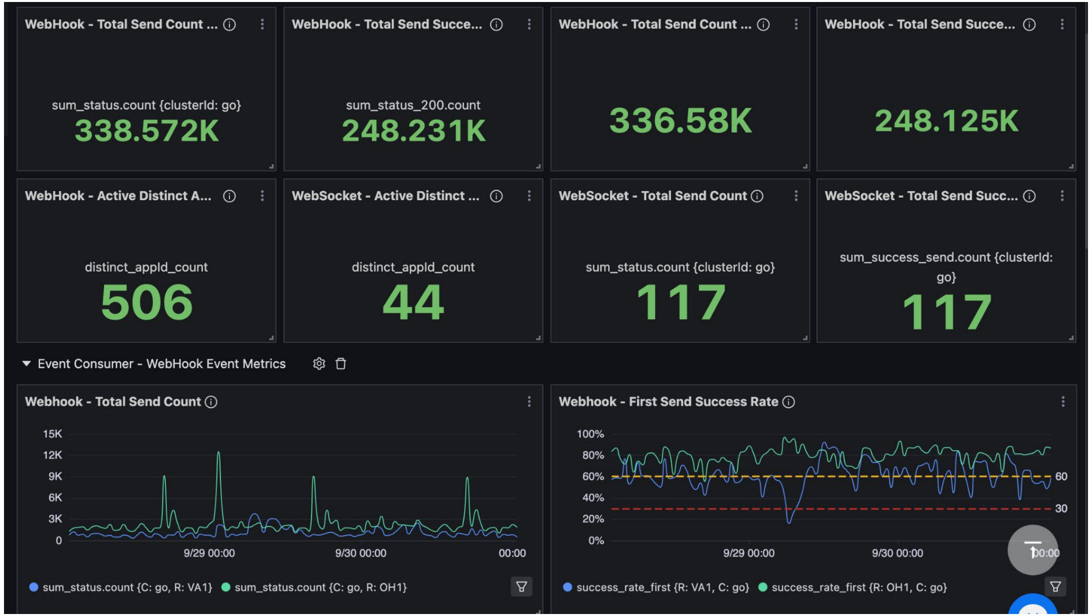


Time	message	@regionId	@timestamp	loglevel
2024-10-01T01:30:31.638Z	2024-10-01 01:30:31.638[][search-client-token-refresher][WARN][ZoomSearch:?] ? - Get token retry, Wait: 60000 ms, retry times: 27065!	VA1	2024-10-01T01:30:32.916Z	WARN
2024-10-01T01:30:31.638Z	2024-10-01 01:30:31.638[][search-client-token-refresher][ERROR][ZoomRestClientTransport:?] ? - GetToken fail ackingId: 92b9f26a-5b83-41ce-b67f-17c8f1e7fcb2 org.elasticsearch.client.ResponseException: method [POST], host [https://search.zipow.com], URI [/1.0/universal/client/newToken], status line [HTTP/1.1 502 Bad Gateway] <html> <head><title>502 Bad Gateway</title></head> <body> <center> 502 Bad Gateway </center> <hr><center>nginx</center> </body> </html> at org.elasticsearch.client.ZoomRestClient.convertResponse(ZoomRestClient.java:339) at org.elasticsearch.client.ZoomRestClient.performRequest(ZoomRestClient.java:306) at org.elasticsearch.client.ZoomRestClient.performRequest(ZoomRestClient.java:278) at co.elastic.clients.zoom.ZoomRestClientTransport.performRequest(ZoomRestClientTransport.java:511) at co.elastic.clients.zoom.ZoomRestClientTransport.getToken(ZoomRestClientTransport.java:170) at ...	VA1	2024-10-01T01:30:32.017Z	ERROR
2024-10-01T01:30:03.992Z	2024-10-01 01:30:03.992[asyncmqd842614cf06d4533ad553e148ab0b948][pool-8-rs4j-appPlatform-16][INFO][AppPlatformClient:?] ? - [AppPlatformClient#getCustomHeaderSetting] ---> POST {E}https://devmp.zoomdev.us/api/v1/event-consumer/subscription/customHeader{/E} START HTTP/1.1	VA1	2024-10-01T01:30:04.170Z	INFO
2024-10-01T01:30:03.992Z	2024-10-01 01:30:03.992[asyncmqd842614cf06d4533ad553e148ab0b948][pool-8-rs4j-appPlatform-16][INFO][AppPlatformClient:?] ? - [AppPlatformClient#getCustomHeaderSetting] ---> body: {E}["Y2mtWEO8SGOuMZg4ExPE9w","zRZawQkcRR21A41J8199eg"]{/E}	VA1	2024-10-01T01:30:04.170Z	INFO

Code 3.3



- **Extra goal #2:** attempt to add some sort of metric collector to your webhook consumer
  - Utilize components you already have built! You have a redis cache and a SQLite database ready to go
  - Track thing such as: events triggered, success/failure rates, time webhooks take to send, endpoints etc.



SECTION 4: Future Improvements

Code Recap!

https://github.com/jerryang1023/jax-webhook-2024/tree/workshop-part-3

Some issues to consider…

Don’t roll your own crypto

- Let’s implement asymmetric signatures for webhooks: easy!
- One keypair for the service, just put public key on the website.
- Win! 🎉

Don’t roll your own crypto

- Other customers of your service will be able to send each other messages.
  - These messages will pass validation!
- Easy: let’s add a “user identifier” in a header that people can check.
- Win! 🎉

Don’t roll your own crypto

- You need to include that header as part of the signature (otherwise it can be faked).
- User identifiers have to be generated by the service and be immutable.
- Your customers need to ACTUALLY verify it: unlikely.

Don’t roll your own crypto (better)

- Don’t send the user identifier, but make it be part of the signature.
- This way people will need to store their user identifier and will be forced to pass it to the signature.
- Win (this time for real)! 🎉

You can’t guarantee webhook ordering

- Failed deliveries will have to block all of the deliveries (denial of service)
- Even if webhooks are sent in order, they may not be processed in order due to differences in the processing speed of the webhook handlers.
- Waiting for the processing of the first webhook to complete before sending the second would significantly limit delivery rate.

- Read more: https://www.svix.com/blog/guaranteeing-webhook-ordering/

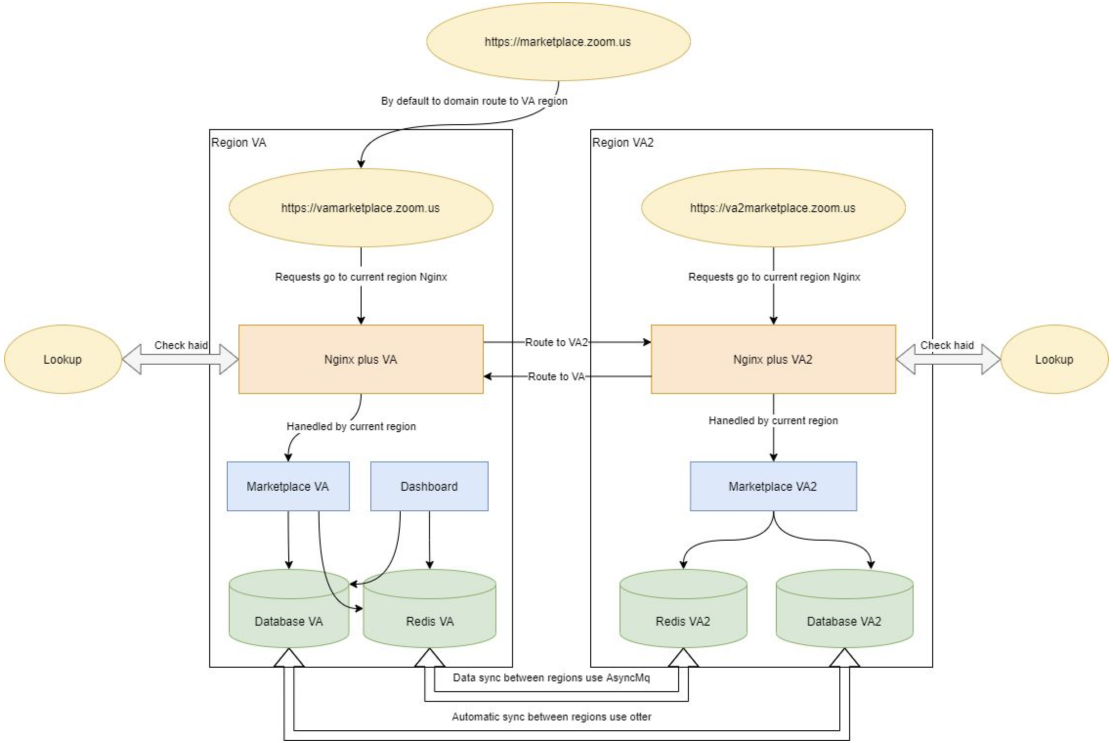
Server side request forgery (SSRF)

- SSRF allows attackers to manipulate server-side requests to call different destinations - inherent with webhooks!
- Common targets include internal networks and services.
- Mitigation strategies include network segmentation, proxying, and application-level request blocking based on IP ranges.

Some improvements to be made…

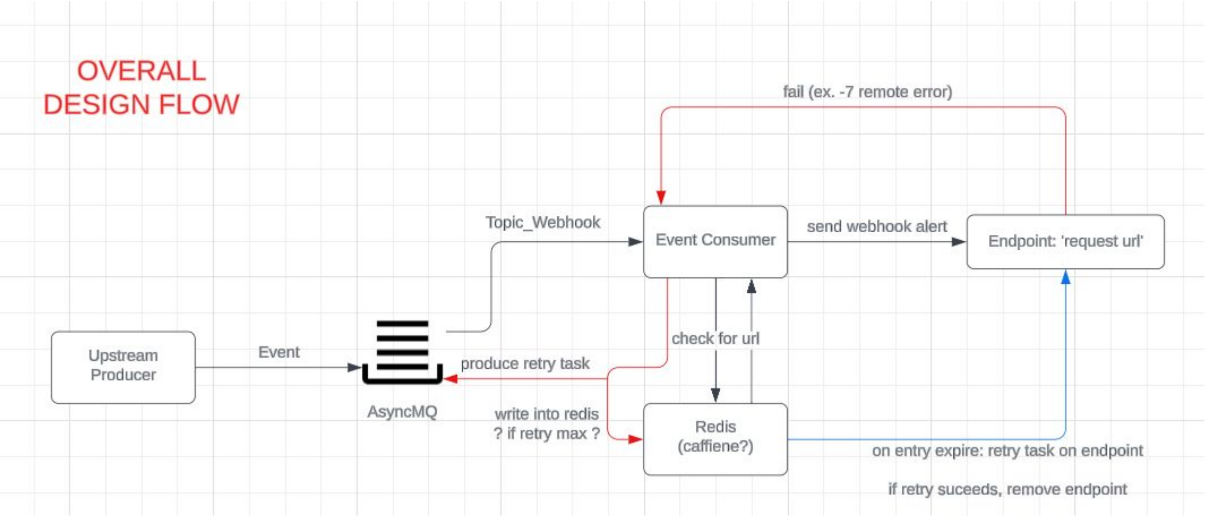
Active-Active deployments

- High availability architecture
- Relies on having separate deployments in (ideally) different **physical** locations
- Reroutes API calls through a gateway/proxy server
- Offers both failover and load balancing



Fast Fail Error Handling

- Problem: Customer endpoints can go down or become unavailable all the time. Why waste our own resources when it could possibly be their problem?
- Solution: Fast fail on those endpoints



Lets get back to coding…

Code 4.1

- **Main goal #1:** Nothing concrete here, sorry!
  - Try to implement a fix to one problems described above
  - Go back to work on previous sections of code

EDA: Key Takeaways

1. **Enables Real-Time Responsiveness**
2. **Operational Efficiency Through Dynamic Resource Allocation**
3. **Decouples System Components**
4. **Improves Scalability and Flexibility**
5. **Future of Developer Ecosystems**

DevOpsCon by devmio

Any Questions ?

We ask for **your feedback!**

PLEASE VOTE NOW!

★★★★★

devmio