



Build a Multiplayer Real-time Game with Astra Streaming

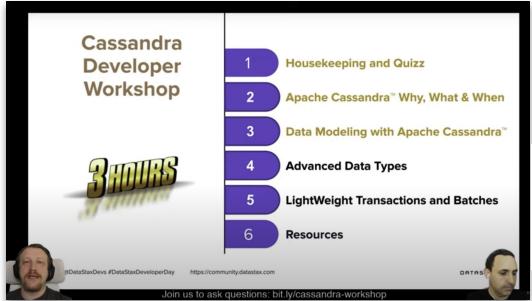


LEVEL
UP
with the

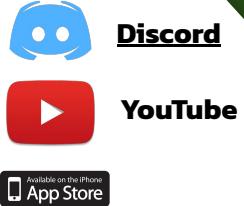
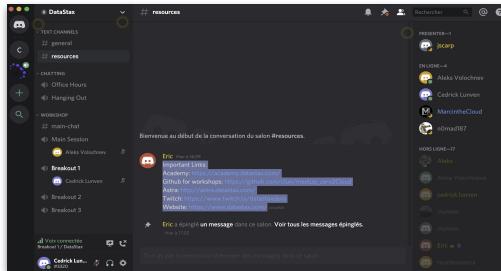
DataStax
Developers

Housekeeping #1: Attending the workshop

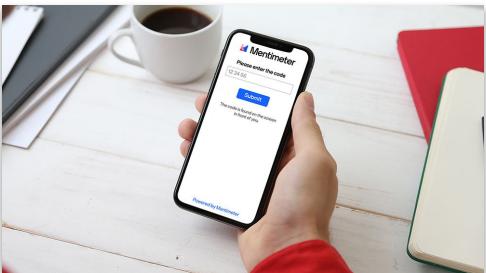
Livestream: youtube.com/DataStaxDevs



Questions: <https://dtsx.io/discord>



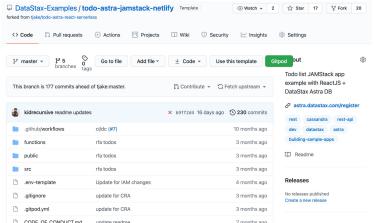
Games menti.com



Housekeeping #2: Doing Hands-On

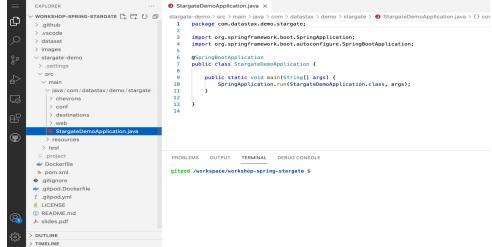
Nothing to install !

Source code + exercises + slides



A screenshot of a GitHub repository page for 'DataStax-Examples / todo-astra-jamstack-netlify'. It shows a file tree with 'src/main/java/com/datasstax/demo/stargate' and a commit history for 'Add Stargate demo'.

IDE



A screenshot of an IDE (IntelliJ IDEA) showing the code for 'StargateDemoApplication.java'. The code defines a main method that runs a Spring Boot application.

```
package com.datasstax.demo.stargate;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class StargateDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(StargateDemoApplication.class, args);
    }
}
```



Database + GraphQL + PlayGround



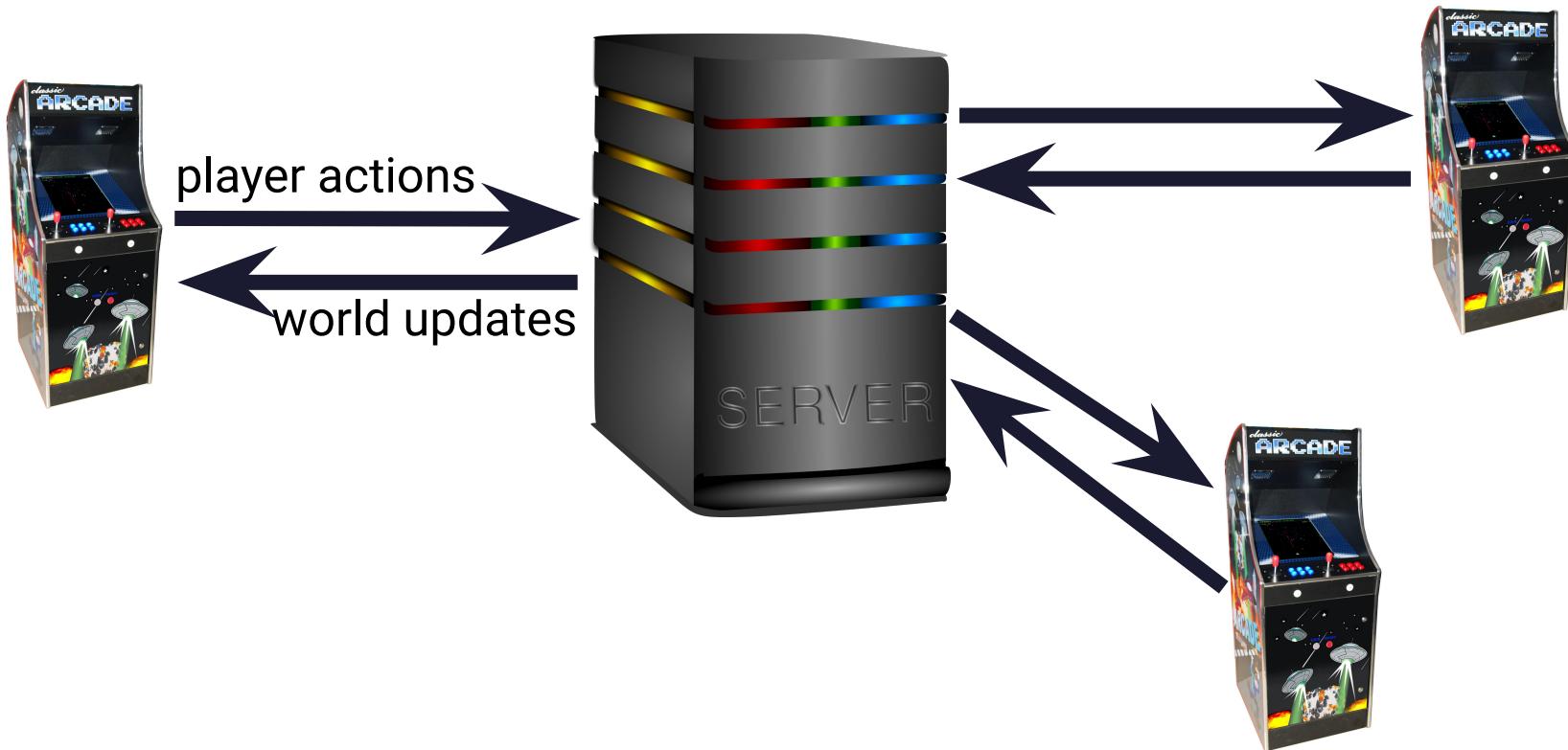
DataStax
Astra DB

DataStax Developers

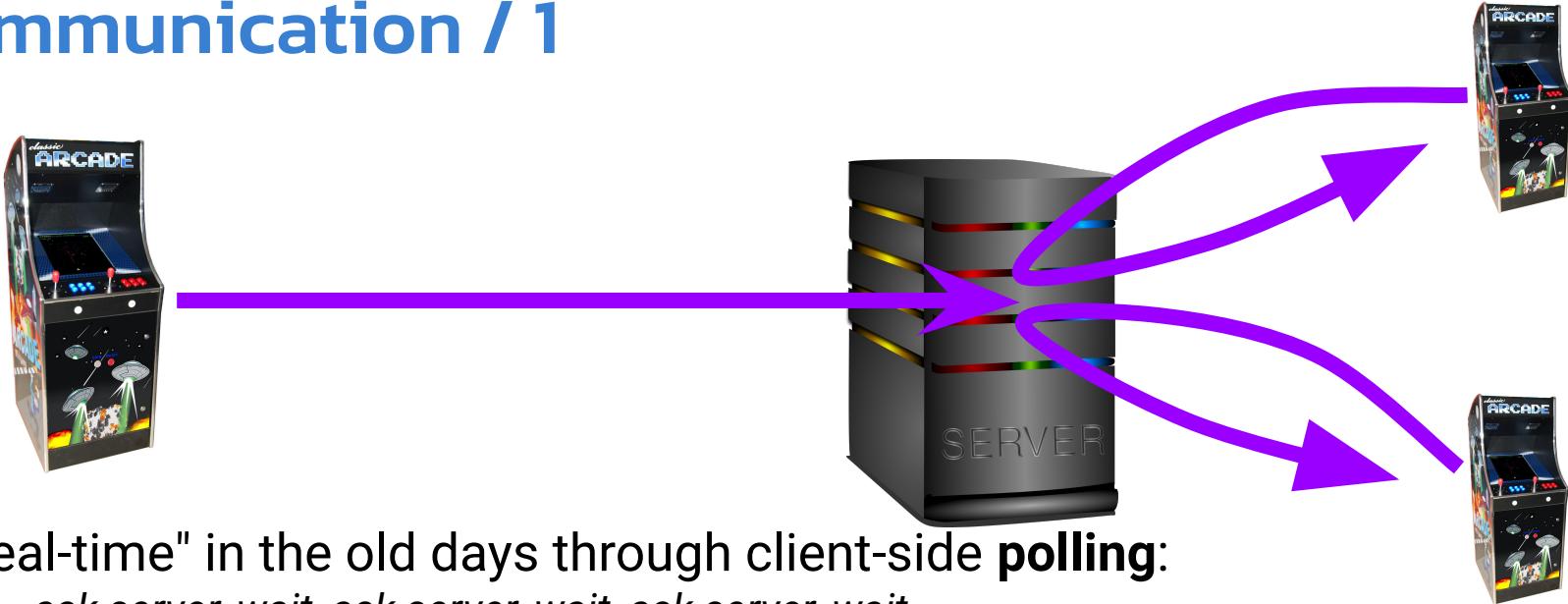
Let's start!



So you want a multiplayer real-time game ... ?



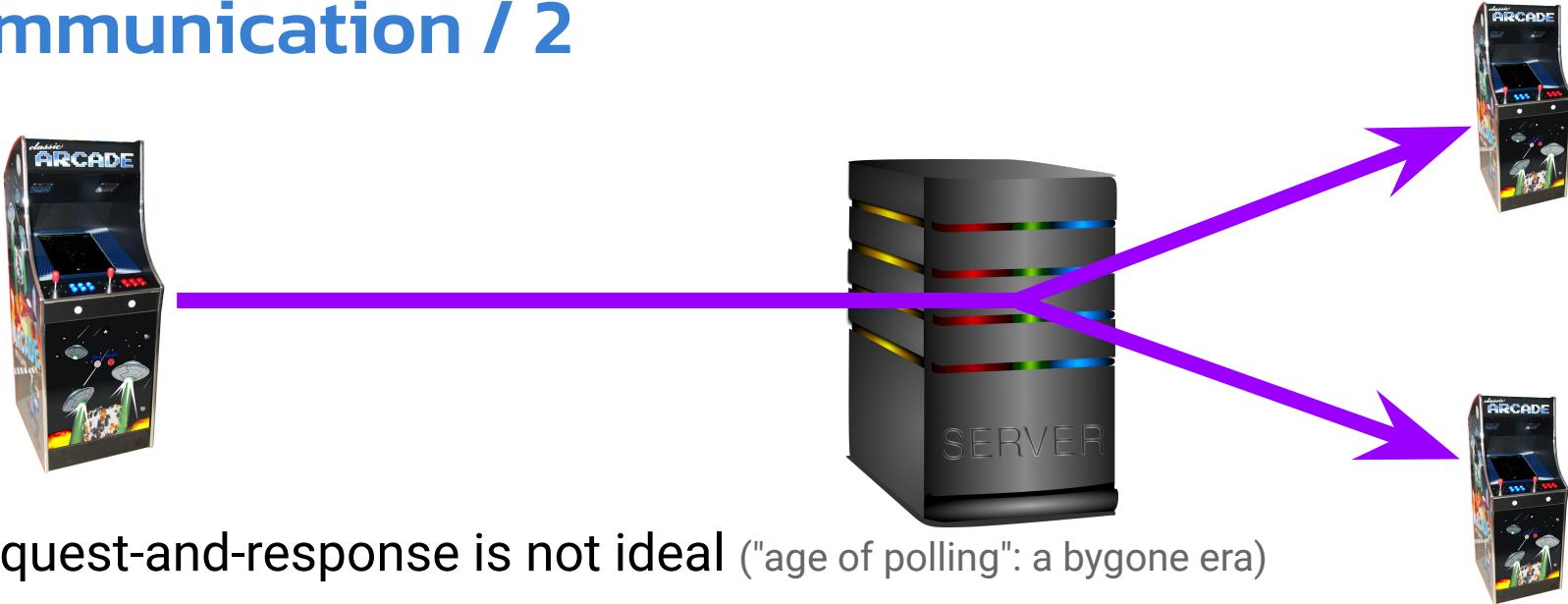
Communication / 1



"Real-time" in the old days through client-side **polling**:
ask server, wait, ask server, wait, ask server, wait ...

- ✗ A flood of request-and-responses (overhead!)
- ✗ Not really real-time (latencies)
- ✗ Waste of server resources
- ✗ Sometimes a mess to handle in the application

Communication / 2



Request-and-response is not ideal ("age of polling": a bygone era)

Rather: server pushing data to clients at once.

⇒ We need a messaging platform

⇒ And server-initiated communication

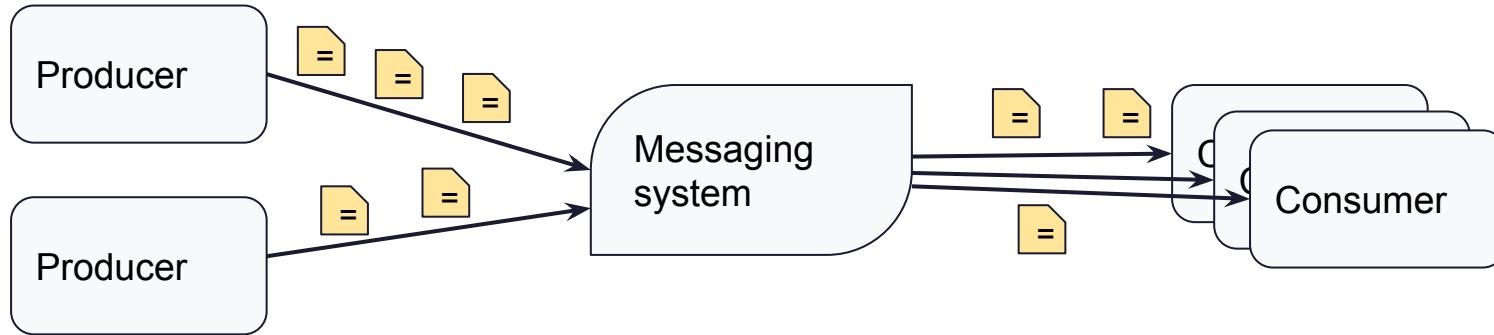
Messaging systems

- **Born out of the need for:**
 - non-blocking communication
 - "push, not pull"
 - decoupling
 - faster response to new data
- **Good match with modern microservice architectures**
- **Receive and deliver "messages"**
 - decouple sending and receiving (also: cross-language)
 - Act as a "buffer" for communication
 - various messaging patterns (queue, pub/sub)
 - (optional) Message schemas: JSON, Avro, ...

Messaging systems

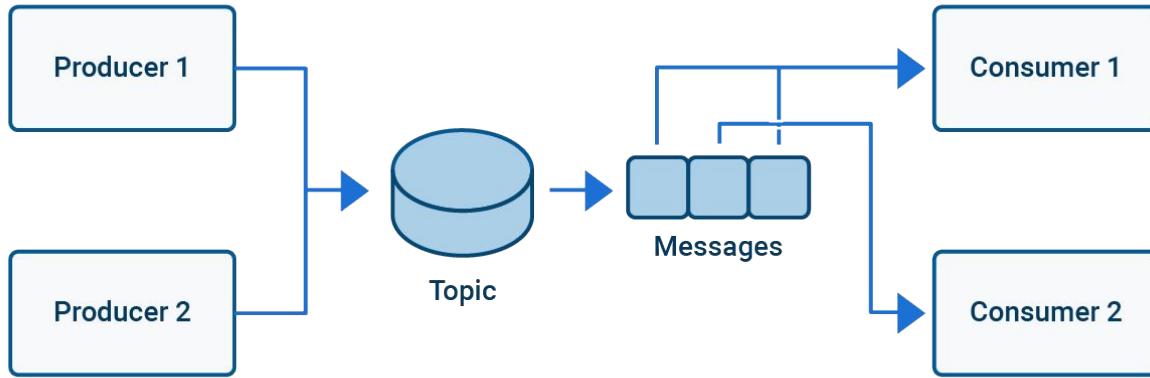
- **We live in a "golden age for messaging services"**
RabbitMQ 2007, Apache Kafka 2009, Apache Pulsar 2016
- **Use cases for messaging:**
 - the dataset is never complete
 - continuous stream of events/datapoints
 - shared queues (items to process)
 - ...
- **In practice:** IoT, social media, car automation, finance, online gaming, ...

Messaging at a glance



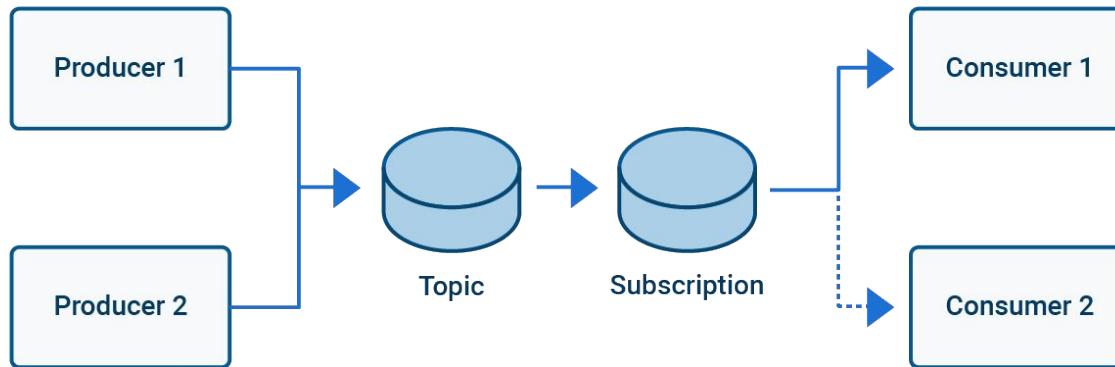
- **Asynchronous ⇒ messages retained**
- **Delivery semantics ⇒ receipt acknowledgement**
(the hard part is correct delivery)

Message Consumption – Queuing



- Exactly-once semantics
- Backlog of messages
- Distributed processing
- Out-of-order possible
- Message failover
- Easy to scale

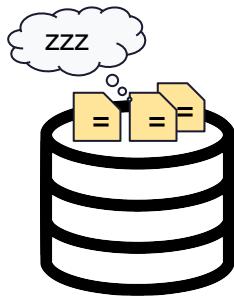
Message Consumption - Pub/Sub



- Producers publish to Topic
- Consumers subscribe to Topic
- Multiple consumers
- Easy to extend

"**Each** player shall receive **all updates** to the game world"

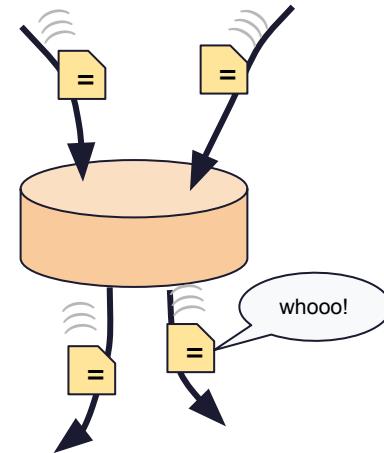
Messaging != databases



Databases
data at rest



Messaging
data in transit



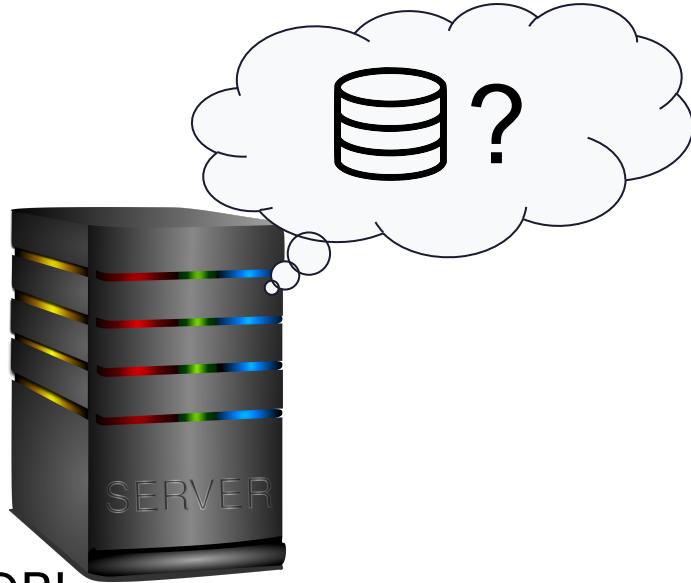
But: blurred boundaries

Most messaging systems offer some *persistence*

Some databases equipped with *CDC* ("change data capture")

Mixing the two: a very common pattern anyway

Storage (?)



This workshop is "messages only", no DB!

(a richer game will need some persistent storage...)

A tricky business

High-performance, correct delivery, failover, replication, scaling...

Technically it's a tough challenge

Consider also operations/management

... that's why today we rely on a managed solution!

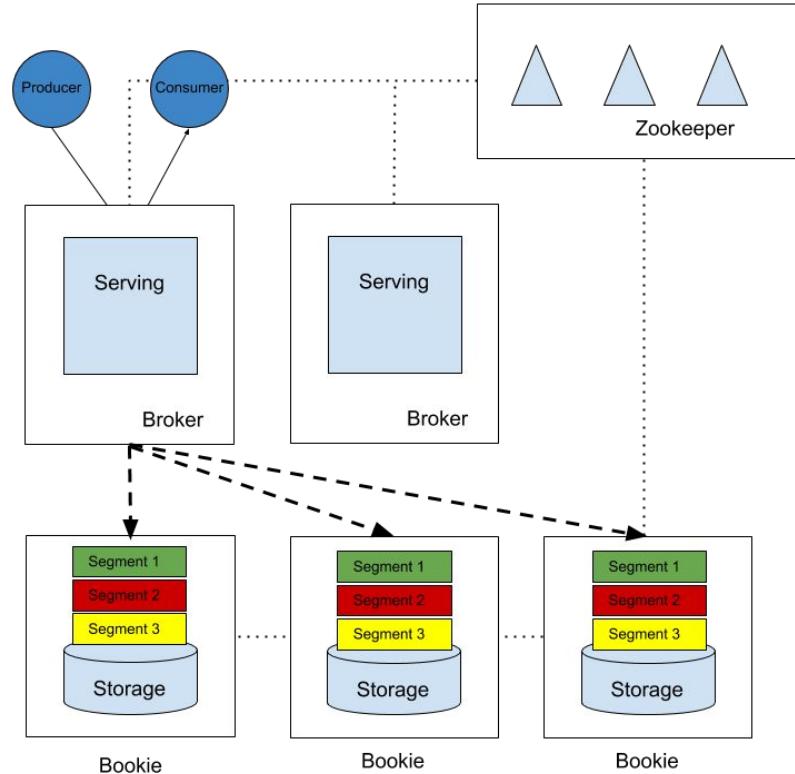
Apache Pulsar ⇒ Astra Streaming



- Yahoo ⇒ Apache Foundation
 - Supports pub/sub and queueing
 - Split compute from storage
 - Modern features:
 - Multi-tenancy
 - Horizontal scalability
 - High performance / Low latency
 - Geo-replication
- + schemas, functions, Pulsar SQL, deduplication, IO (sinks/sources) ...
- Managed Pulsar in the cloud currently FREE open beta
 - Create ready-to-use streaming topics
 - Tight integration with Astra DB available (as between Pulsar and Cassandra)

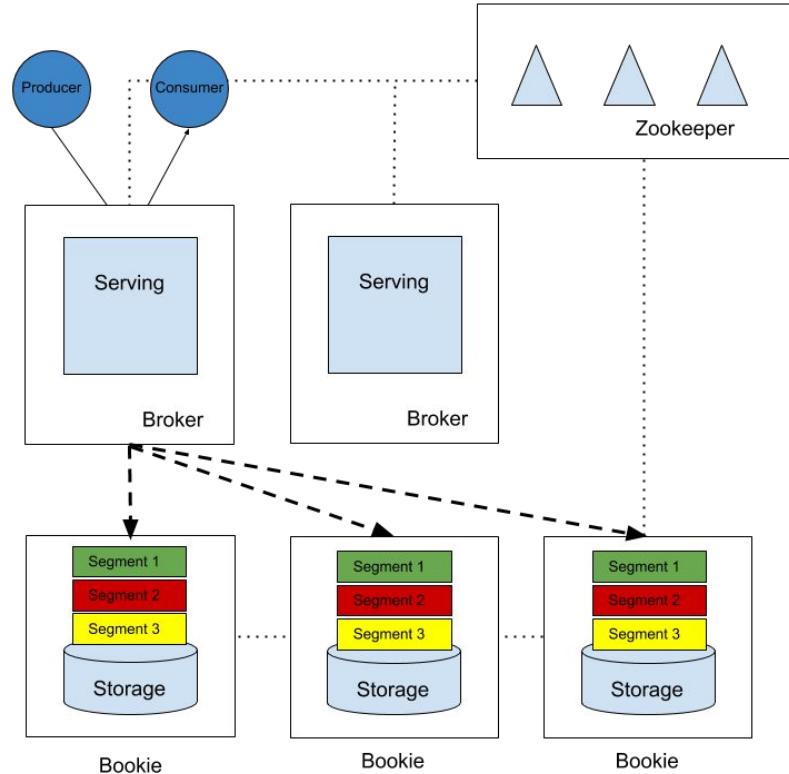
Pulsar architecture

- Distributed, tiered architecture
- Separates compute from storage
- Zookeeper holds metadata for the cluster
- Stateless Broker handles producers and consumers
- Storage is handled by Apache Bookkeeper



Pulsar architecture (cont'd)

- BookKeeper (“bookies”) distributed, append-only log
- Data is broken into ledgers and segments written to multiple bookies
- Producer acknowledged when quorum of bookies acknowledge
- No single bookie holds entire log



HandsOn: Streaming topic



Astra
STREAMING

Get your instance here:

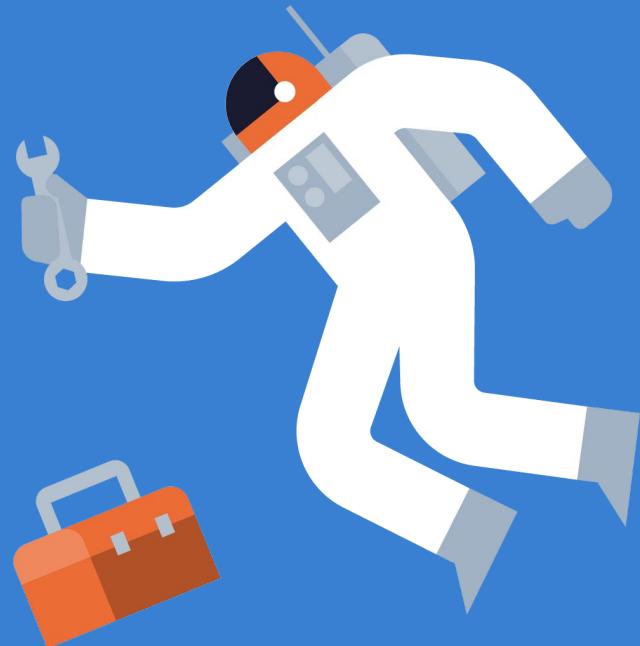
- <https://astra.dev/11-17>



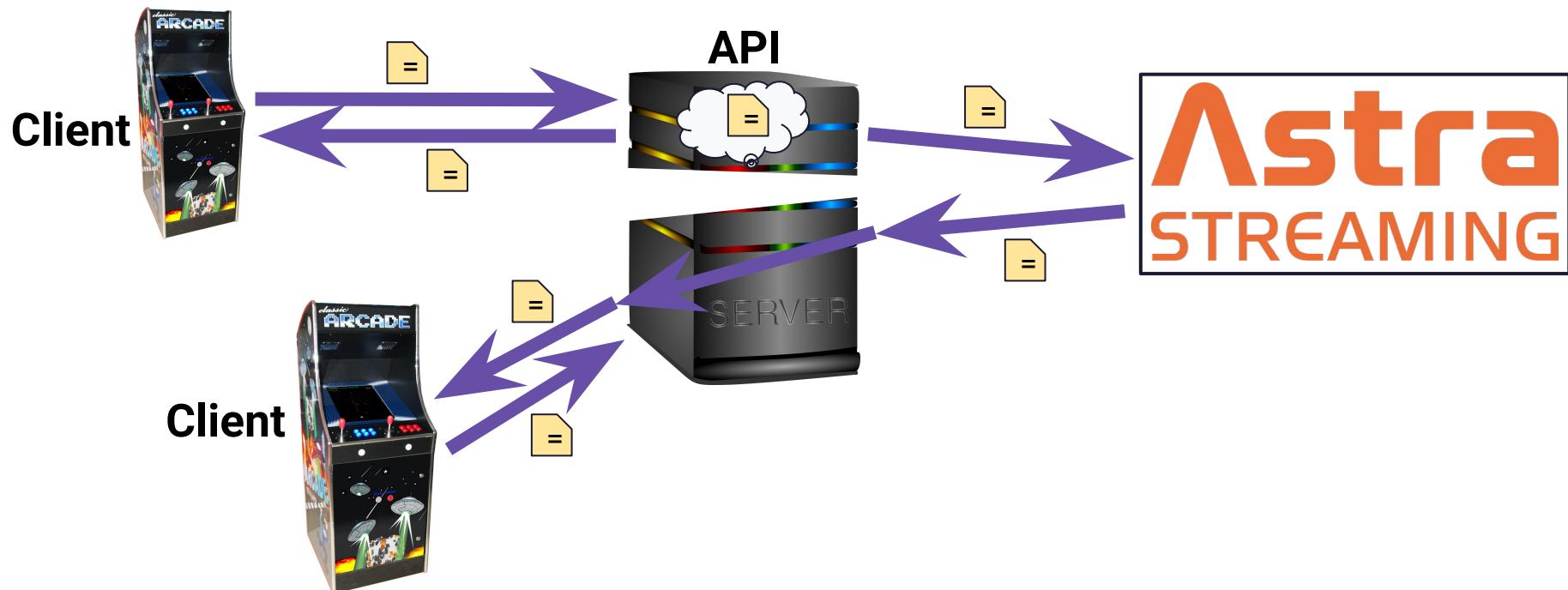
GitHub

Repository:

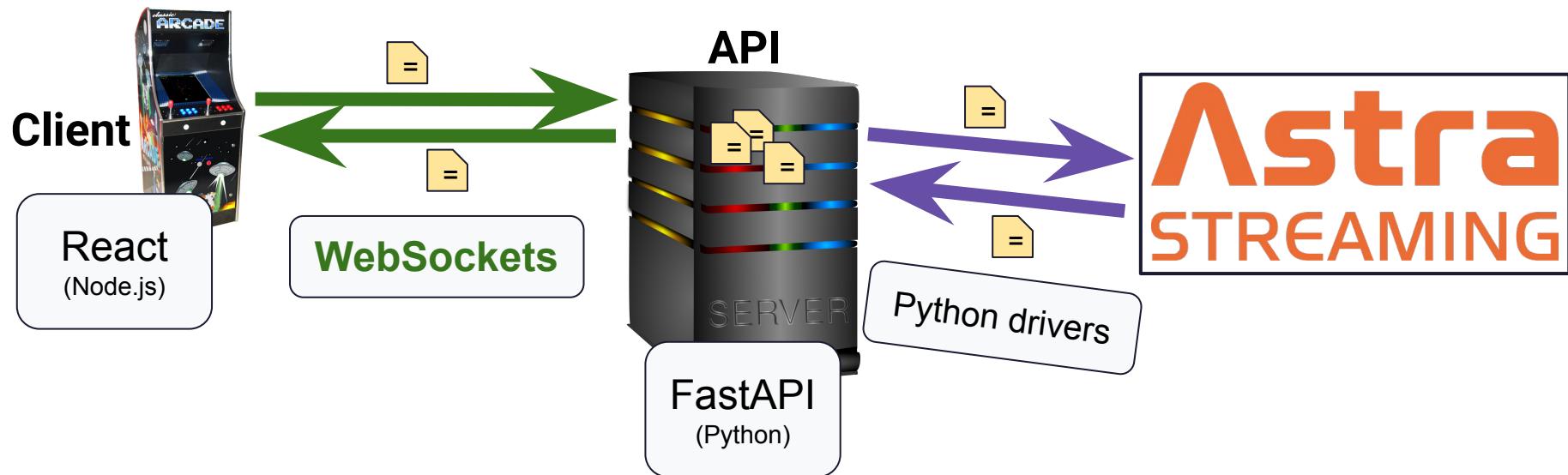
- <https://github.com/datastaxdevs/workshop-streaming-game>



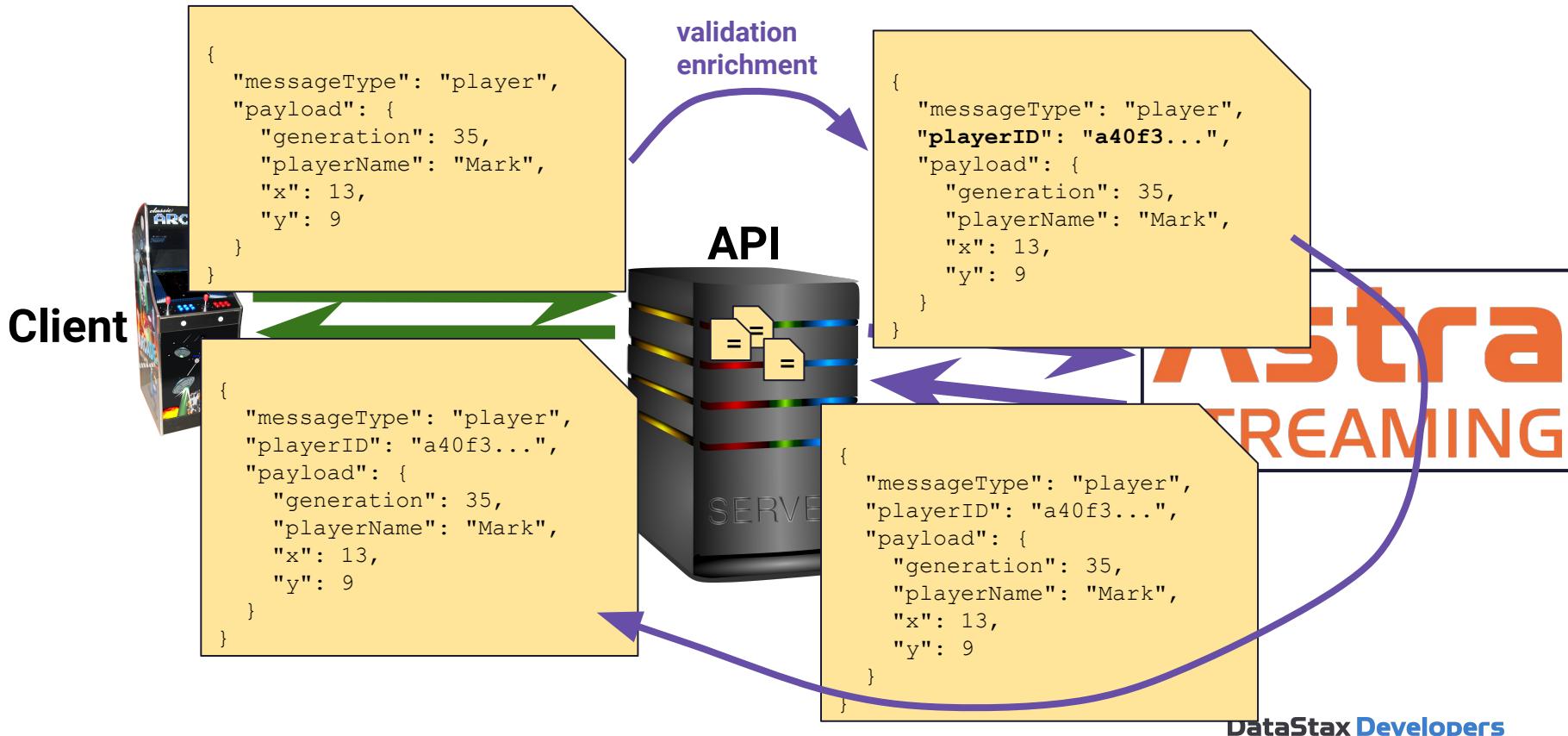
Game architecture, revisited



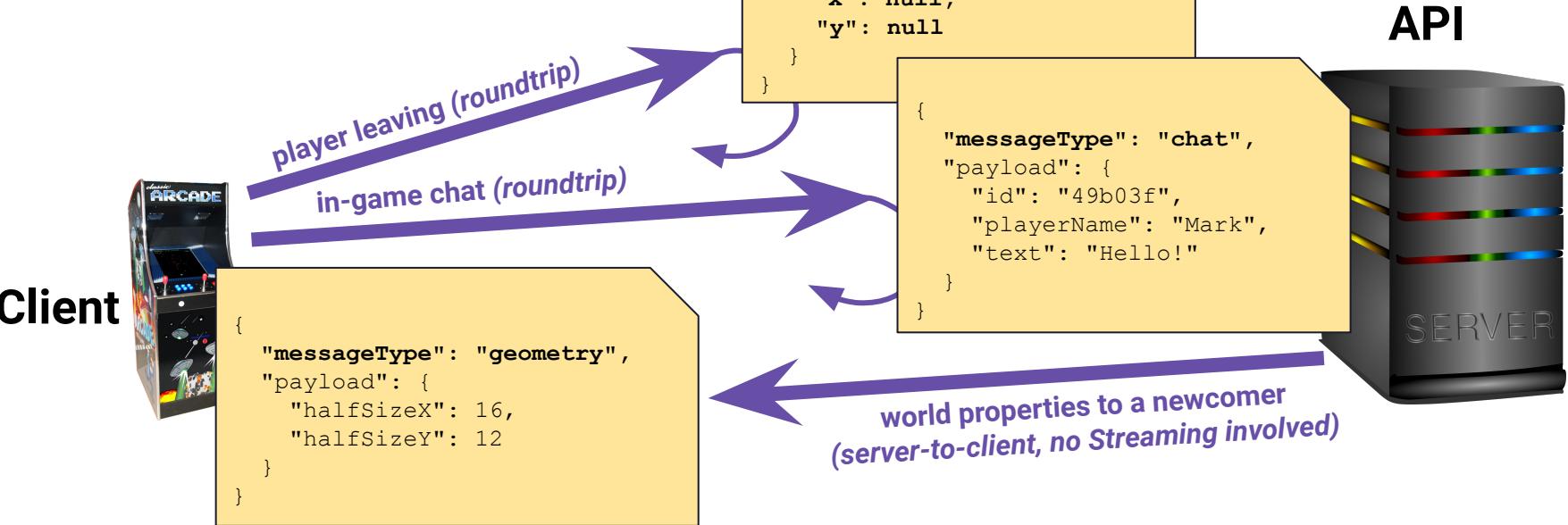
Technologies



Messaging protocol



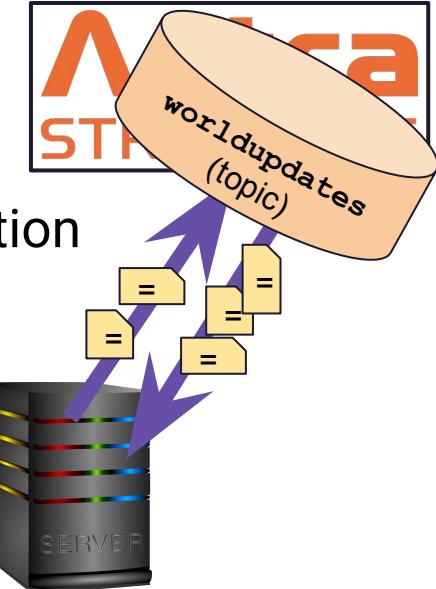
Other messages



Architectural choices

All messages through a single Streaming topic:

- no schema, just (JSON) message serialization
- simplified setup, focus on core concepts
- in a real-life app: several topics



Two WebSockets per client:

"player": client-to-server messages
"world": server-to-client messages



HandsOn: Gitpod setup



Astra
STREAMING

Get your instance here:

- <https://astra.dev/11-17>



GitHub

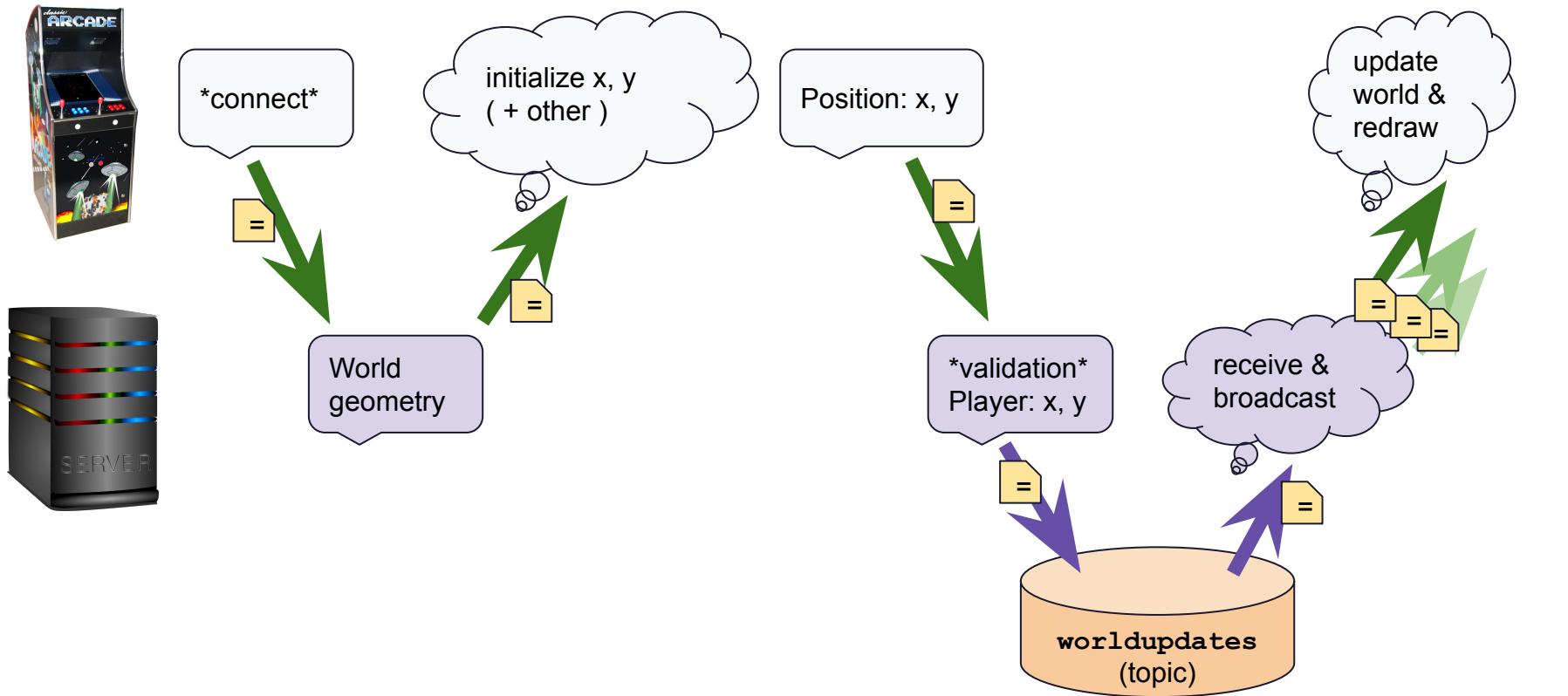
Repository:

- <https://github.com/datastaxdevs/workshop-streaming-game>



When the client starts...

time



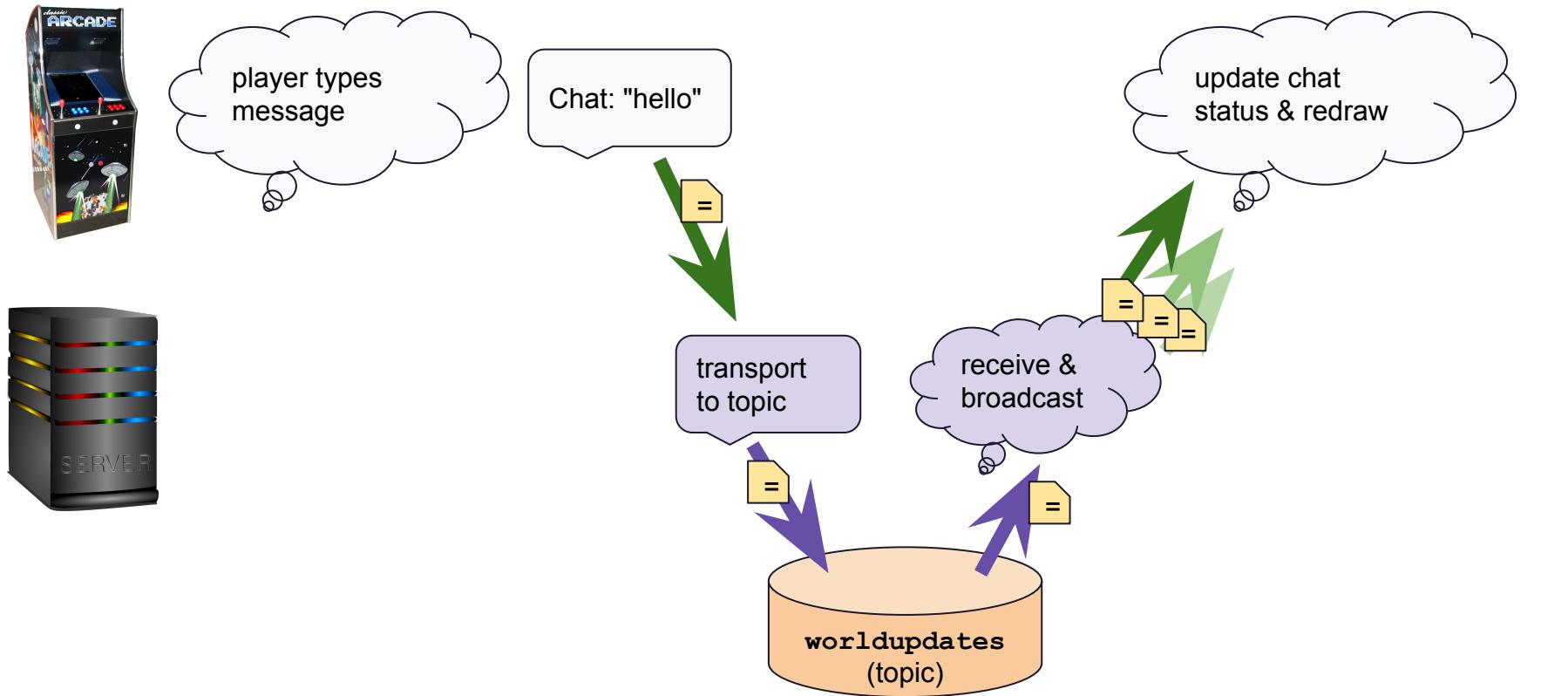
Gameplay

time



In-game chat

time



HandsOn: PLAY!



Astra
STREAMING

Get your instance here:

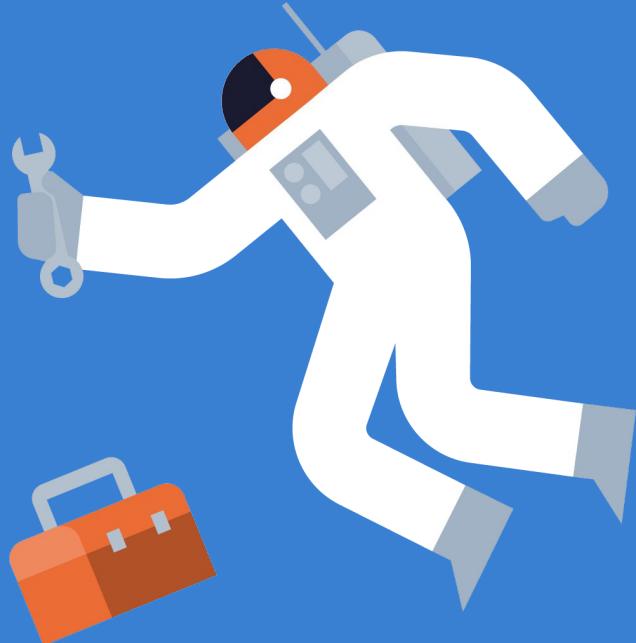
- <https://astra.dev/11-17>



GitHub

Repository:

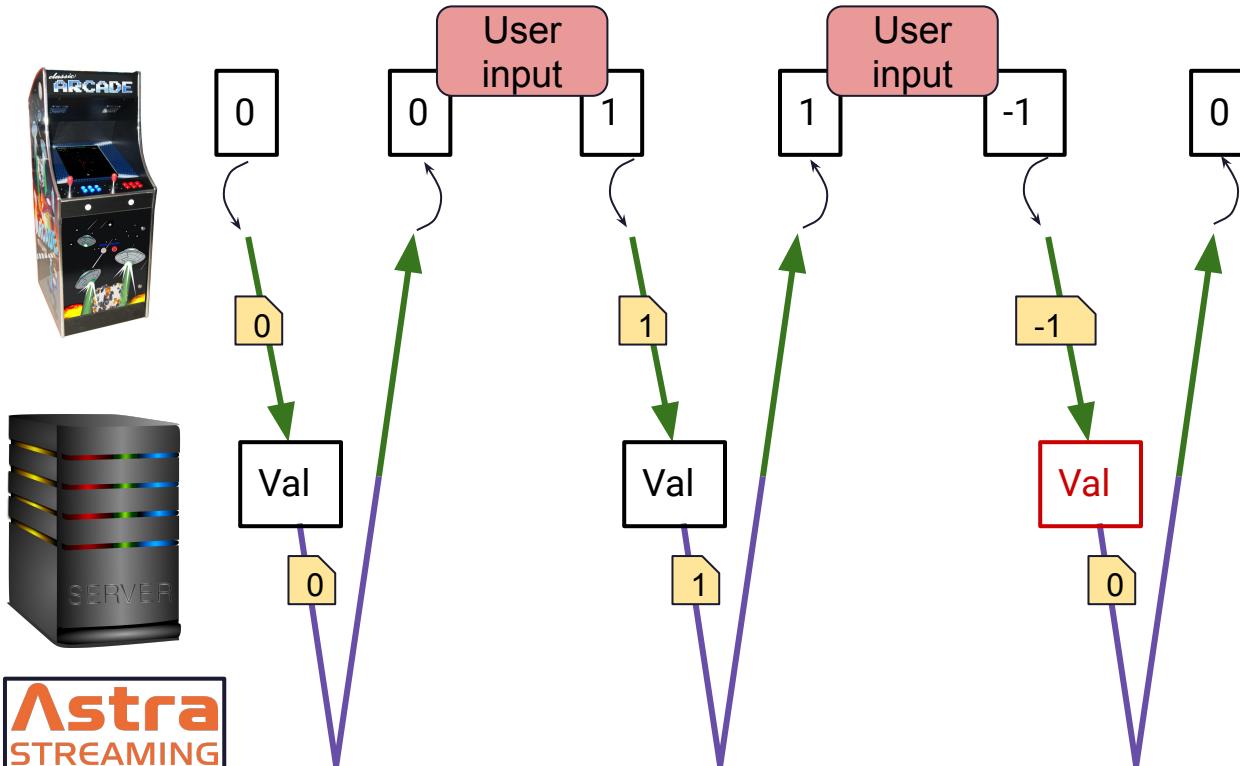
- <https://github.com/datastaxdevs/workshop-streaming-game>



Async and server-side validation

time 

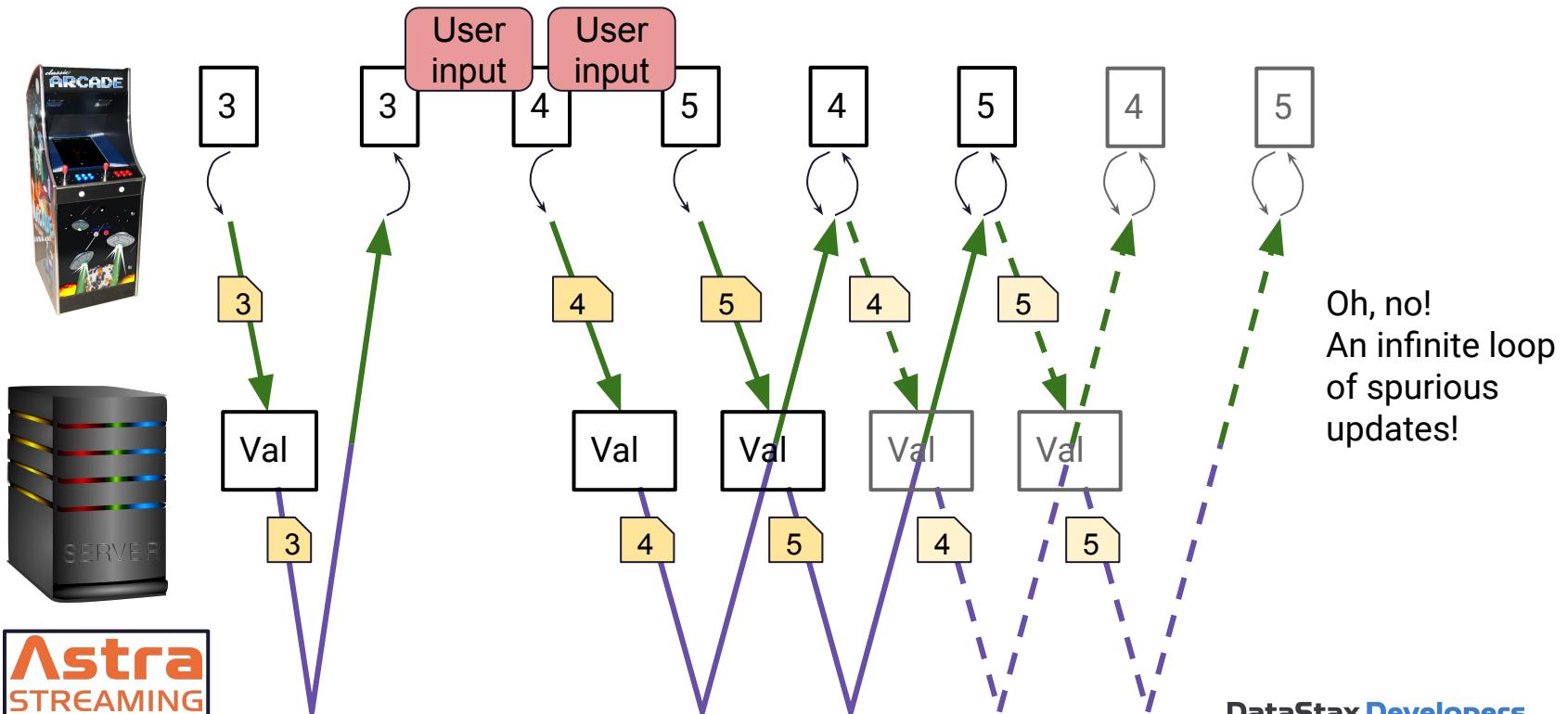
Position is in client memory and validated by the API



Async and server-side validation / 2

time →

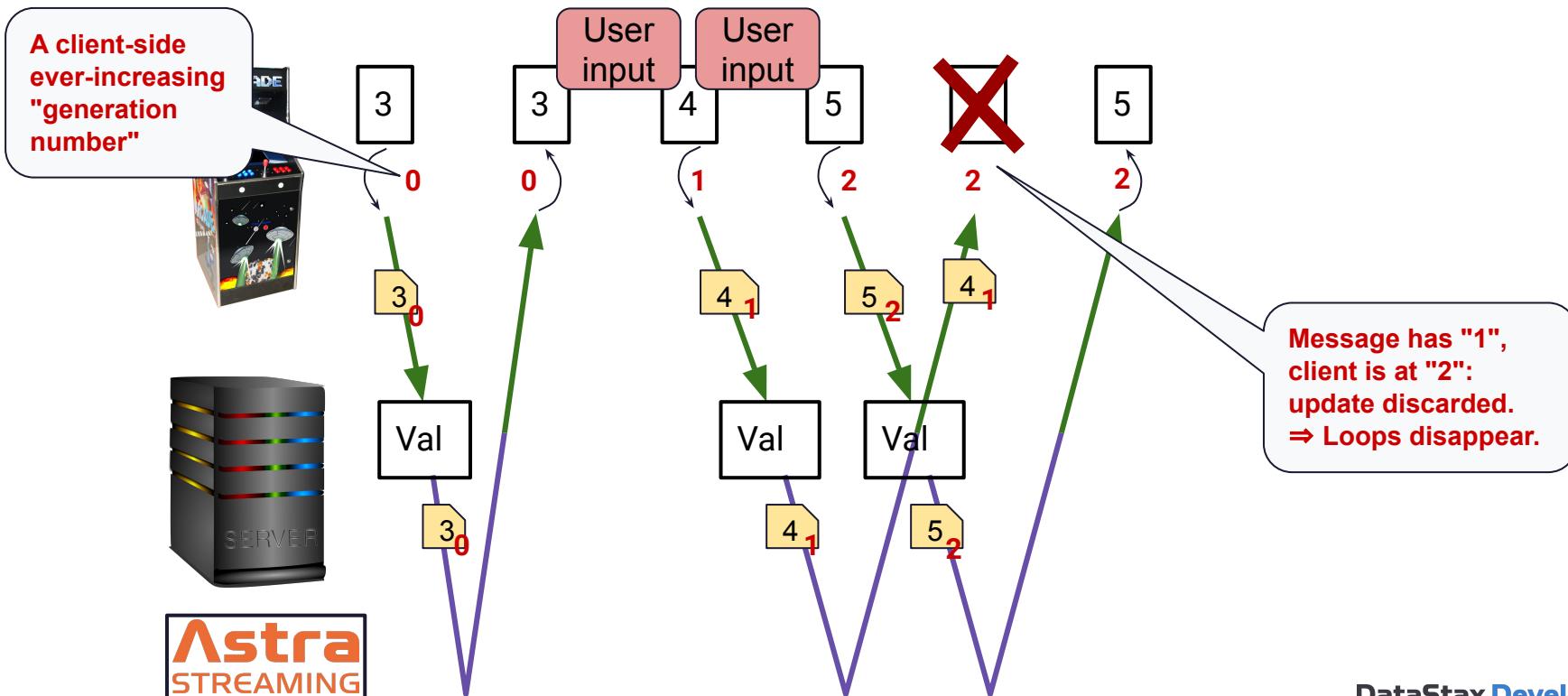
Client receives position updates (and may fire new updates)



Async and server-side validation / 3

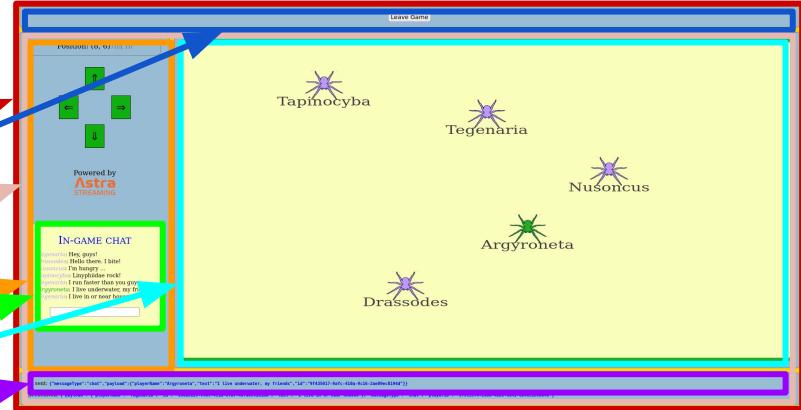
time →

Solution: discard **obsolete** server updates



Client app structure

```
1 | let pws = null
2 | let wws = null
3 |
4 | const App = () => {
5 |
6 |   // React states (useState)
7 |   // React references (useRef)
8 |
9 |   // keyDown handler
10 |
11 |   useEffect( () => {
12 |     if (inGame) {
13 |       if ( websockets === null ) {
14 |         websockets.onmessage = evt => {
15 |           // parse and interpret message
16 |           // depending on messageType (player/geometry/chat)
17 |           // HERE WE MODIFY REACT STATE
18 |         }
19 |       } else {
20 |         // send "goodbye" to API
21 |         websockets.disconnect()
22 |         websockets = null
23 |       }
24 |     }
25 |   }, [inGame])
26 |
27 |   useEffect( () => {
28 |     // send player update to the API
29 |   }, [coordinates, etc])
30 |
31 |   return (
32 |     App
33 |     PlayerForm
34 |     GameArea // if inGame
35 |     sidebar
36 |     GameInputs
37 |     buttons, etc
38 |     ChatArea
39 |     GameField
40 |     statusbar
41 |   )
42 | }
```



React & WebSockets: useRef

WebSocket on client side: **subscription to events**

Problem: we're closing over a React state

```
import { useState } from "react"

const [generation, setGeneration] = useState(0)

// WebSocket event subscription
wws.onmessage = evt => {
  // Stale value because of closure!
  console.log(generation)
}
```

```
import { useState, useRef } from "react"

const [generation, setGeneration] = useState(0)

const generationRef = useRef()
generationRef.current = generation

// WebSocket event subscription
wws.onmessage = evt => {
  // This is what we want!
  console.log(generationRef.current)
}
```

Solution: useRef to access a state within a closure!



A modern, high-perf Python library to create APIs

- HTTP/2 & WebSockets support
- Minimal boilerplate
- Sophisticated route dependency handling
- Asynchronous post-response task support
- Well documented (fastapi.tiangolo.com)

FastAPI, code layout

Routes = functions, return value = response (etc etc...)

What about WebSockets?

```
app = FastAPI()

@app.websocket("/ws/world/{client_id}")
async def worldWSRoute(worldWS: WebSocket, client_id: str):
    await worldWS.accept()
    #
    pulsarClient = getPulsarClient()
    pulsarConsumer = getConsumer(client_id, pulsarClient)
    #
    try:
        geomUpdate = makeGeometryUpdate(HALF_SIZE_X, HALF_SIZE_Y)
        await worldWS.send_text(json.dumps(geomUpdate))

        while True:
            worldUpdateMsg = receiveOrNone(pulsarConsumer, RECEIVE_TIMEOUTS_MS)
            if worldUpdateMsg is not None:
                await worldWS.send_text(worldUpdateMsg.data().decode())
            await asyncio.sleep(SLEEP_BETWEEN_READS_MS / 1000)

    except WebSocketDisconnect:
        pulsarConsumer.close()
```

WebSocket decorator

function definition

once-per-client part

the function runs for a long time ...

detect client-side disconnect

Pulsar drivers for Python

```
# pip install pulsar-client==2.7.3
import pulsar

client = pulsar.Client(
    SERVICE_URL,
    authentication=pulsar.AuthenticationToken(ASTRA_TOKEN),
    tls_trust_certs_file_path=TRUST_CERTS,
)

streamingTopic = 'persistent://gameserver-0123/default/worldupdates'

# consume messages
consumer = client.subscribe(streamingTopic, 'this-subscription-id')
receivedMessage = consumer.receive(timeout)
# (process message ...)
consumer.acknowledge(receivedMessage)

# alternatively: client.subscribe(... message_listener=myListenerFunction)

# produce messages
producer = client.create_producer(streamingTopic)
pulsarProducer.send('{"message":"hello!"}'.encode('utf-8'))
```

Python quickstart on Astra Streaming UI

The screenshot shows the Astra Streaming UI interface. On the left, there's a sidebar with navigation links for Dashboard, Databases (listing hemidactylus, techblog, measurements, workshops, multics), and Streaming (listing stefano, og1, snktest). Below these are Sample App Gallery and Other Resources. The main content area is titled "Dashboard / snktest". It has tabs for Quickstart, Try Me, Connect, Topics, Functions, Sinks, Sources, Namespaces, and Settings. Under "Connect", the Python tab is selected, showing Python Code Samples. It explains that Astra Streaming is powered by Apache Pulsar and provides links for PyPi and documentation. It also notes that Python versions 3.4 to 3.7 are supported. Below this, it says "Astra Streaming is running Pulsar version 2.7.2. You should use this API version or higher." Under "CONSUMER", it shows configuration fields for Cluster (pulsar-gcp-europewest1), Namespace (default), Topic (jsnktopic), and Subscription (my-subscription). At the bottom, there's a code editor window displaying Python code for connecting to a Pulsar topic:

```
import pulsar, time

service_url = 'pulsar+ssl://pulsar-gcp-europewest1.streaming.datastax.com:6651'

# Use default CA certs for your environment
# RHEL/CentOS:
trust_certs='/etc/ssl/certs/ca-bundle.crt'
# Debian/Ubuntu:
# trust_certs='/etc/ssl/certs/ca-certificates.crt'

token=''
```

Python: Pulsar+WebSockets

```
try:  
    # first we tell this client how the game-field looks like  
    geomUpdate = makeGeometryUpdate(HALF_SIZE_X, HALF_SIZE_Y)  
    # Note: this message is built here (i.e. no Pulsar involved)  
    # and directly sent to a single client, the one who just connected:  
    await worldWS.send_text(json.dumps(geomUpdate))  
    while True:  
        worldUpdateMsg = receiveOrNone(pulsarConsumer, RECEIVE_TIMEOUT_MS)  
        if worldUpdateMsg is not None:  
            # We forward any update from  
            # for all clients out there:  
            await worldWS.send_text(worldUpdateMsg)  
        await asyncio.sleep(SLEEP_BETWEEN_UPDATES)  
    except WebSocketDisconnect:  
  
def receiveOrNone(consumer, timeout):  
    """  
    A modified 'receive' function for a Pulsar topic  
    that handles timeouts so that when the topic is empty  
    it simply returns None.  
    """  
    try:  
        msg = consumer.receive(timeout)  
        return msg  
    except Exception as e:  
        if 'timeout' in str(e).lower():  
            return None  
        else:  
            raise e
```

React & SVG graphics

```
const GameField = ({playerMap, playerID, boardWidth, boardHeight}) => {

  return (
    <svg className="game-field" width="100%" height="700" preserveAspectRatio="none"
      viewBox={`0 0 ${100 * boardWidth} ${100 * boardHeight}`}>
      <defs>
        <pattern id="lyco_other" x="50" y="50" width="100" height="100" patternUnits="userSpaceOnUse">
          <image x="0" y="0" width="100" height="100" href="lyco_other.svg"></image>
        </pattern>
        <pattern id="lyco_self" x="50" y="50" width="100" height="100" patternUnits="userSpaceOnUse">
          <image x="0" y="0" width="100" height="100" href="lyco_self.svg"></image>
        </pattern>
        <pattern id="hearts" x="65" y="40" width="100" height="100" patternUnits="userSpaceOnUse">
          <image x="20" y="10" width="60" height="60" href="hearts.svg"></image>
        </pattern>
      </defs>
      <rect width={100 * boardWidth} height={100 * boardHeight} style={{fill: '#f9ffbb'}} />
      { Object.entries(playerMap).map( ([thatPlayerID, thatPlayerInfo]) => {
        const patternName = thatPlayerID === playerID ? 'lyco_self' : 'lyco_other'
        const playerClassName = thatPlayerID === playerID ? 'player-self' : 'player-other'
        return (<g key={thatPlayerID} transform={`translate(${thatPlayerInfo.x * 100}, ${thatPlayerInfo.y * 100})`}>
          <g transform='translate(50,50)'>
            <rect x="-50" y="-50" height="100" width="100" fill={`url(#${patternName})`} />
            {thatPlayerInfo.h && <rect x="-50" y="-50" width="100" height="100" fill='url(#hearts)'} />
            <g transform={`translate(0, ${thatPlayerInfo.y + 1} >= boardHeight ? -70 : 70)`}>
              <text className="player-name" textAnchor='middle' fontSize='40'>
                {thatPlayerInfo.playerName}
              </text>
            </g>
          </g>
        </g>
      ))}
    </svg>
  )
}
```

Ideas for improvement

- players change the world/interact with others
- schemaful messages (⇒ several topics)
- a sink to Astra DB for easy persistence: uses

Thank you!

Subscribe



Subscribe

Thank you!

DataStax Developers

Thank you!



Subscribe

