DataStax

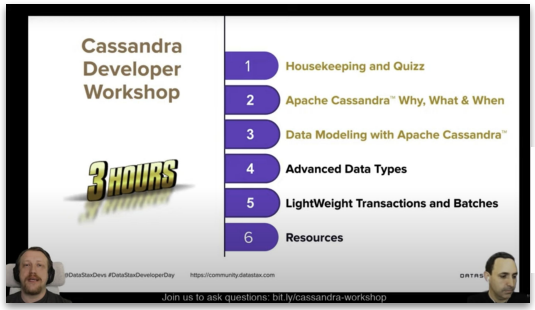
Build a Multiplayer Real-time Game with Astra Streaming

LEVEL UP with the DataStax Developers

Housekeeping #1: Attending the workshop

Live and interactive

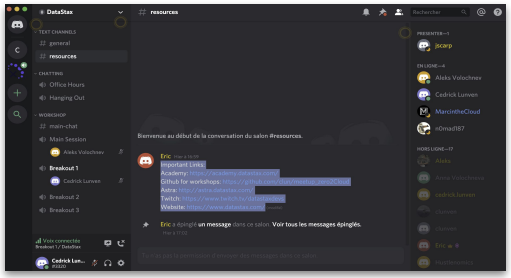
Livestream: youtube.com/DataStaxDevs



YouTube

Twitch

Questions: https://dtsx.io/discord

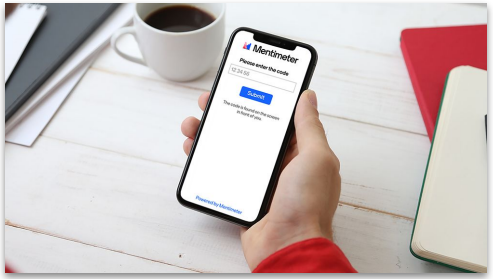


Discord

YouTube

Available on the iPhone App Store

Games menti.com



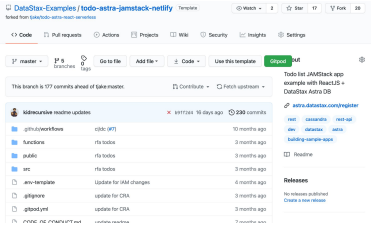
Mentimeter

GET IT ON Google play

Housekeeping #2: Doing Hands-On

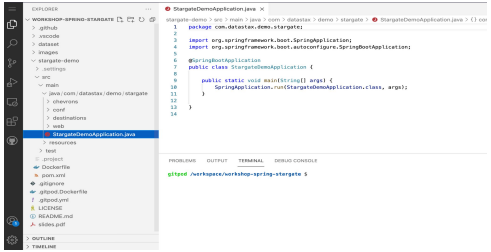
Nothing to install !

Source code + exercises + slides



GitHub

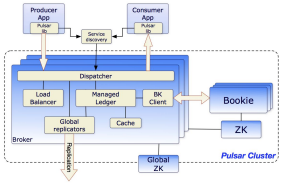
IDE



Gitpod

Messaging system (based on Pulsar)

PULSAR



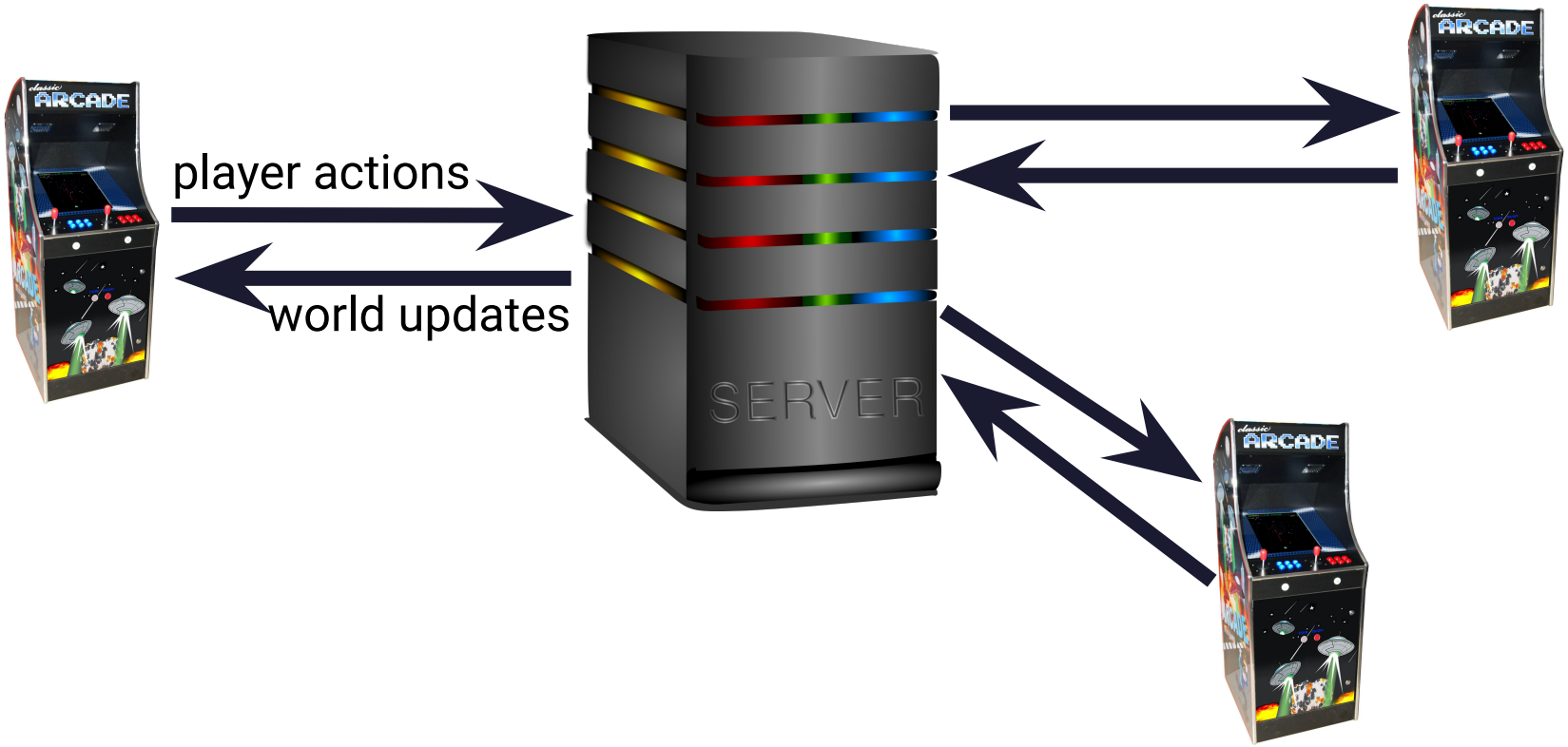
DataStax Astra STREAMING

DataStax Developers

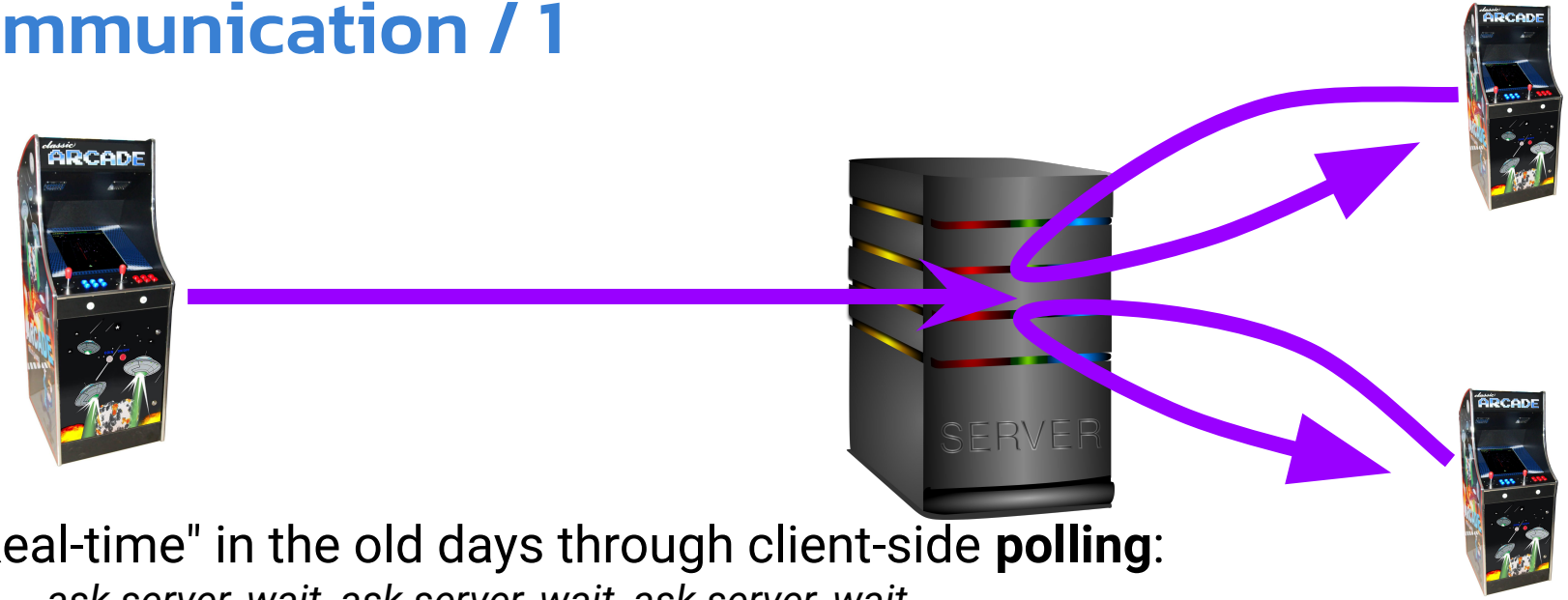
Get your Badge for today !

Let's start!

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



Communication / 1

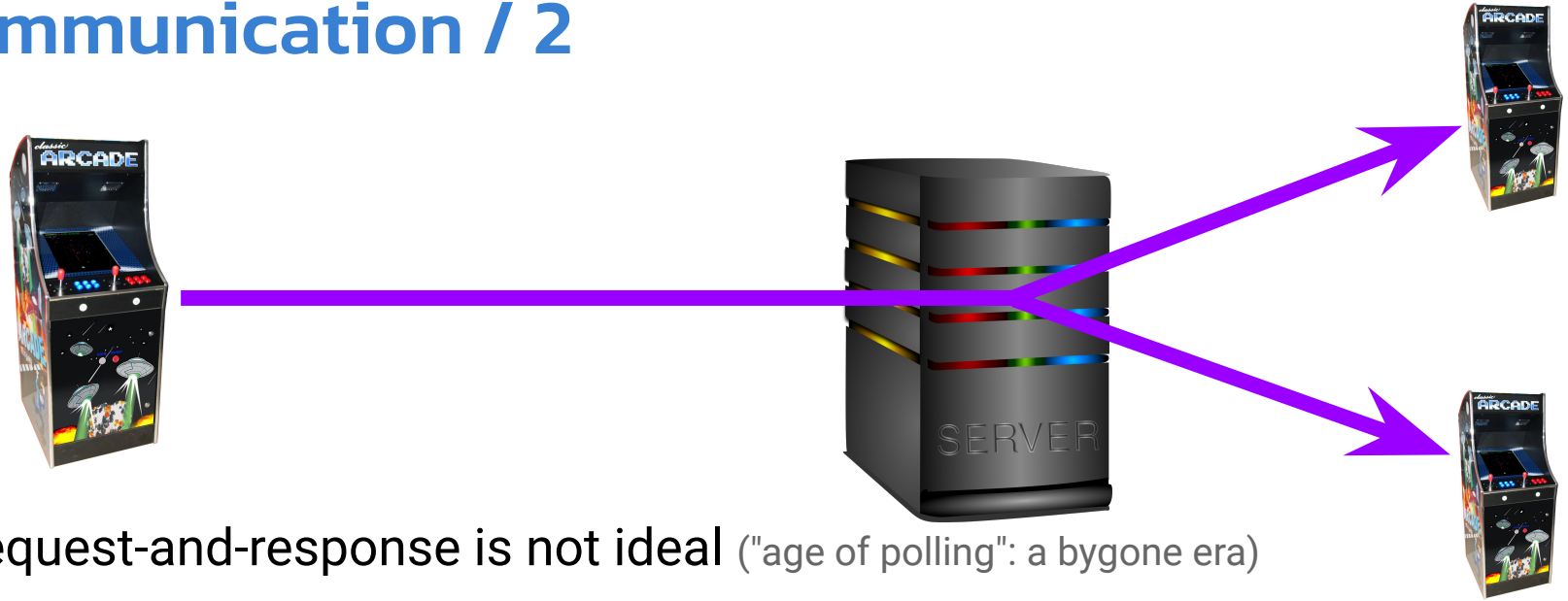


"Real-time" in the old days through client-side **polling**:

ask server, wait, ask server, wait, ask server, wait …

- x A flood of request-and-responses (overhead!)
- x Not really real-time (latencies)
- x Waste of server resources
- x 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, …

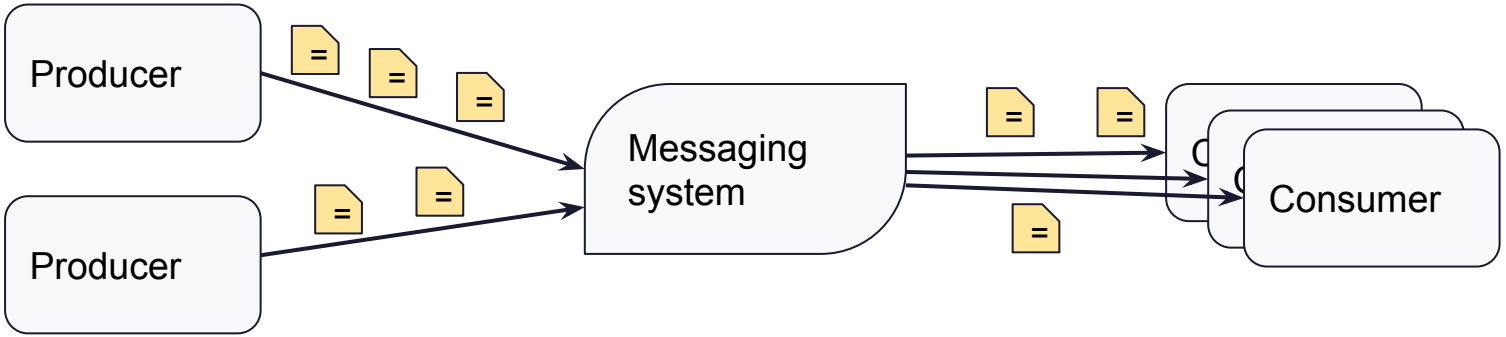
Messaging systems

- **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)
  - …

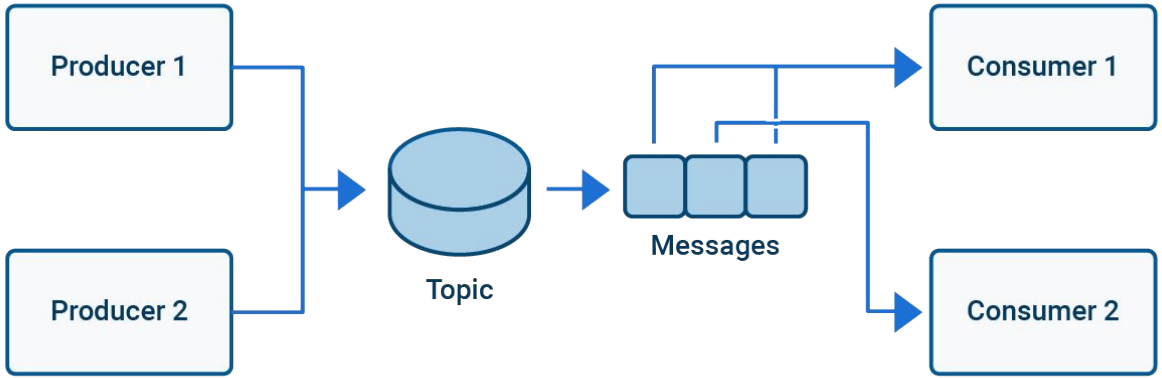
- **In practice:** IoT, social media, car automation, finance, online gaming, ...

Messaging at a glance



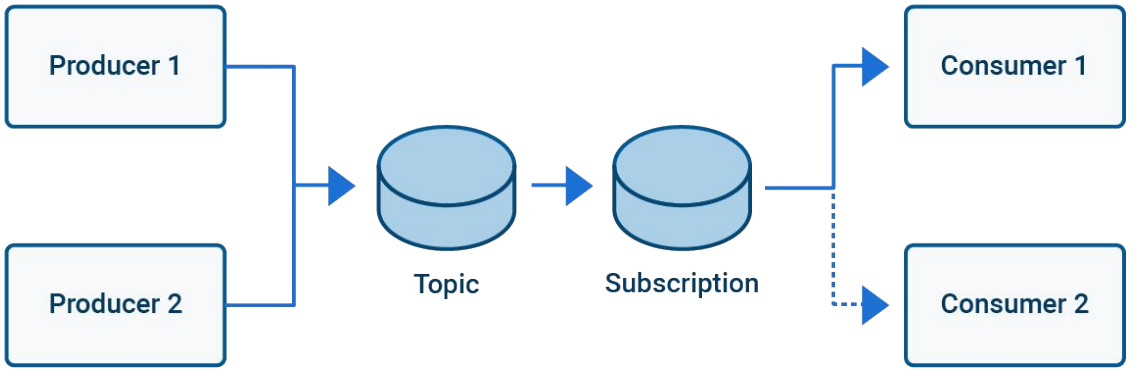
- **Asynchronous ⇒ messages retained**
- **Delivery semantics ⇒ receipt acknowledgement**
  (the hard part is correct delivery)

Message Consumption – Queuing



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

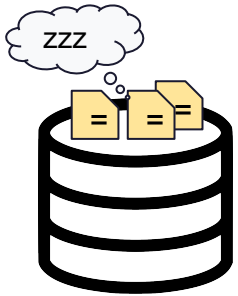
Message Consumption – Pub/Sub



- 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

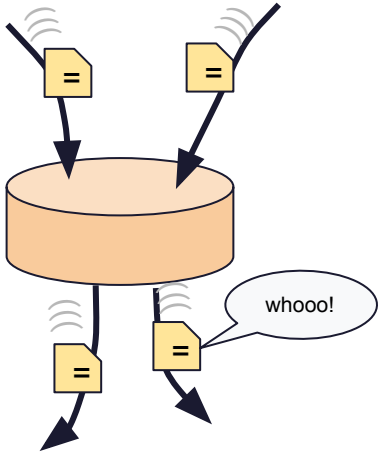
(in "tables/collections")

VS

Messaging

data in transit

(through "topics")



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!

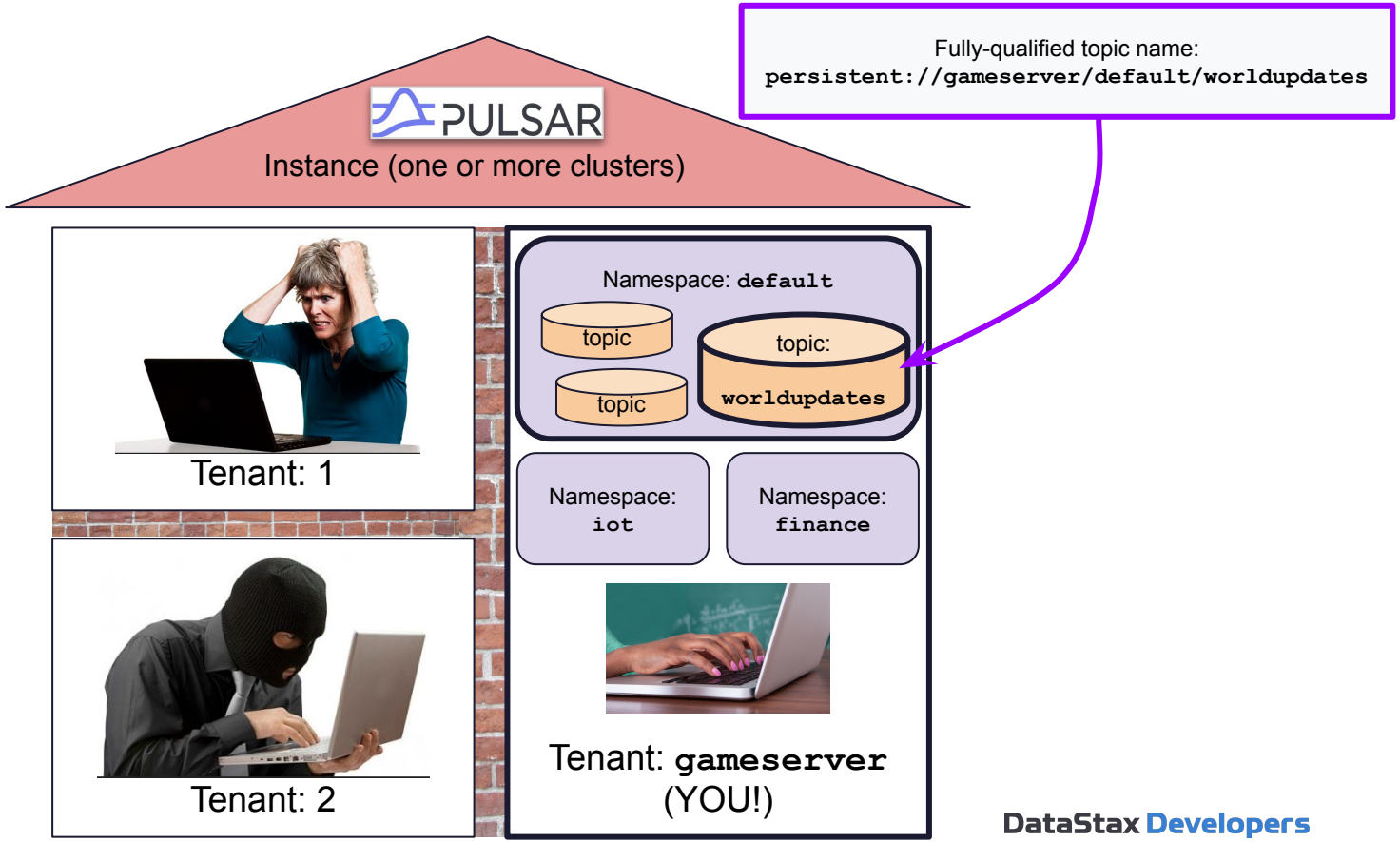
Apache Pulsar ⇒ Astra Streaming

- **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) ...

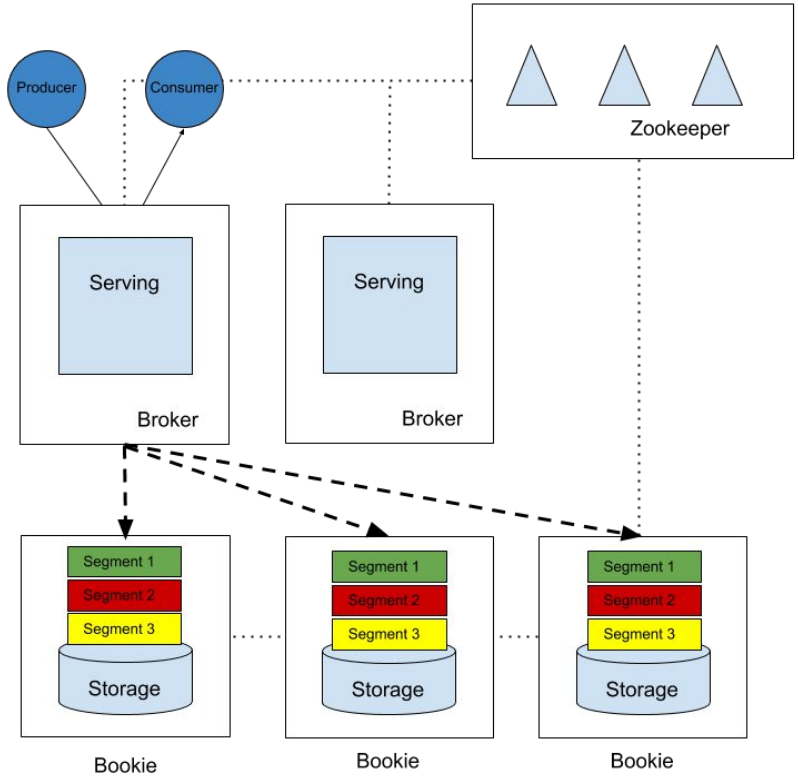
- **Managed Pulsar in the cloud currently FREE open beta**
- Create ready-to-use streaming topics
- Tight integration with Astra DB available *(as between Pulsar and Cassandra)*

Pulsar messaging, logical structure



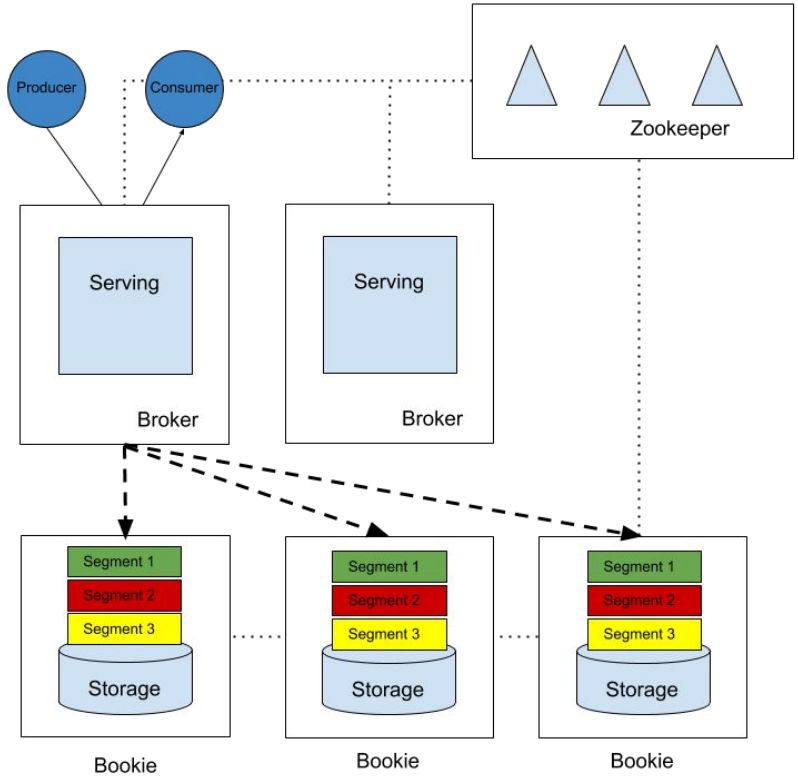
Pulsar architecture

- 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

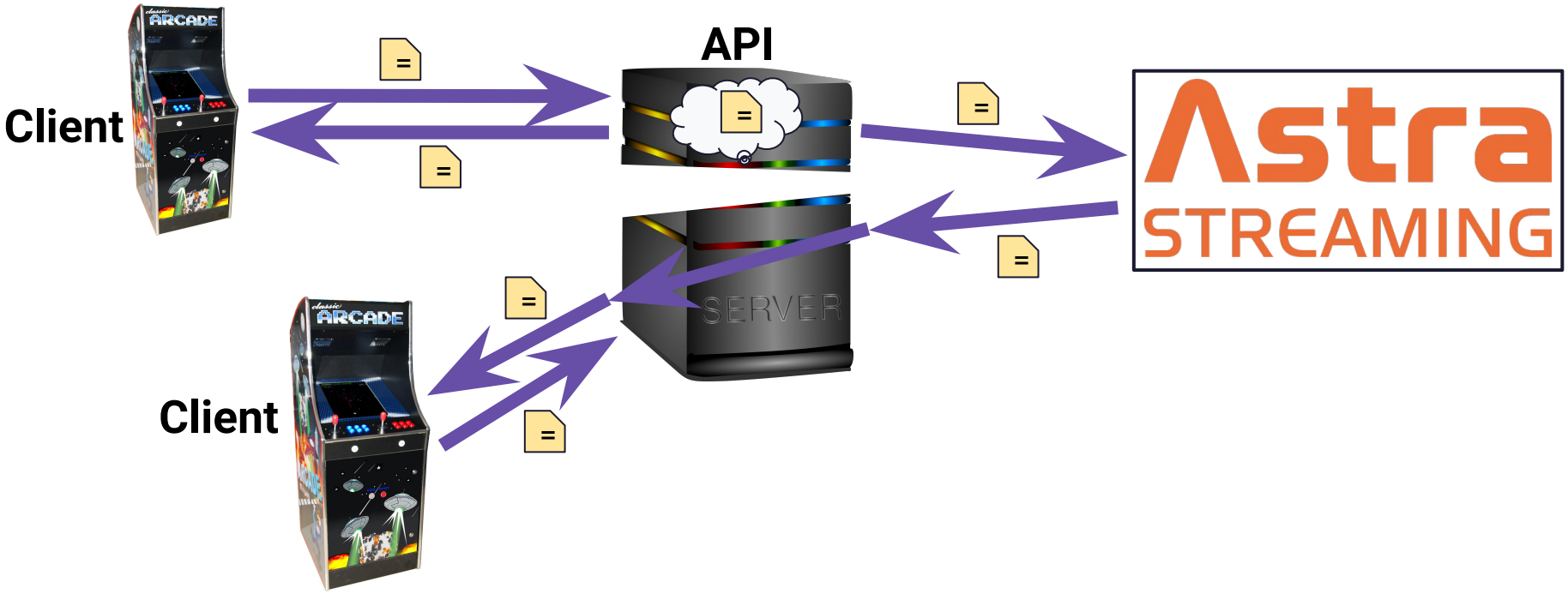
GitHub

Repository:

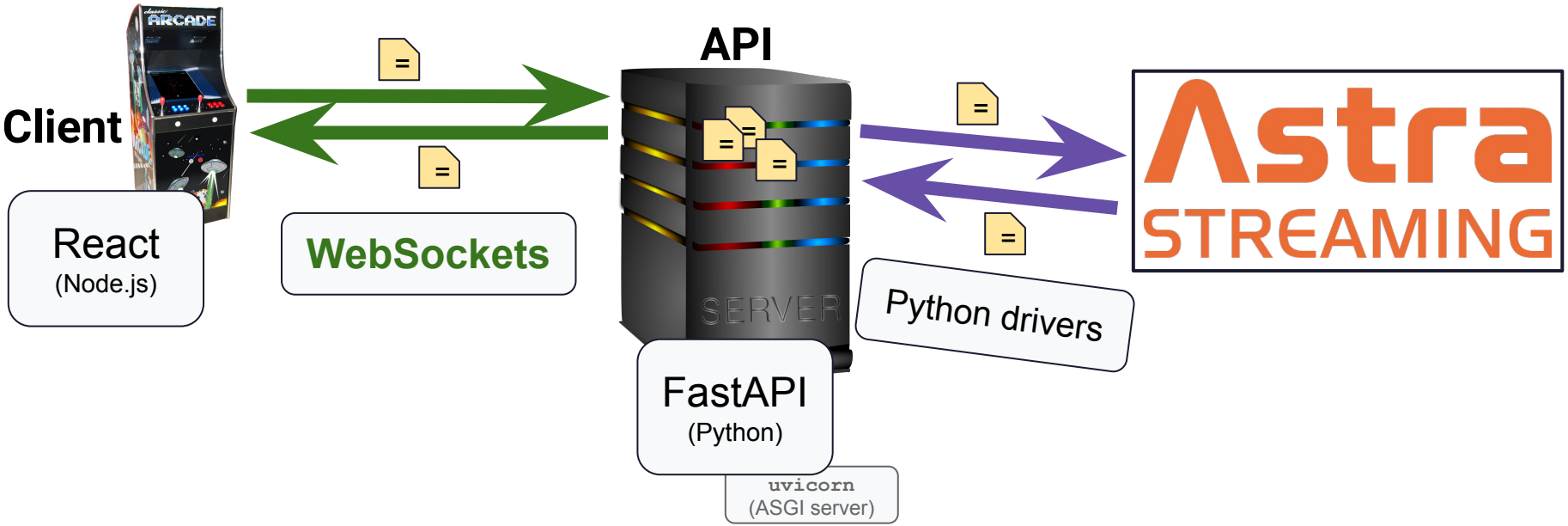
- https://github.com/datastaxdevs/workshop-streaming-game

DataStax Developers

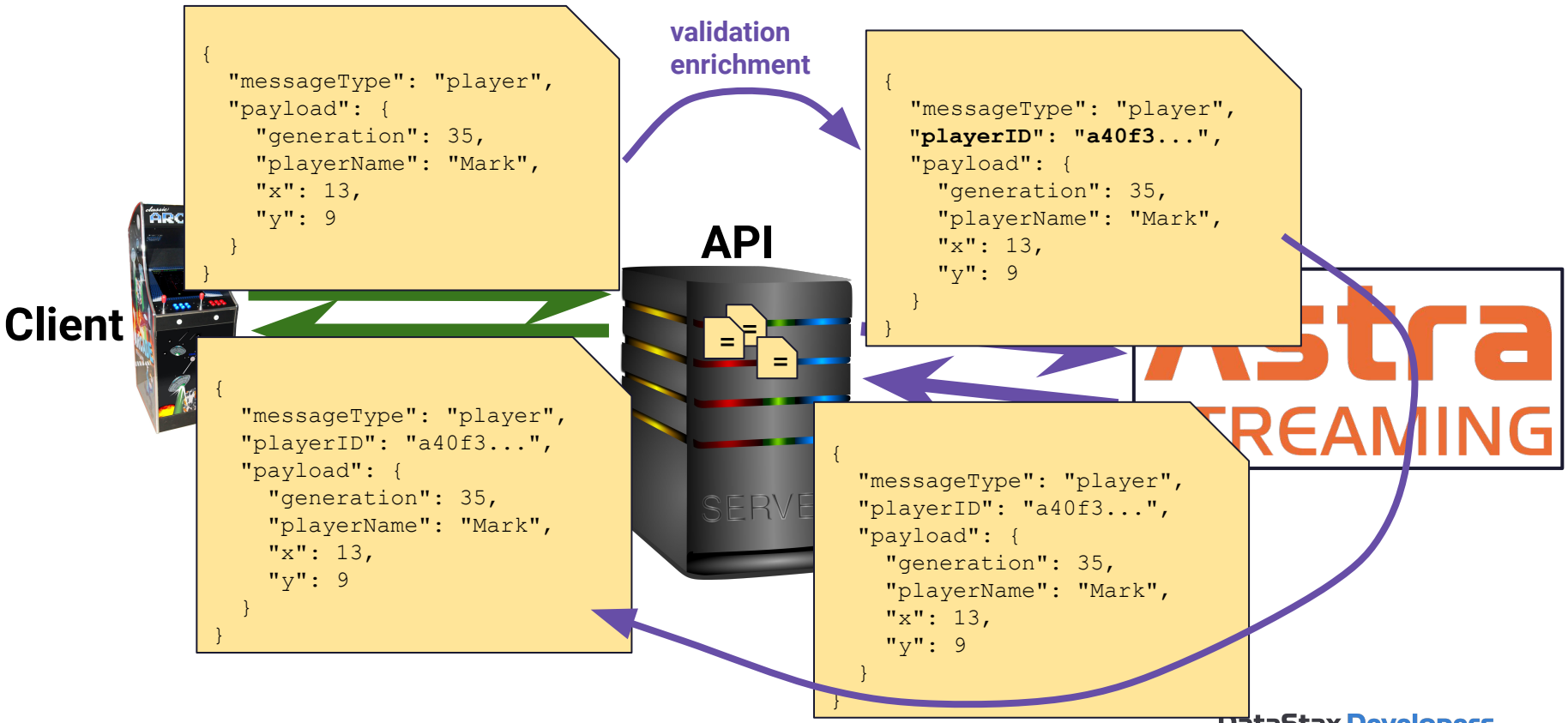
Game architecture, revisited



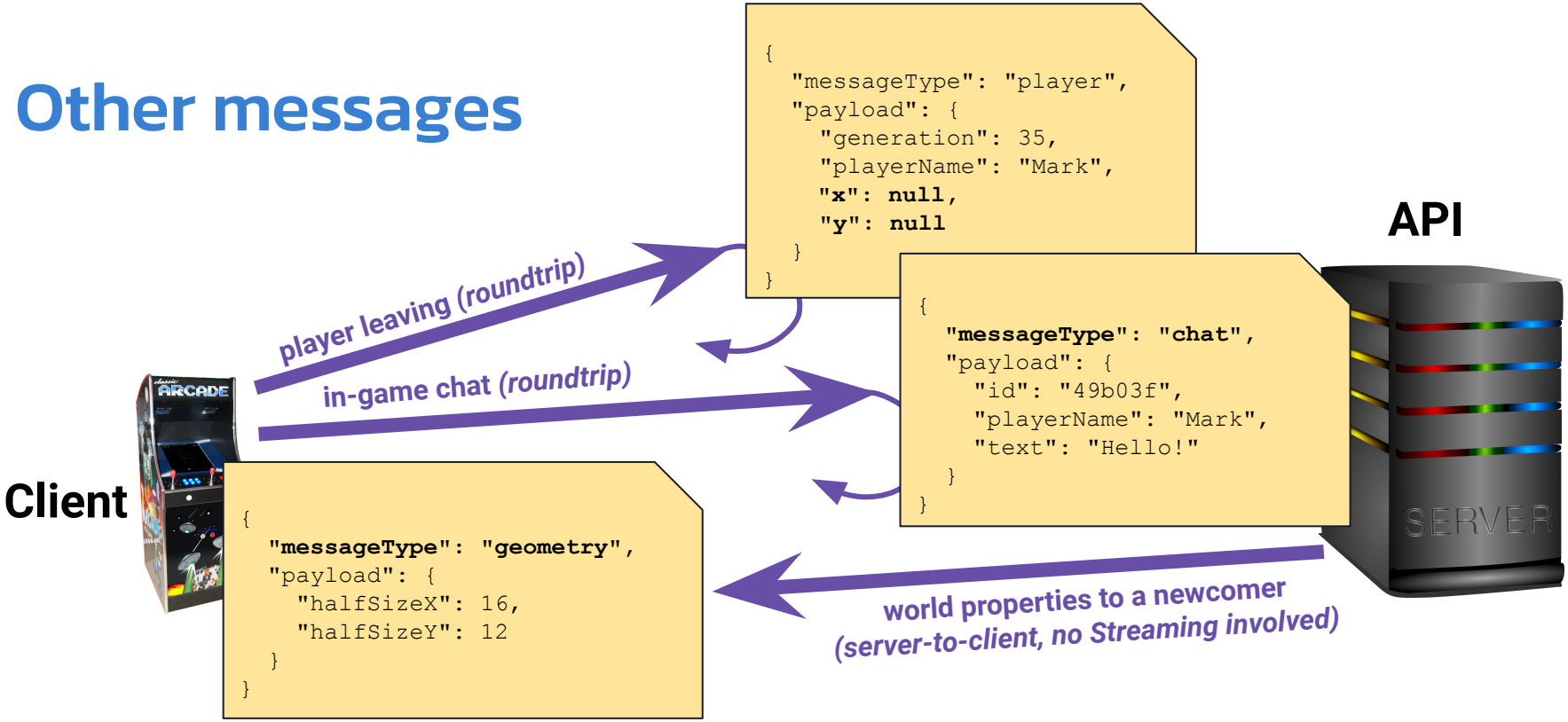
Technologies



Messaging protocol



Other messages



Architectural choices

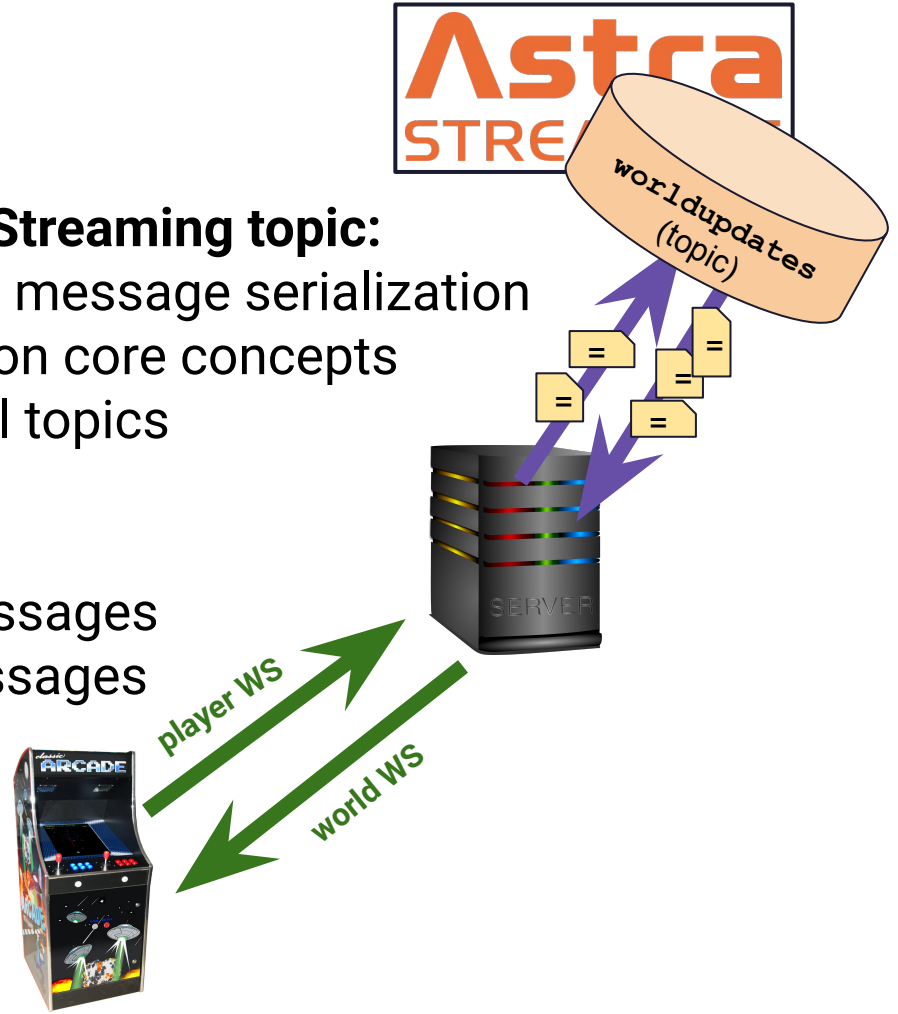
All messages through a single Streaming topic:

- 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

DataStax Developers

When the client starts...

time

connect

initialize x, y
(+ other)

Position: x, y

update
world &
redraw

World
geometry

validation
Player: x, y

receive &
broadcast

worldupdates
(topic)

Gameplay

time

player action
(state change)

Position: x, y

=

validation
Player: x, y

=

worldupdates
(topic)

=

receive &
broadcast

= = =

is it about me?
⇒ update my x, y

update
world &
redraw

In-game chat

time

player types message

Chat: "hello"

=

transport to topic

=

worldupdates (topic)

=

receive & broadcast

= = =

update chat status & redraw

HandsOn: PLAY!

Astra STREAMING

Get your instance here:

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

GitHub

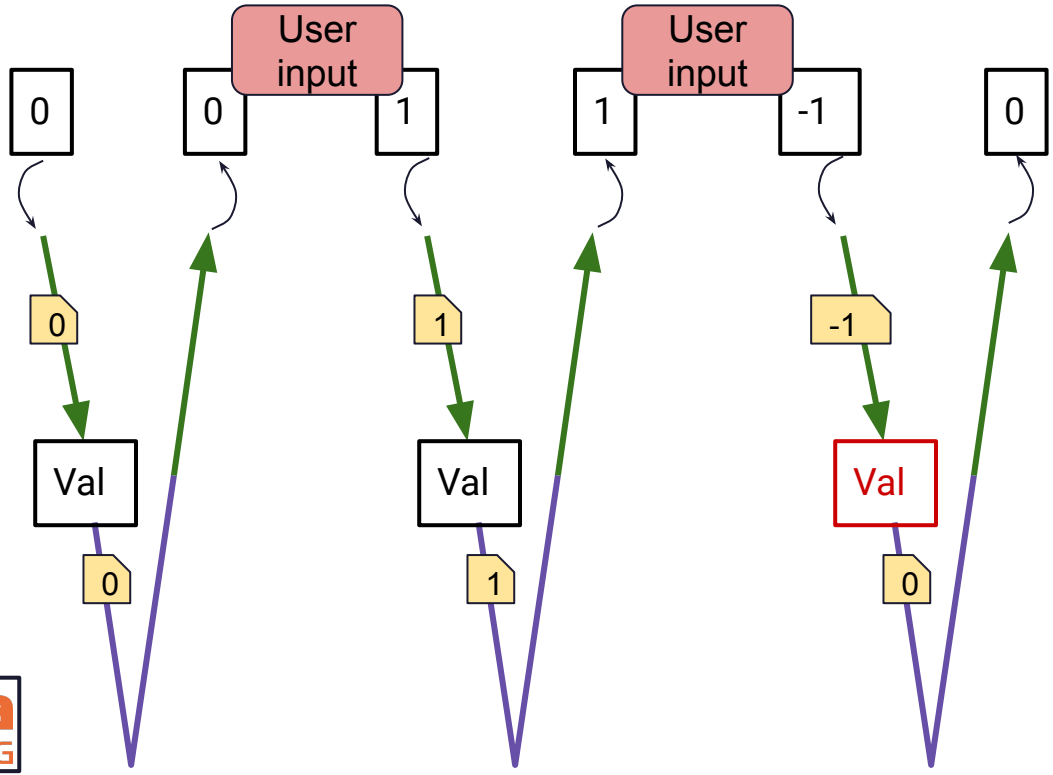
Repository:

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

Async and server-side validation

time

Position is in client memory and validated by the API

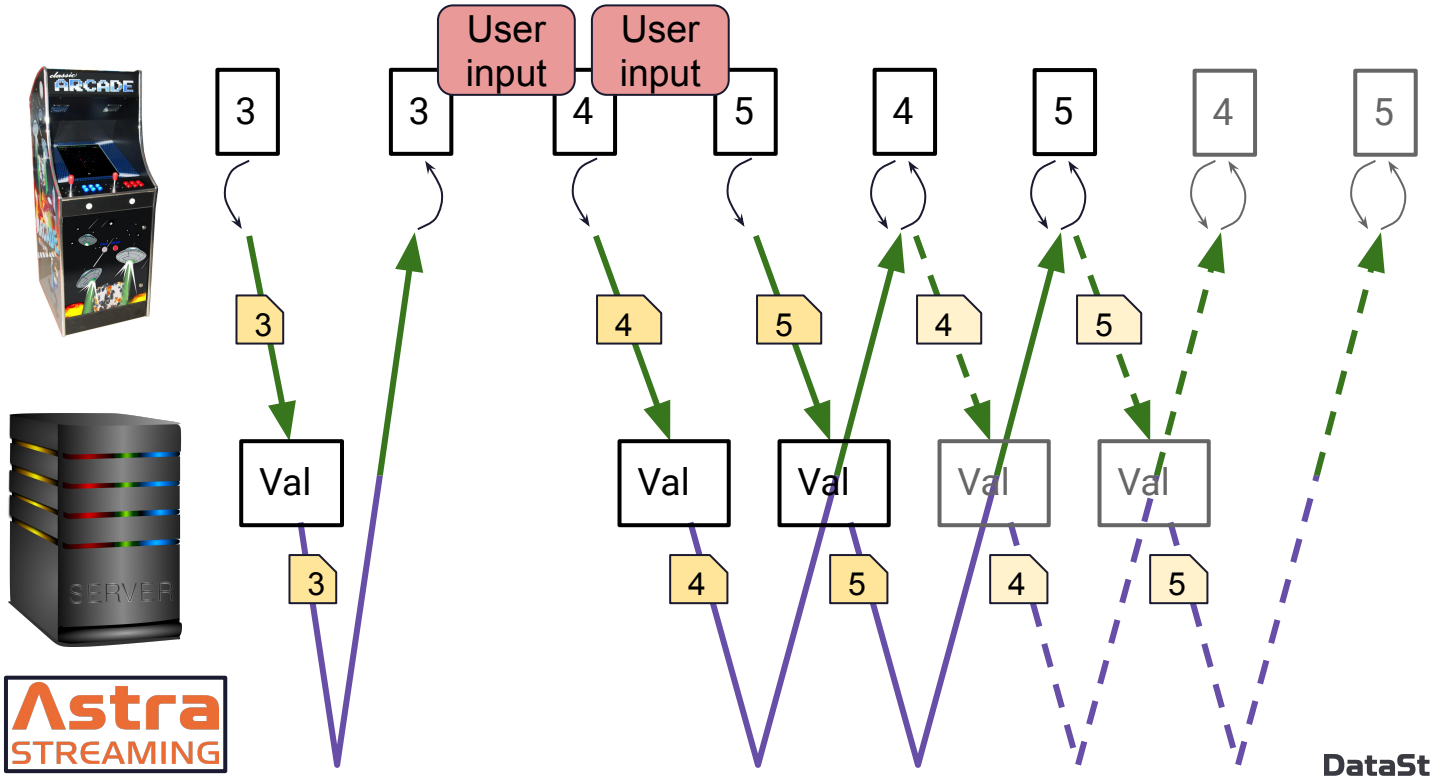


Astra STREAMING

Async and server-side validation / 2

time

Client receives position updates (and may fire new updates)



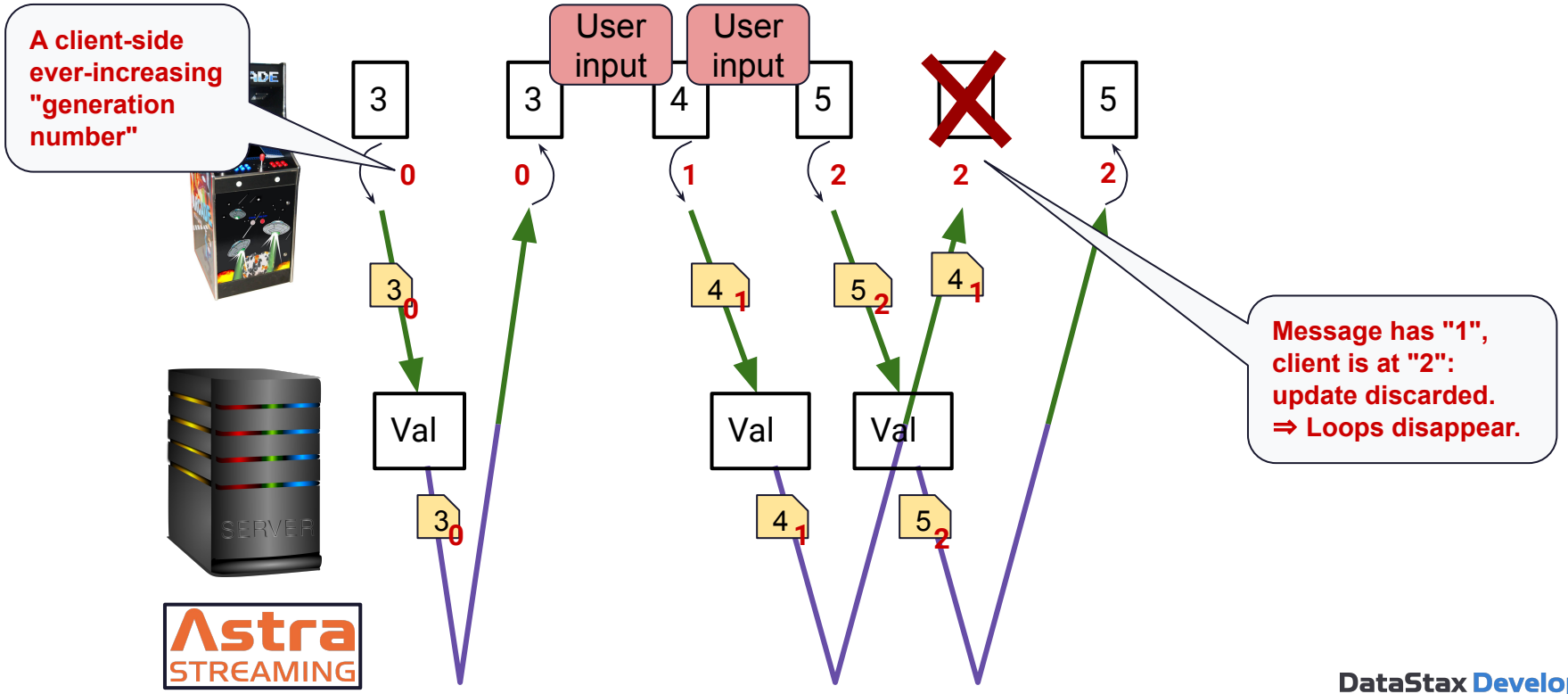
Oh, no!
An infinite loop of spurious updates!

Astra STREAMING

Async and server-side validation / 3

time

Solution: discard **obsolete** server updates



HandsOn: Multiplayer

Astra STREAMING

Get your instance here:

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

GitHub

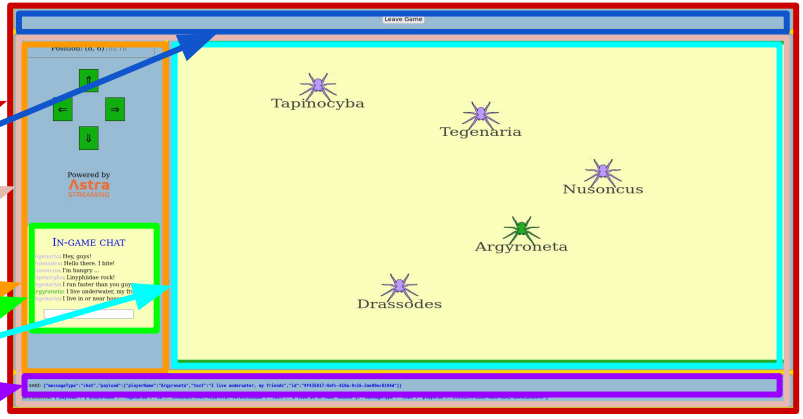
Repository:

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

DataStax Developers

Client app structure

```
1  let pws = null
   let wws = null
2  const App = () => {

     // React states (useState)
3    // React references (useRef)

     // keyDown handler

4    useEffect( () => {
       if (inGame) {
         if ( websockets === null) {
           // create WebSockets
5          websockets.onmessage = evt => {
             // parse and interpret message
             // depending on messageType (player/geometry/chat)
             // HERE WE MODIFY REACT STATE
           }
         } else {
           // send "goodbye" to API
6          websockets.disconnect()
           webockets = null
         }
       }
     }, [inGame])

     useEffect( () => {
7      // send player update to the API
     }, [coordinates, etc])

     return (
8      App
         PlayerForm
         GameArea     // if inGame
           sidebar
             GameInputs
               buttons, etc
               ChatArea
           GameField
           statusbar
     )

   }
```



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

- 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?

```python
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())
                pulsarConsumer.acknowledge(worldUpdateMsg)
            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 ...

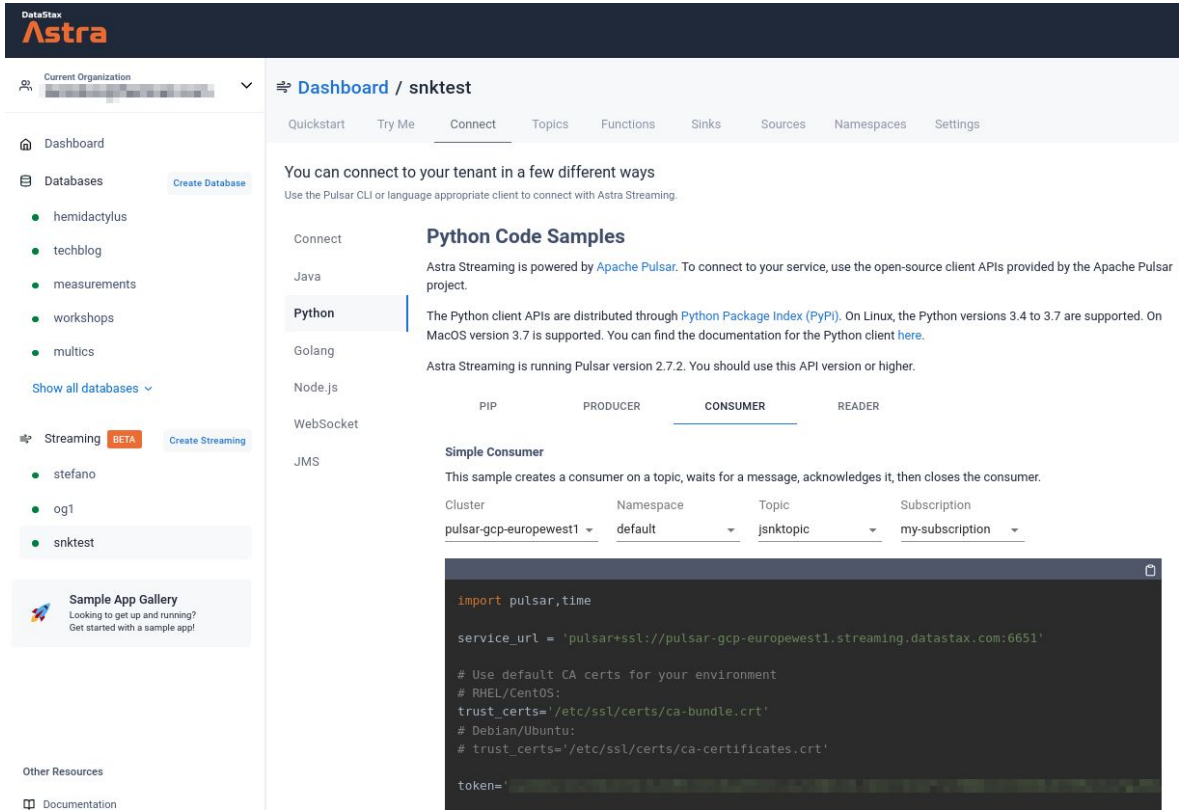
notify successful processing

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



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(p
        if worldUpdateMsg is not None:
            # We forward any update from
            # for all clients out there:
            await worldWS.send_text(worl
        await asyncio.sleep(SLEEP_BETWEE
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

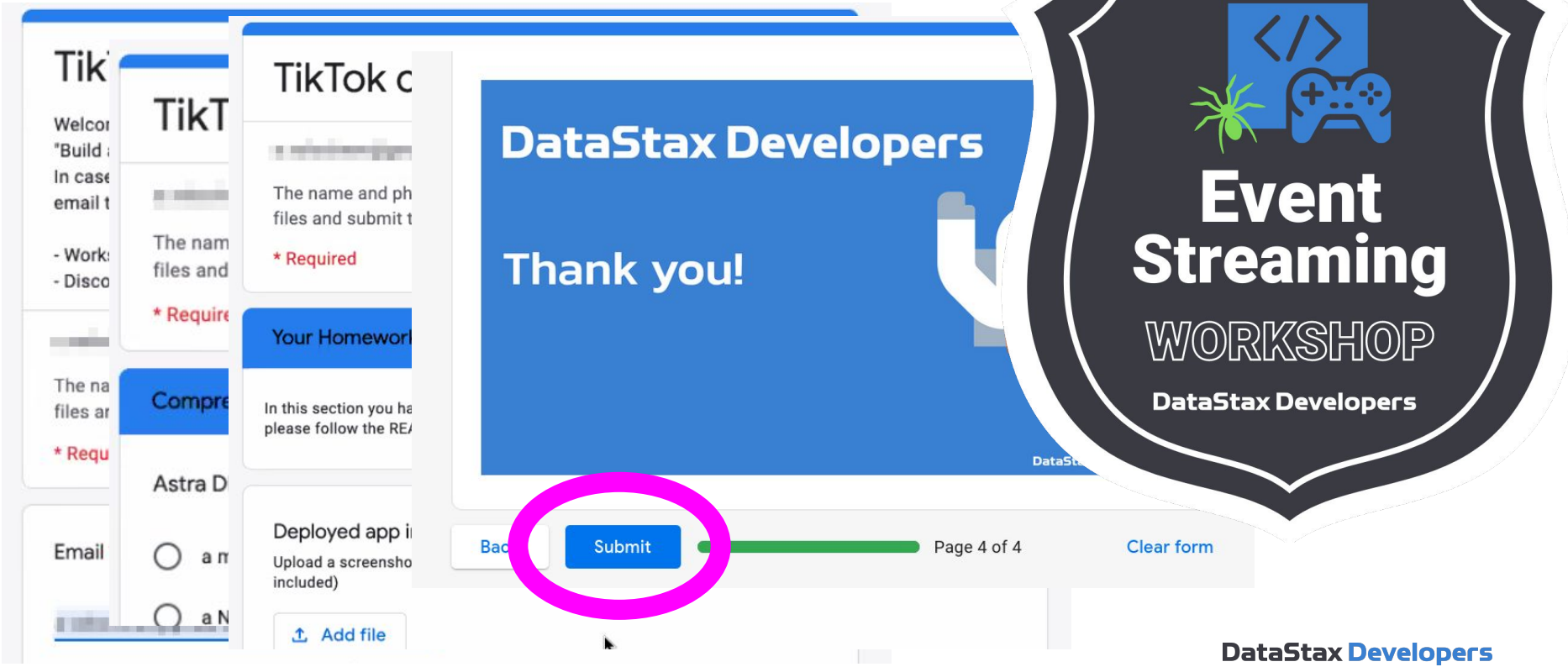
```
const GameField = ({playerMap, playerID, boardWidth, boardHeight}) => {

  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})`}></rect>
            {thatPlayerInfo.h && <rect x="-50" y="-50" width="100" height="100" fill='url(#hearts)'></rect>}
            <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

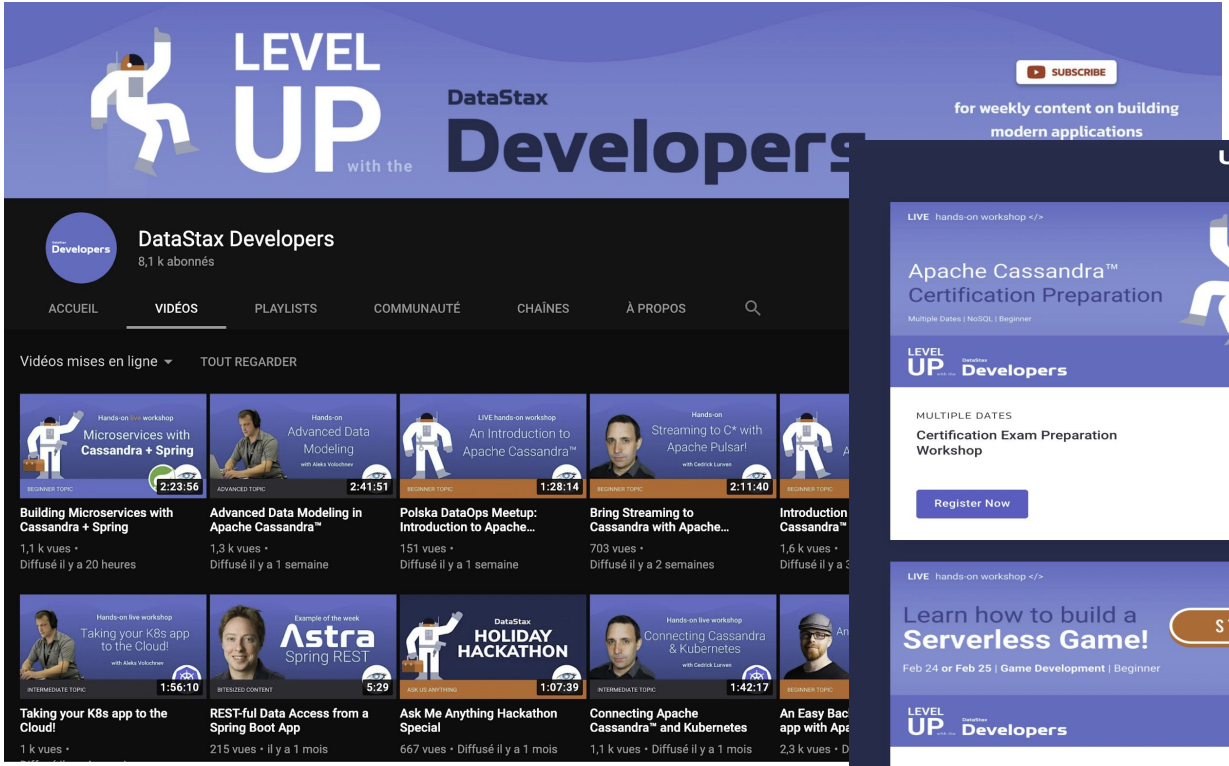
How to submit Homeworks

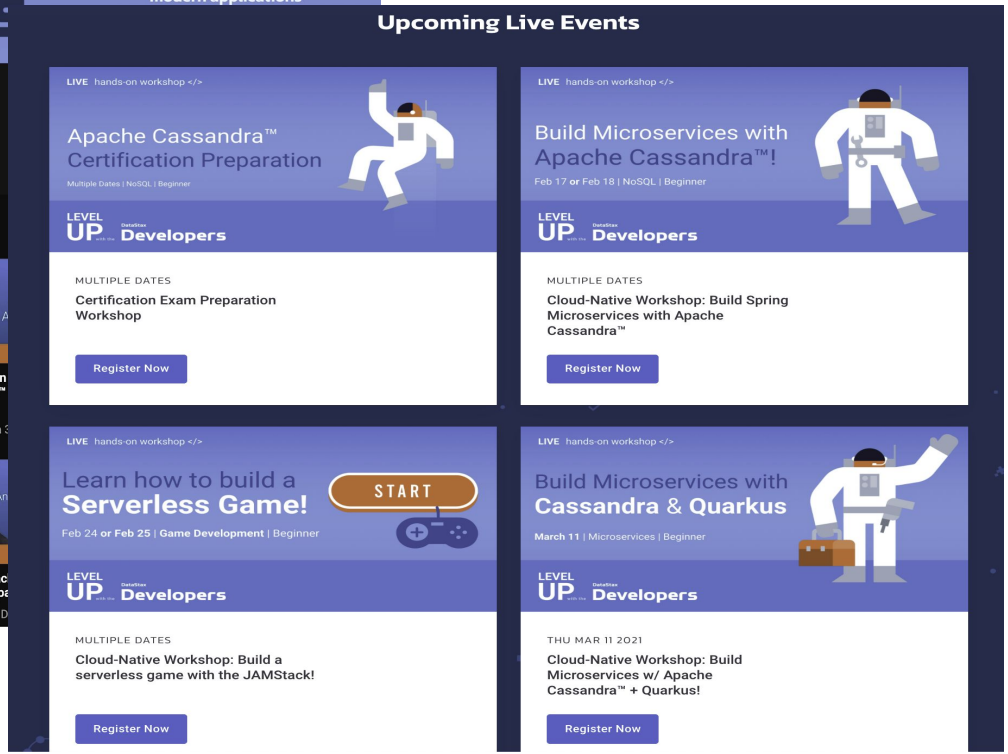
Link to submission form in the README



Weekly Workshops

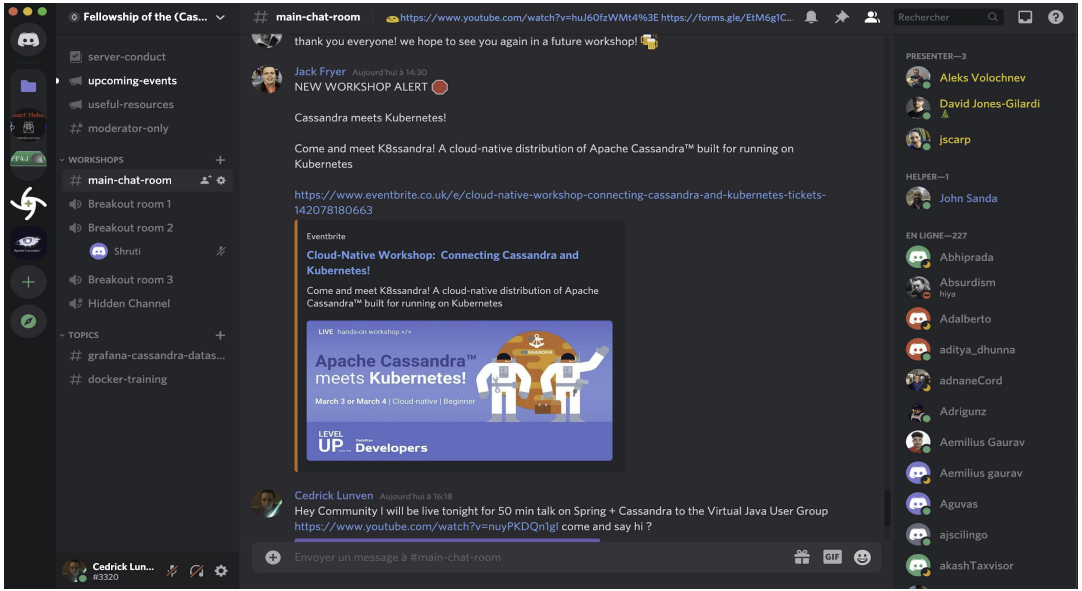
datastax.com/workshops





Join our 17k Discord "DataStax Developers" Community

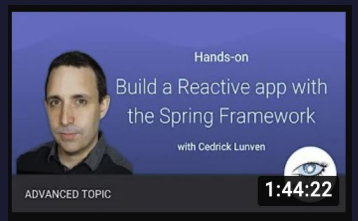
dtsx.io/discord



DataStax Developers

Subscribe

Thank you!

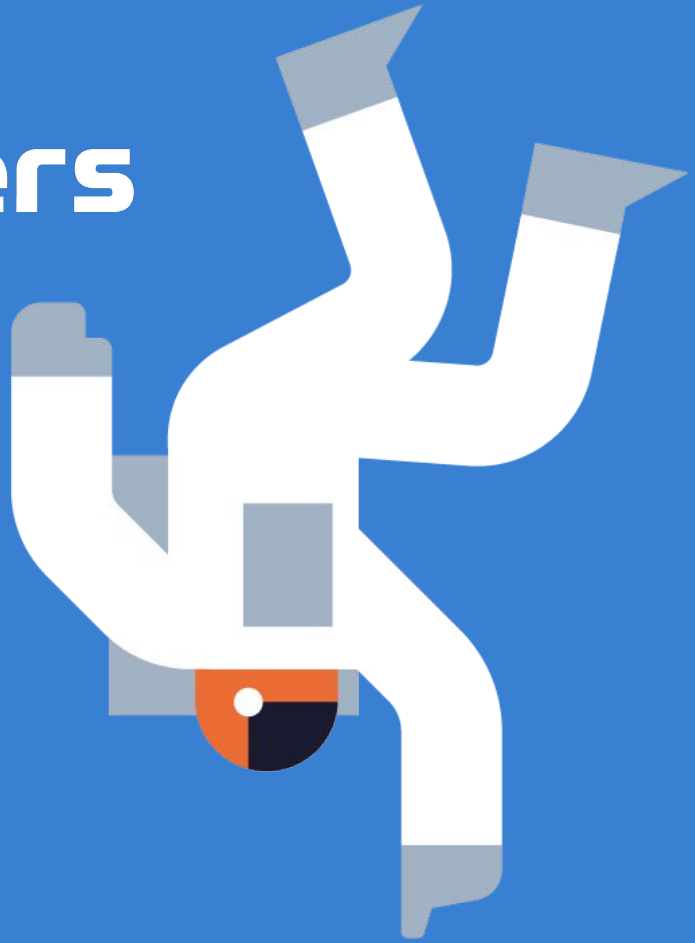


Thank you!

Subscribe

DataStax Developers

Thank you!



Subscribe